Verification of Rust Cryptographic Primitives with Aeneas

Aymeric Fromherz, Son Ho, Jonathan Protzenko

Implementing Cryptography is Tricky

- Implementations must be extremely fast
- Security constraints: constant time operations to avoid leaking secret keys, etc.
- Complex state machines

⇒ Subtle, low-level implementations

Bug in amd-64-64-24k Curve25519:

"Partial audits have revealed a bug in this software (`r1 += 0 + carry` should be `r2 += 0 + carry` in amd-64-64-24k) that would not be caught by random tests"

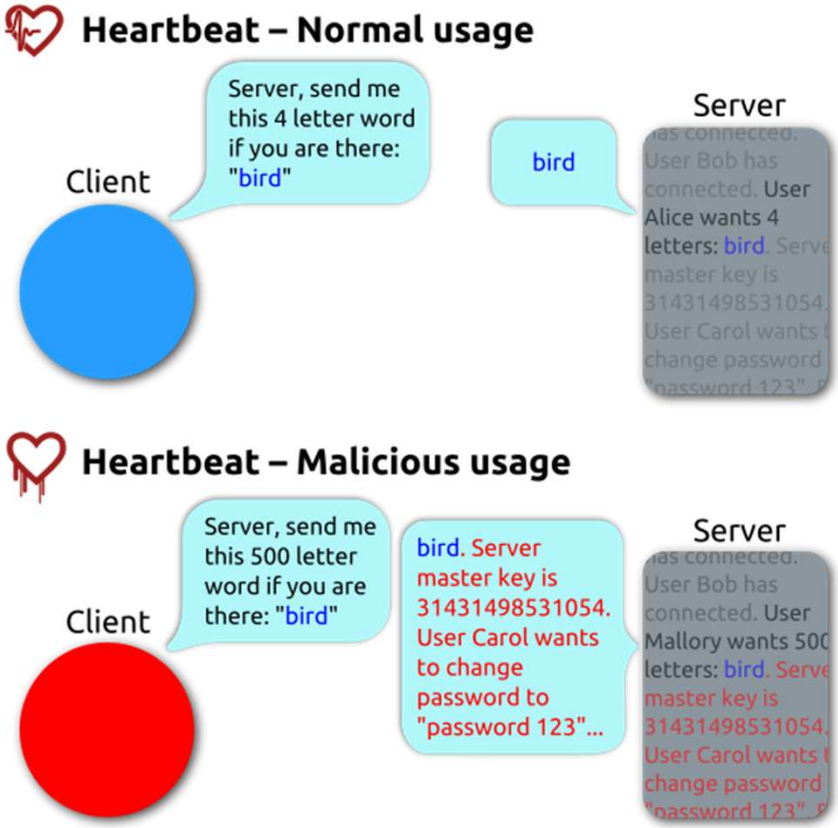
– D.J. Bernstein, W.Janssen, T.Lange, and P.Schwabe

Bug in ref. impl. of SHA-3 (streaming state machine):

A call to `Keccak_HashUpdate()` of $0 < x < $ `rateInBytes` bytes followed by a call of $2^{32} - x$ bytes will conveniently result in another integer overflow: `instance->byteIOIndex` will overflow and end up with a value of zero. Therefore, from the point of view of the implementation, the 4 GiB message is "forgotten" and the computation continues as if nothing has been processed yet.

Now, the padding of SHA-3 becomes relevant. As explained in [12], the padding consists of adding a fixed two- or four-bit suffix to the message (to

Heartbleed (CVE-2014-0160)



wikipedia.org

- Missing bound check during a memcpy

```
response = malloc(length);
memcpy(response, recv.heartbeat, length);
```

```
response = malloc(length);
if length > ssl_state.heartbeat {return 0;}
memcpy(response, recv.heartbeat, length);
```

What Can Go Wrong?

- Big gap between theoretical cryptography and implementations
  - Choice of data representation
  - Handling memory and data storage
  - Fine-grained optimizations

- Very easy to make mistakes!

A Concrete Example: Modular Arithmetic

- Modular arithmetic is frequently used in cryptographic primitives

$$(a * b)\ mod\ p$$

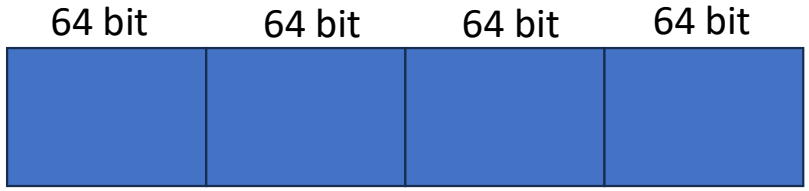
p is frequently a large prime number
(e.g., $2^{255} - 19$)

- Mathematical field operations in $\mathbb{Z}/p\mathbb{Z}$

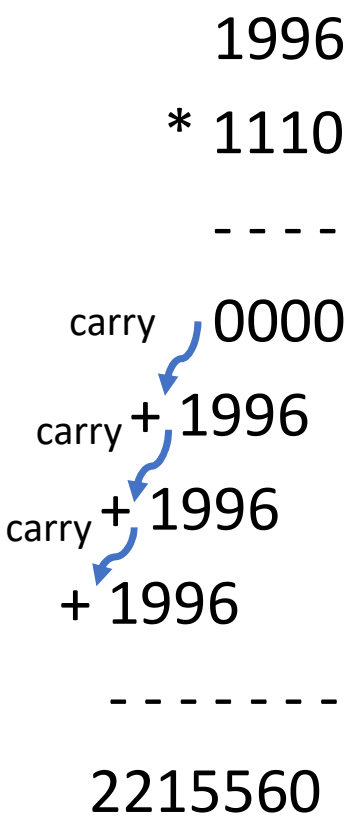
Implementing Modular Multiplication

$$(a * b)\ mod\ p$$

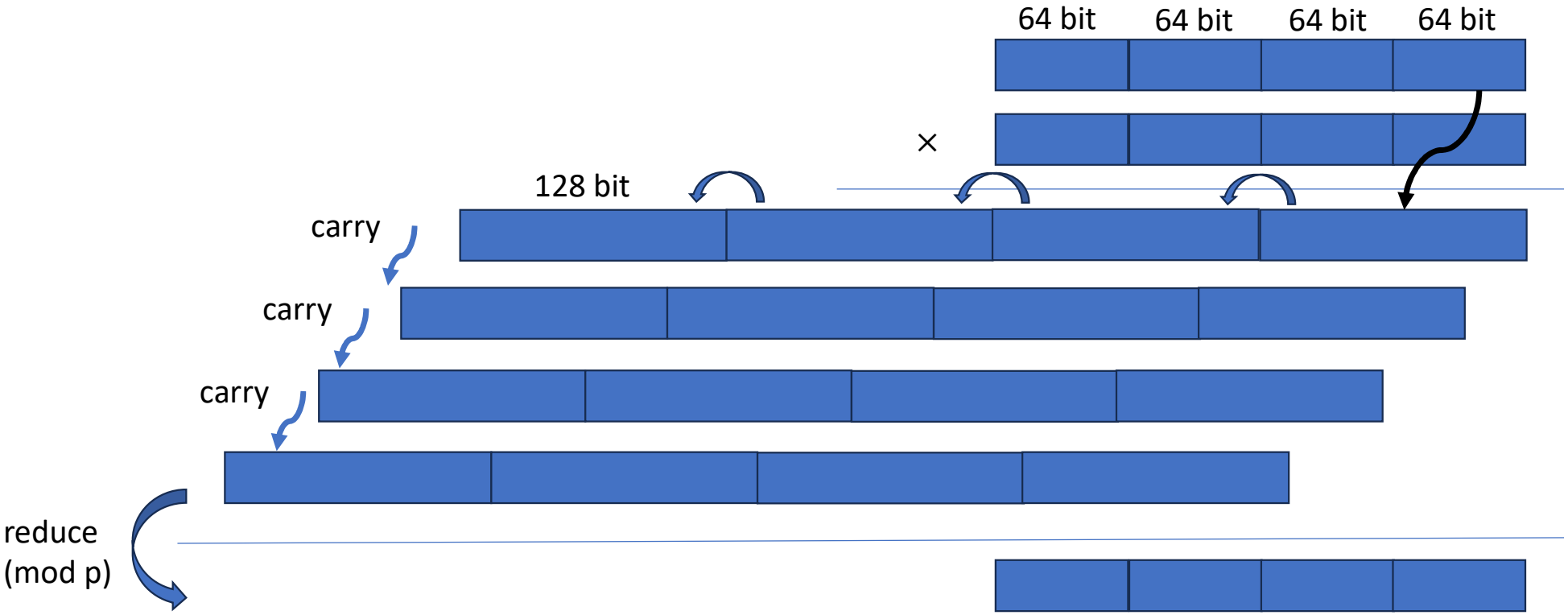
- a, b are big integers (e.g., $2^{255} - 19$)
- Multiplication is even bigger
- Machine integers are (at most) 64 bits, accepting numbers up to 2^{64}
- How to implement this? Need a bignum library



Textbook Multiplication



Implementing Modular Multiplication



Implementing Modular Multiplication

What can go wrong?

- Integer overflow (undefined output)
- Buffer overflow/underflow (memory error)
- Missing carry steps (wrong answer)

- And that's without optimizations!
  - Parallelize multiplications
  - Use alternative reductions (Montgomery, Barrett)
  - Precompute intermediate reusable values
  - …

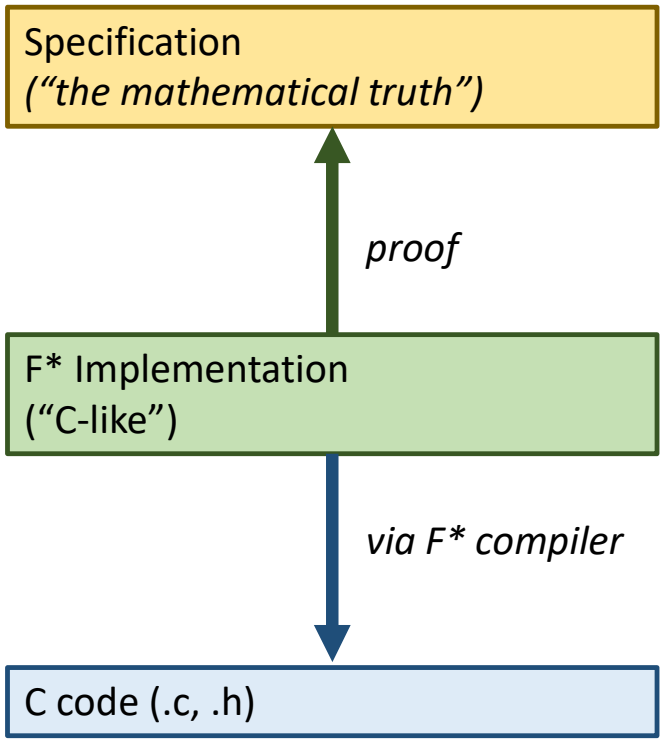
Formal Verification to the Rescue

Critical software needs **guarantees** of its **correctness** and **security**

Formal verification allows to mathematically reason about the correctness and safety of programs

Many past applications to compilers, operating systems, railway and aeronautical embedded systems, …

Program Verification Workflow



- **Ground truth:** Must be as simple and easy to review as possible
- Picks a data representation, implements algorithmic optimizations
- Executable code, integrated in secure applications

Back to Bignum: Specification

Specification
("the mathematical truth")

```
let prime = pow2 255 – 19

let felem = x: int{x <= 0 && x < prime}
let fadd (x: felem) (y: felem) : felem = (x + y) % prime
let fmul (x: felem) (y: felem) : felem = (x * y) % prime
```

Bignum Data Representation

F* Implementation
("C-like")

```
type bignum = b:array uint64{length b = 4}

let feval (m: mem) (b: bignum) : Ghost Spec.felem =
  let s = as_seq m b in
  (s.[0] + s.[1] * pow2 64 + s.[2] * pow2 128 + s.[3] * pow2 192) % Spec.prime
```

- feval allows to retrieve the corresponding mathematical number

Implementing Field Arithmetic

F* Implementation
("C-like")

```
val fmul (a b : bignum) : Stack unit
    (requires λ init → live init a && live init b &&
                       disjoint a b)
    (ensures  λ init _ end → modifies (loc a) init end &&
                             feval end a == Spec.fmul (feval init a) (feval init b)
    )
```

This guarantees:

- Memory Safety
- Functional Correctness

We can apply this more broadly to entire cryptographic constructions!

The HACL* Crypto Library

Microsoft Research

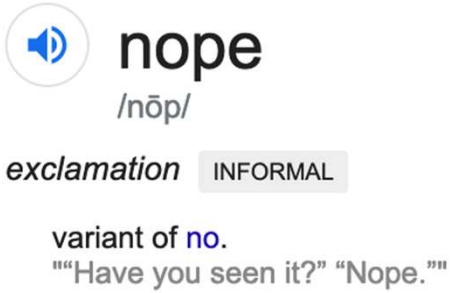
- HACL*: A verified, comprehensive cryptographic provider
- Provides guarantees about memory safety, functional correctness, resistance against side-channels
- ~150k lines of F* code compiling to ~100k lines of C (and Assembly) code
- 30+ algorithms (hashes, authenticated encryption, elliptic curves, …)
- Integrated in Linux, Firefox, Python, and many more

COMPUTER SECURITY

Cryptography That Can't Be Hacked

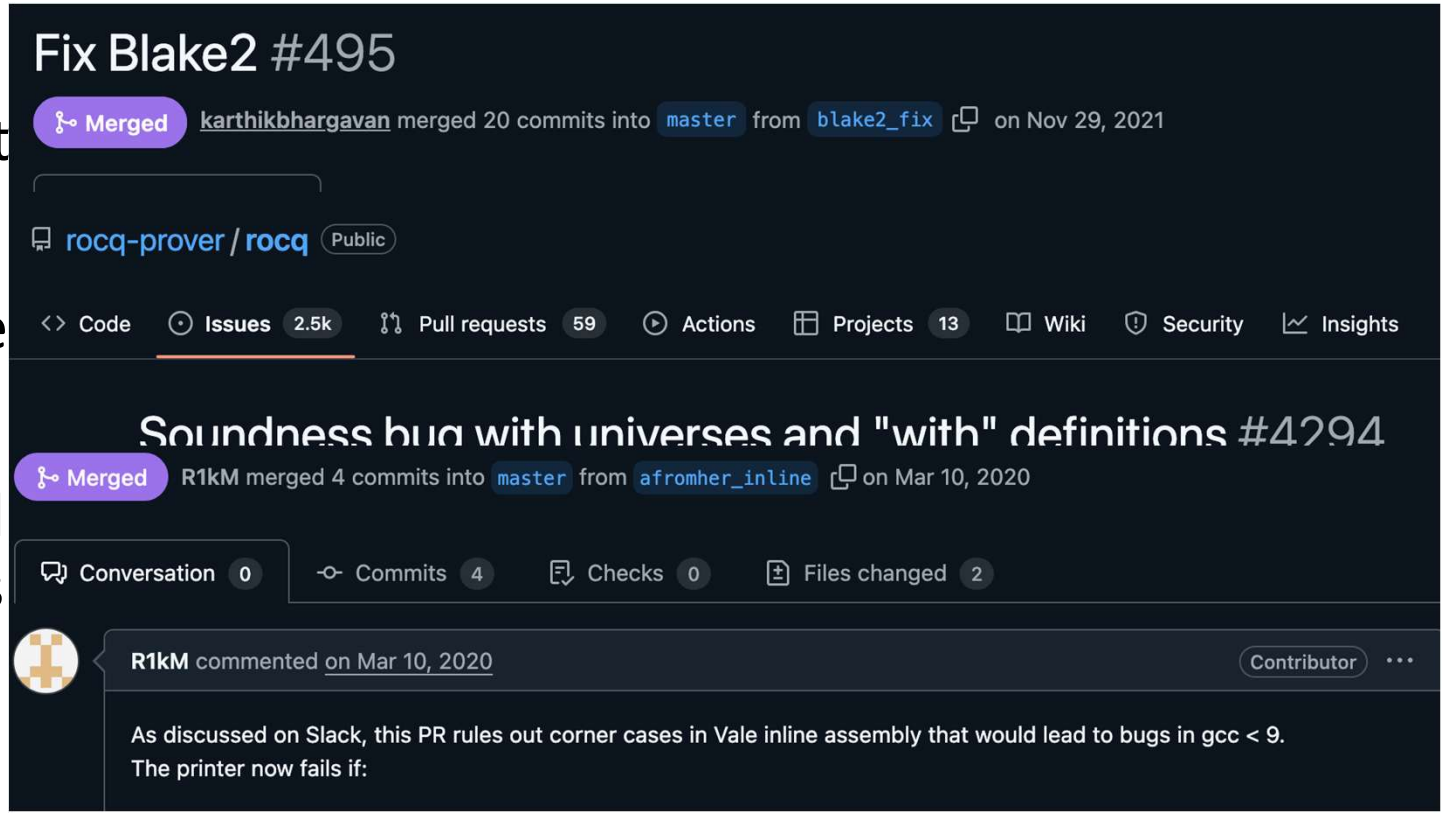
20 |

Researchers have just released hacker-proof cryptographic code — programs with the same level of invincibility as a mathematical proof.



Limits of Formal Verification

- We trust the spec to accurately model the RFC
- We trust our verificat
- We trust our compile
- Formal verification al
  our implementations



Challenge: Usability

- Reliance on specific, uncommon languages (F*, Coq, …)
- Requires deep expertise in formal methods to be usable
- Researcher-oriented toolchains

- How to democratize the use of formal methods?
- How to better integrate formal methods in development processes?

Challenge: Verification Tools Diversity

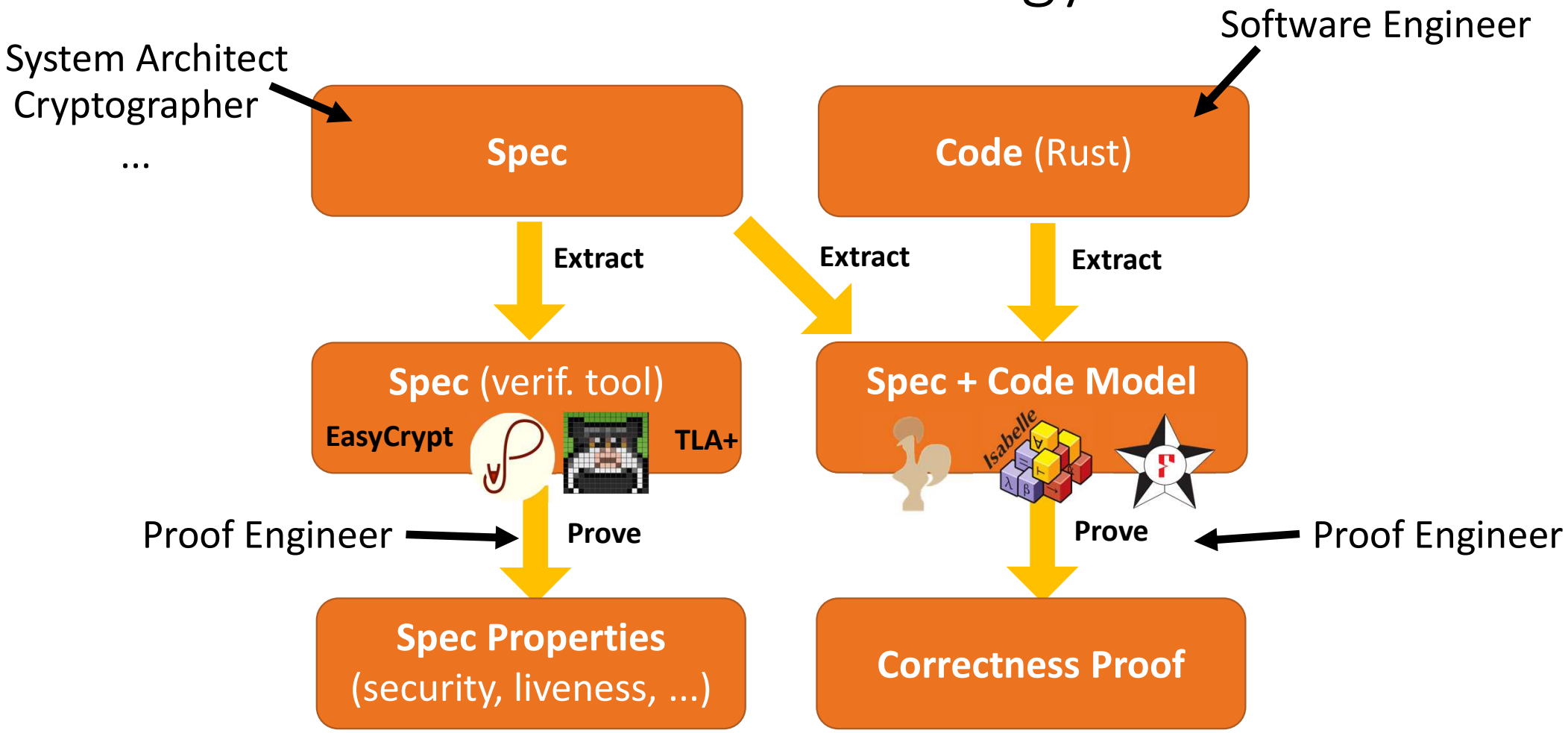
- Variety of tools for different uses
  - Generic proof assistants (F*, Coq, Lean, …)
  - Cryptographic protocol verifiers (ProVerif, CryptoVerif, DY*, EasyCrypt, …)
  - Specification model checkers (TLA+, …)
- Little interoperation between tools
- Specialized expertise, uncommon to master several tools
- How to support a multitude of tools for diverse usecases?
- How to empower teams with different proof expertises?

Challenge: Scalability

- Memory reasoning in C/C++ is tricky
  - Aliasing, liveness, memory safety…
- Need complex models or logics
  - Dynamic frames, separation logic
- Tedious and time-consuming, limits complexity of studied programs
- Distracts from “core” parts of verification

- How to simplify memory reasoning, and provide custom automation for different classes of programs?

A Novel Verification Methodology



Rust Overview

- At the forefront of “Safe Coding” development advocated by governments
- Adoption into Windows, Linux, Android, …
- High-level language, with type polymorphism and “typeclasses” (traits)
- Ownership-based type system, ensuring memory safety

```
fn swap (x: &mut u8, y: &mut u8)
```

- Explicit data mutability, allows aliasing for immutable borrows
- Also provides low-level (C-like) idioms through ***unsafe*** escape hatch

Aeneas: Translating safe Rust to Pure Code

Rust:

```
fn incr(x : &mut i32) {
    *x = *x + 1;
}


fn main() {
    let mut x = 0;
    incr(&mut x);
    incr(&mut x);
    incr(&mut x);
    assert!(x == 3);
}
```

Translation:

```
let incr (x : i32) : i32 = x + 1
```

```
let main () =
    let x = 0 in
    let x = incr x in
    let x = incr x in
    let x = incr x in
    assert (x == 3)
```

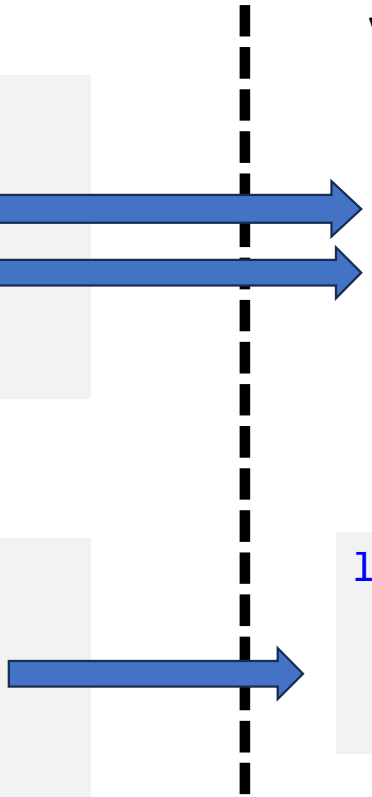
Translating safe Rust to Pure Code: Advantages

C:

```
void main() {
  int x, y = 0;
  incr(&x);
  incr(&y);
}
```

Rust:

```
fn main() {
    let mut x, y = 0;
    incr(&mut x);
    incr(&mut y);
}
```



Verification:

Do x and y alias?
If yes, x = 1, y = 1, else x = 1, y = 0

Do x and y alias?
If yes, x = 2, y = 2, else x = 1, y = 1

```
let main () =
    let x, y = 0 in
    let x = incr x in
    let y = incr y
```

x and y are **different variables**, no memory reasoning required

Translating safe Rust to Pure Code

Rust:

```
fn choose<'a>(
  b : bool, x : &'a mut i32, y : &'a mut i32)
  -> &'a mut i32
{
    if b { return x; }
    else { return y; }
}
```

Types derived from Rust signature *only*

Translation:

```
let choose_fwd (b : bool) (x : i32) (y : i32) : i32 =
  if b then x else y

let choose_back (b : bool) (x : i32) (y : i32) (z : i32) :
  i32 * i32 =
  if b then (z, y) else (x, z)
```

```
let mut x = 0;
let mut y = 1;
let z = choose(true, &mut x, &mut y);

*z = 2; // Update x

// Observe the changes
assert!(x == 2);
assert!(y == 1);
...
```

'a ends here

```
let x = 0 in
let y = 1 in
let z = choose_fwd true x y in

let z = 2 in

let (x, y) = choose_back true x y z in
...
```

Modular translation with *forward* and *backward* functions

Translating Recursion

Rust:

```
pub enum List<T> {
  Cons(T, Box<List<T>>),
  Nil,
}

fn nth<'a, T>(l: &'a mut List<T>, i: u32)
  -> &'a mut T {
  match l {
    List::Cons(x, tl) => {
      if i == 0 {
        return x;
      }
      else {
        return nth(tl, i - 1);
      }
    }
    List::Nil => { panic!() }
  }
}
```

Translation:

```
let rec nth_fwd (t : Type) (l : list_t t) (i : u32) : result t =
  begin match l with
  | ListCons x tl ->
    if i = 0
    then Return x
    else begin i0 <-- u32_sub i 1; nth_fwd t tl i0 end
  | ListNil -> Fail Failure
  end
```

```
let rec nth_back (t : Type) (l : list_t t) (i : u32) (ret : t) :
  result (list_t t) =
  begin match l with
  | ListCons x tl ->
    if i = 0
    then Return (ListCons ret tl)
    else begin
      i0 <-- u32_sub i 1;
      tl0 <-- nth_back t tl i0 ret;
      Return (ListCons x tl0) end
  | ListNil -> Fail Failure
  end
```

Forward and backward functions behave like **lenses**

Translating Loops

Rust:

```
pub enum List<T> {
  Cons(T, Box<List<T>>),
  Nil,
}

pub fn nth<T>(mut ls: &mut List<T>, mut i: u32)
  -> &mut T {
  loop {
    match ls {
      List::Cons(x, tl) => {
        if i == 0 { return x; }
        else {
          ls = tl;
          i -= 1;
          continue;
        }
      }
      List::Nil => { panic!() }
    }
  }
}
```

Translated functions are **similar to the recursive case**

Translation:

```
let rec nth_loop_fwd
  (t : Type) (ls : list_t t) (i : u32) : result t =
  begin match ls with
  | ListCons x tl ->
    if i = 0 then Return x
    else begin i0 <-- u32_sub i 1; nth_loop_fwd t tl i0 end
  | ListNil -> Fail Failure
  end

let nth_fwd t ls i = nth_loop_fwd t ls i
```

```
let rec nth_loop_back
  (t : Type) (ls : list_t t) (i : u32) (ret : t) :
  result (list_t t) =
  begin match ls with
  | ListCons x tl ->
    if i = 0 then Return (ListCons ret tl)
    else begin
      i0 <-- u32_sub i 1;
      tl0 <-- nth_loop_back t tl i0 ret;
      Return (ListCons x tl0) end
  | ListNil -> Fail Failure
  end

let nth_back t ls i ret = nth_loop_back t ls i ret
```

Translation: Key Ingredients

- Translation based on a **symbolic execution** computing the borrow graph
  - Reimplements a borrow checker for Rust
  - Formalized, captures the essence of borrow checking
  - Allows to formally study extensions of the Rust type system
  - Mechanization in Rocq ongoing

Applying Aeneas to ML-KEM

Why deploy post-quantum cryptography **today**?
Harvest now, decrypt later attacks: messages sent today are vulnerable to quantum attackers tomorrow

FIPS 203

Federal Information Processing Standards Publication

Module-Lattice-Based Key-Encapsulation Mechanism Standard

Category: Computer Security **Subcategory: Cryptography**

ML-KEM (post-quantum key encapsulation mechanism): first widely deployed PQ crypto algorithm

ML-KEM (i): RFC vs Code

NIST RFC:

Algorithm 9 $\mathrm{NTT}(f)$ (19 algorithms in total)

Computes the NTT representation $\hat{f}$ of the given polynomial $f \in R_q$.

Input: array $f \in \mathbb{Z}_q^{256}$.

Output: array $\hat{f} \in \mathbb{Z}_q^{256}$.

$\hat{f} \leftarrow f$
$i \leftarrow 1$
for ($len \leftarrow 128$; $len \geq 2$; $len \leftarrow len/2$)
  for ($start \leftarrow 0$; $start < 256$; $start \leftarrow start + 2 \cdot len$)
    $zeta \leftarrow \zeta^{\mathrm{BitRev}_7(i)} \bmod q$
    $i \leftarrow i + 1$
    for ($j \leftarrow start$; $j < start + len$; j++)
      $t \leftarrow zeta \cdot \hat{f}[j + len]$
      $\hat{f}[j + len] \leftarrow \hat{f}[j] - t$
      $\hat{f}[j] \leftarrow \hat{f}[j] + t$
    end for
  end for
end for
return $\hat{f}$

Multiplication modulo 3329

Rust (constant-time, optimized implementation):

```
const ZETA_BIT_REV_TIMES_R: [u16; 128] = [
  2285, 2571, 2970, 1812, 1493, 1422,  287,  202,
  3158,  622, 1577,  182,  962, 2127, 1855, 1468,
  ...
];

const ZETA_BIT_REV_TIMES_R_TIMES_NEG_Q_INV_MOD_R: [u16; 128] = [
    19, 34037, 50790, 64748, 52011, 12402, 37345, 16694,
  20906, 37778,  3799, 15690, 54846, 64177, 11201, 34372,
  ...
];

fn mod_sub(a: u32, b: u32) -> u32 {
  debug_assert!( a < 2*Q );
  debug_assert!( b <= Q );

  let res = a.wrapping_sub(b);
  debug_assert!( ((res >> 16) == 0) || ((res >> 16) == 0xffff) );
  let res = res.wrapping_add(Q & (res >> 16));
  debug_assert!( res < Q );

  res
}

fn mont_mul(a: u32, b: u32, b_mont: u32) -> u32 {
  debug_assert!( a < Q );
  debug_assert!( b < Q );
  debug_assert!( b_mont <= RMASK );
  debug_assert!( b_mont == ((b * NEG_Q_INV_MOD_R) & RMASK) );

  let mut res = a * b;
  let inv = (a * b_mont) & RMASK;
  res += inv * Q;
  res >>= RLOG2;

  mod_sub( res, Q )
}

...
```

ML-KEM (ii): Formal Spec

NIST RFC:

Algorithm 9 NTT(f)

Computes the NTT representation $\hat{f}$ of the given polynomial $f \in R_q$.

Input: array $f \in \mathbb{Z}_q^{256}$.

Output: array $\hat{f} \in \mathbb{Z}_q^{256}$.

1: $\hat{f} \leftarrow f$
2: $i \leftarrow 1$
3: **for** ($len \leftarrow 128$; $len \geq 2$; $len \leftarrow len/2$)
4: **for** ($start \leftarrow 0$; $start < 256$; $start \leftarrow start + 2 \cdot len$)
5: $zeta \leftarrow \zeta^{\text{BitRev}_7(i)} \bmod q$
6: $i \leftarrow i + 1$
7: **for** ($j \leftarrow start$; $j < start + len$; j++)
8: $t \leftarrow zeta \cdot \hat{f}[j + len]$
9: $\hat{f}[j + len] \leftarrow \hat{f}[j] - t$
10: $\hat{f}[j] \leftarrow \hat{f}[j] + t$
11: **end for**
12: **end for**
13: **end for**
14: **return** $\hat{f}$

toPoly (Code.ntt f) ≃ Spec.ntt (toPoly f)

RFC ported to Lean:

Spec is executable and testable

```
def Spec.ntt (f : Polynomial) : Polynomial := Id.run do
 let mut f := f
 let mut i := 1
 for h0: len in [128 : >1 : /= 2] do
  for h1: start in [0:256:2*len] do
   let zeta := ζ ^ (bitRev 7 i)
   i := i + 1
   for j in [start:start+len] do
    let t := zeta * f[j + len]
    f := f.set (j + len) (f[j] - t)
    f := f.set j (f[j] + t)
 pure f
```

Bounds are statically checked

⇒ Trustworthy specification

Lean Theorem:

```
def inBounds (a : Std.Array U16 256#usize) :=
 ∀ i < 256, a[i].val < 3329

def toPoly (a : Std.Array U16 256#usize) : Polynomial := ...

theorem Code.ntt.spec
 (f : Std.Array U16 256#usize) (h : inBounds f) :
 Code.ntt f ⦃ f1 =>
  toPoly f1 = Spec.ntt (toPoly f) ∧ inBounds f1 ⦄
```

Current Status

- SymCrypt Windows library is adopting Rust code verified with Aeneas
- First completed target: optimized parts of ML-KEM post-quantum key-exchange primitive
- Ongoing work: FrodoKEM, SHA3, AES-GCM

Interoperating with Legacy Systems

- More and more projects move towards Rust, but many existing projects still rely on C

- Eurydice can translate a subset of safe Rust to C code
  - Type monomorphization
  - Trait elimination
  - Translation of high-level iterators to for/while loops

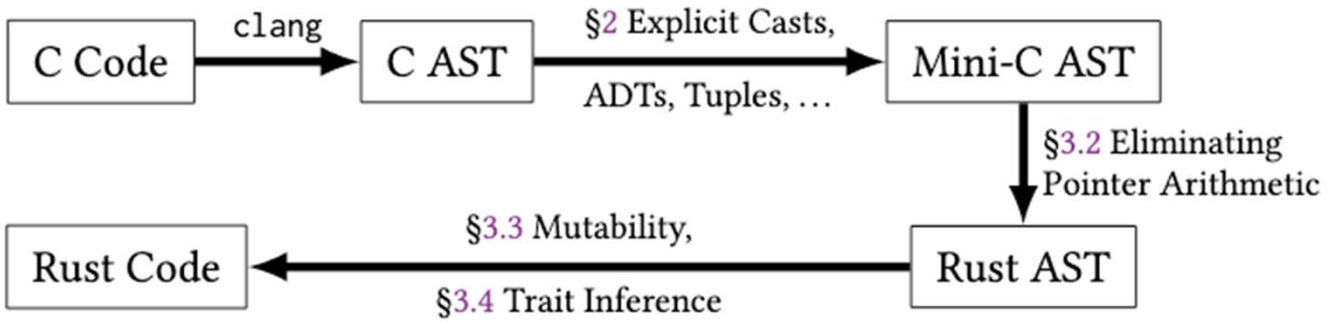
- Allows to develop (and verify) code in Rust, and deploy C code when needed

https://github.com/AeneasVerif/eurydice

Porting Legacy Code to Rust

- Several projects (TRACTOR, C2Rust) aim to automatically translate existing C/C++ code to Rust
- To support all of C, they mostly target *unsafe* Rust, losing safety guarantees
- Translations do not necessarily preserve semantics
- **Our approach:**
  - Target a **small subset of C**, but translate it to **safe Rust**
  - Perform **targeted rewritings** in existing C code to match our supported subset

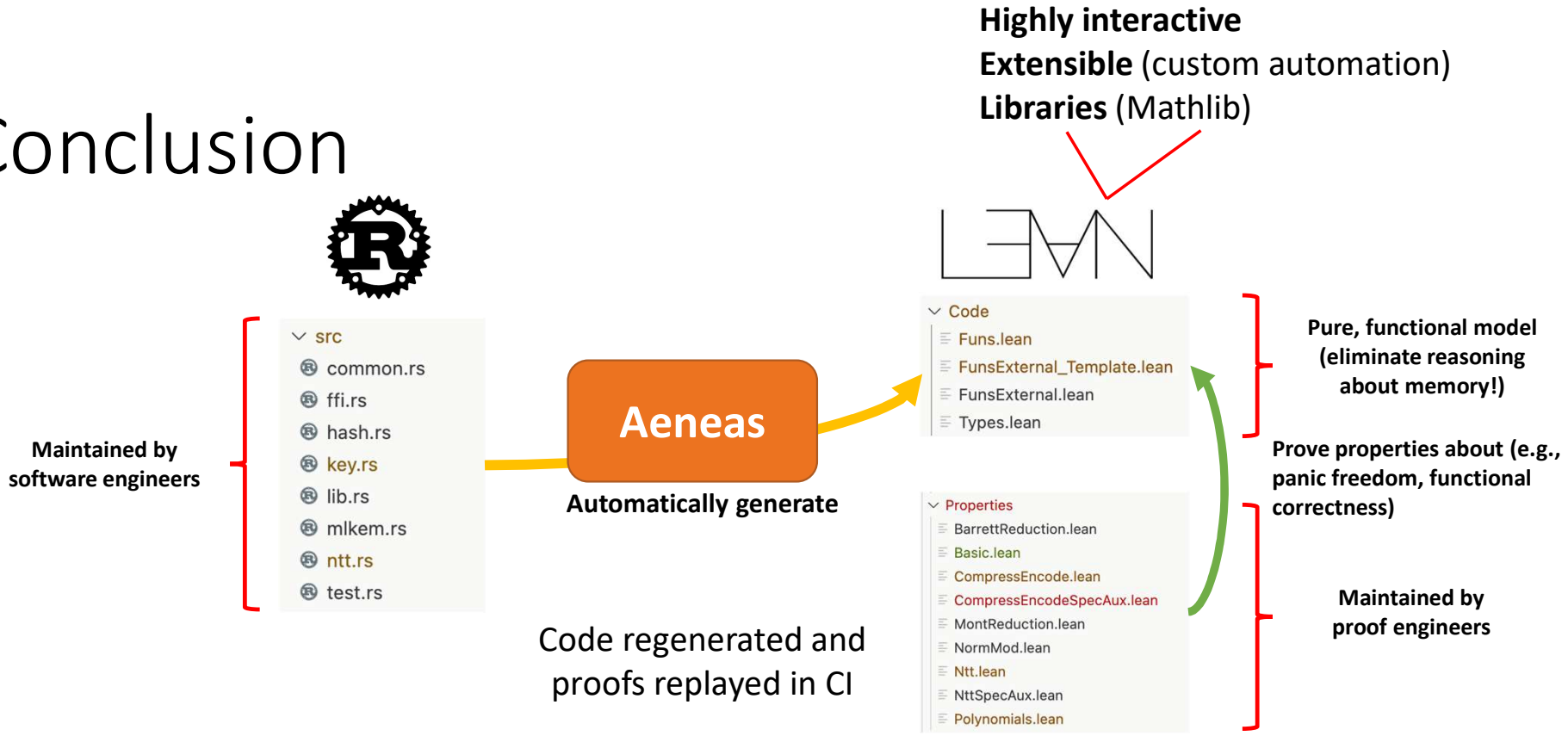
Scylla: Translating a Subset of C to safe Rust



- Case studies: cryptographic libraries, binary parsers, compression algorithms
- Performance of the translated code on par with the C sources

https://github.com/AeneasVerif/scylla

Conclusion



- Applied to modern post-quantum cryptographic implementations
- Supported by an ecosystem of tools to simplify the transition to Rust

https://github.com/AeneasVerif/aeneas

aymeric.fromherz@inria.fr