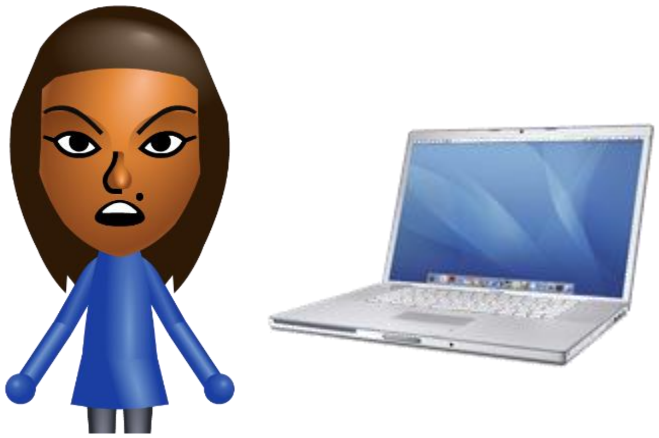
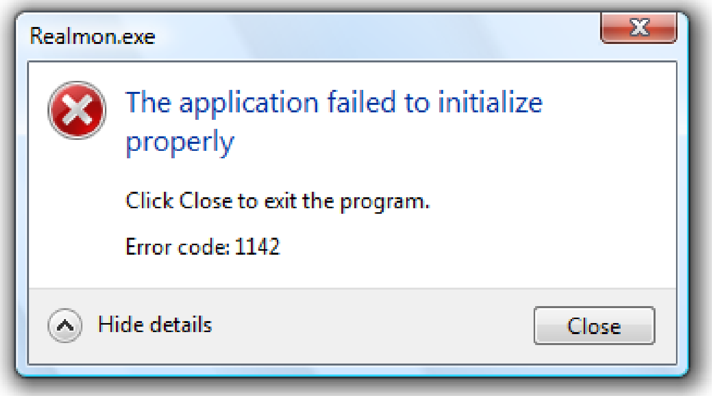
The Aeneas Ecosystem: Formal Verification of Rust Programs

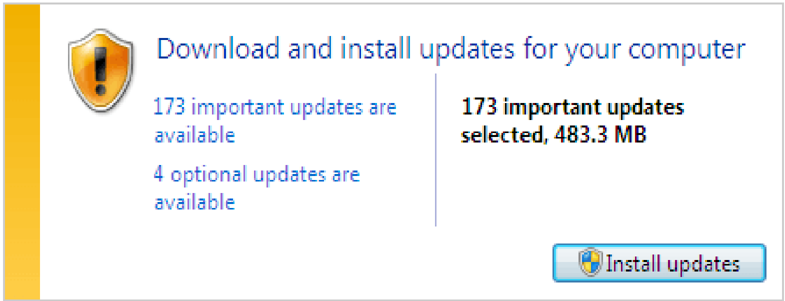
Aymeric Fromherz (Inria Paris),

With *Guillaume Boisseau, Son Ho, Yoann Prak, Jonathan Protzenko*

An Iterative Development Process







- Solves the immediate problem
- But only addresses already identified bugs

What About Critical Software?

Report: Software Design Errors Caused Ariane 5 Explosion

July 23, 1996

What About Critical Software?

This ‘Magical Bug’ Exposed Any iPhone in a Hacker's Wi-Fi Range

A Google researcher found flaws in Apple's AWDL protocol that would have allowed for a complete device takeover.

EDITORS' PICK | 22,046 views | Feb 19, 2020, 04:37am EST

Hackers Made Tesla Cars Autonomously Accelerate Up To

Ransomware Attacks Grow, Crippling Cities and Businesses

Hackers are locking people out of their networks and demanding big payments to get back in. New data shows just how common and damaging the attacks have become.

What Can We Do Better?

- More testing/auditing?
    - TweetNaCl Vulnerability

> This bug is triggered when the last limb `n[15]` of the input argument `n` of this function is greater or equal than `0xffff`. In these cases the result of the scalar multiplication is not reduced as expected resulting in a wrong packed value. This code can be fixed simply by replacing `m[15]&=0xffff;` by `m[14]&=0xffff;`.

seb.dbzteam.org

What Can We Do Better?

- More testing/auditing?
  - TweetNaCl Vulnerability
  - 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

What Can We Do Better?

- More testing/auditing?
  - TweetNaCl Vulnerability
  - Bug in amd-64-64-24k Curve25519
- How to understand threats considered, and react to emerging threats?
- Critical software needs **strong, formal guarantees** of its **correctness** and **security**

Formal Verification to the Rescue

- Formal verification can provide **mathematical guarantees** about program behaviours
- Considers all execution paths

- Several successes over the past decade
  - CompCert C compiler
  - seL4 micro-kernel
  - Astrée static analyzer
  - Many efforts for cryptographic implementations

HACL*: A Verified Cryptographic Library

- Joint development between Inria and Microsoft Research
- Provides guarantees about memory safety, functional correctness, resistance against side-channels
- Developed using the F* proof assistant
- ~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, Tezos, Python, and many more

The F* Ecosystem

Proof Frameworks

- **Vale**: Verification of assembly code
- **Low***: Verification of C code
- **Steel**: Verification of concurrent, low-level systems

Secure Applications

- **HACL***: Verified Crypto library
  - Integrated in Mozilla, Python, Linux, …
- **EverParse**: Verified Binary Parsers
  - Integrated in Microsoft’s Hyper-V
- **StarMalloc**: Verified, Concurrent, Security-Oriented Memory Allocator
  - Usable as drop-in replacement for Firefox, redis, …

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

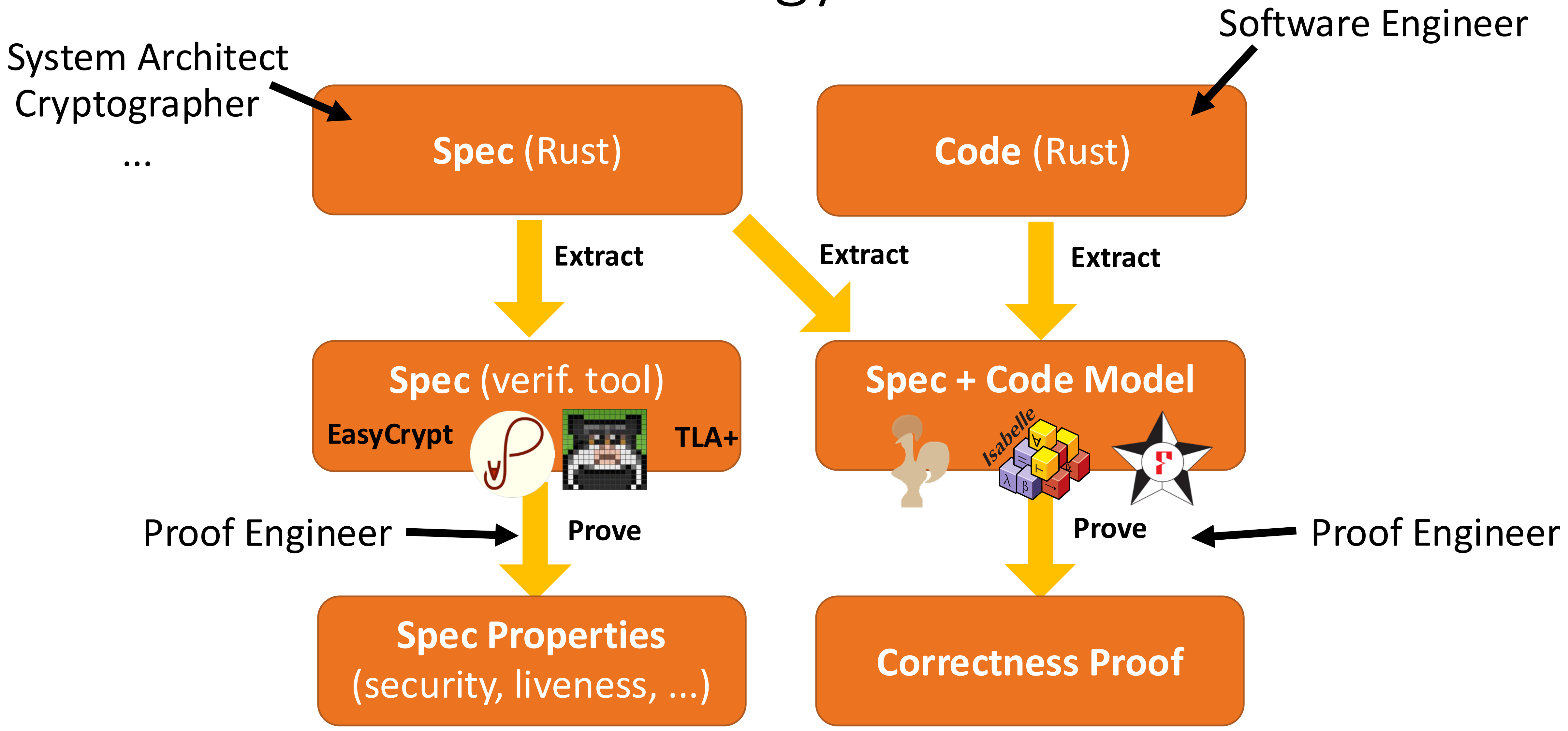
- How to support a multitude of tools for diverse usecases?
- How to empower teams with different proof expertises?

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?

The Aeneas 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

Rust Issues

- Aborted executions at run-time (panic)
  - E.g., out-of-bounds memory accesses. Risks of DoS
- Memory vulnerabilities in *unsafe* code
  - Similar to C/C++
  - Unsafe code needs to interoperate with *safe* code
- Design flaws, incorrect implementations, …

Still need formal verification!

Translating Rust to Pure Code

- **Core idea:** Leverage invariants provided by the Rust type system to simplify verification

- We translate Rust programs to semantically equivalent functional models

- We can reason about these models in different proof assistants

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:

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

Rust:

```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; }
}
```

```
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

Translation:

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

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

let z = 2 in

...
```

?

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; }
}
```

```
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);
...
```

Translation:

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

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

let z = 2 in

let (x, y) = if true then (z, y) else (x, z) in
...
```

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);
...
```

```
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 Coq ongoing

Reusing Proof Assistant Ecosystems

- Functional models can be extracted to many proof assistants
- Allows reusing existing libraries and verified developments
  - Mathlib in Lean
  - Verified compilation and semantics in Coq
  - SMT automation and side-channel reasoning in F*

- Possible to prove different properties on the same program in different tools

Rust Static Analysis

- Reasoning in proof assistants is time-consuming
- Some properties do not require a high expressiveness of the tools
- Can be checked through static analyses

- Interacting with the Rust compiler is tricky
  - Compilation-oriented representations
  - Internal details and specific APIs
  - Some information needs to be reconstructed

Charon: a Rust Analysis Framework

- Charon offers analysis-oriented representations for Rust programs
- Integrates with the Cargo build system, abstracts compiler internals
- Reconstructs analysis-relevant information (trait instantiations, …)
- Simplifies representations (constants, pattern-matching, …)
- Reconstructs control-flow

https://github.com/AeneasVerif/charon

Charon Applications

- Common interface for the Rust compiler, entrypoint of most of our tools

- Prototype of side-channel analysis on top of Charon
  - Can detect timing and cache-based side channels in cryptographic code
  - Rediscovers the KyberSlash vulnerability in PQC implementations

- Other uses in the broader Rust verification community

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

Handling Pointer Arithmetic

- In applicative C code, pointer arithmetic is mostly used for array manipulations

```
let ab = abcd + 0;     // Slice of two elements of abcd
let dc = abcd + 2;     // Slice of two elements of abcd
```

```
let ab = (&mut tmp1)[0..];
let dc = (&mut tmp1)[2..];
```

```
let sp = tmp1.split_at_mut(2usize);
let ab = sp.0;
let dc = sp.1;
```

Data Mutability

- By default, all data is implicitly mutable in C

```
void add(uint8* out, uint8* x, uint8* y) { out[0] = x[0] + y[0] }
add(out, x, x);
```

- Mutable data cannot alias in Rust

```
fn add(out: &mut u8, x:&mut u8, y: &mut u8) { ... }
add(out, x, x);
```

```
fn add(out: &u8, x:&u8, y: &u8) { out[0] = x[0] + y[0] }
```

Inferring Mutability

- By default, we translate all pointers to immutable borrows
- We perform a backward analysis to precisely infer mutability

```
fn add(out: &u8, x:&u8, y: &u8) { out[0] = x[0] + y[0] }
```

⬇

```
fn add(out: &mut u8, x:&u8, y: &u8) { out[0] = x[0] + y[0] }
```

- Small changes and insertion of copies sometimes needed in source code to match Rust semantics

Scylla: Preliminary Results

- Very experimental. Currently implemented as extractor for highly structured, verified F* code
- WIP: libclang frontend to parse C files directly
- C subset considered is sufficient for verified cryptography
- We can translate HACL* to 80,000 lines of safe Rust
- Can also translate parts of SymCrypt, bzip2 directly from C
- **Not applicable to generic C code**, but can help with legacy applicative C code

https://github.com/AeneasVerif/scylla

Conclusion

- While being safer than C/C++, Rust opens new challenges and avenues for formal verification
- Aeneas proposes to verify safe Rust programs through a translation to a functional model:
  - Eliminates memory reasoning
  - Allows the use of different proof assistants (and leverage different expertises)
- An ecosystem of tools around Aeneas helps handling legacy systems, and developing new analyses

https://aeneas-verif.zulipchat.com/ https://github.com/AeneasVerif

aymeric.fromherz@inria.fr