Project Everest: Building a Provably Secure HTTPS Ecosystem with F*

An Introduction to Formal Methods in Cryptography

Daniel Dia
The Key Exchange (CMPS297AD/CMPS396AI)
American University of Beirut

October 20, 2025

Overview

1. A New Way of Thinking About Types
2. Motivation: Why Crypto Needs Formal Proof
3. Core Concepts: Dependent Types & F*
4. Case Study: Project Everest
5. Conclusion

A Question to Start

Has any one of you written code in a "bare metal" programming language?

(C, C++, Rust, Asm, whatever...)

Data, Behavior, and Types

Classical OOP (C++, Java, etc.)

Data and behavior are tightly coupled within objects.

- Relies on Inheritance.
- Leads to rigid hierarchies.

Data-Oriented Design (Rust, Haskell, etc.)

Data and behavior are separate.

- Favors Composition.
- Build behavior onto simple data.

What is a "Type," Really?

The Systems View (e.g. C++)

A type is a description of data in memory.

- 'int' is 4 bytes.
- 'MyObject' is a memory layout.
- The focus is on the hardware.

The Formal Methods View (F*, Lean4)

A type classifies values and expresses constraints.

- A tool for reasoning about correctness before the program runs.
- The focus is on provable properties.

Languages like F* and Lean4 apply this powerful view of types to build provably correct software.

The Gorilla and the Banana (OOP)

Joe Armstrong, creator of Erlang, on OOP:

"You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."

- Inheritance forces you to accept the entire complex hierarchy ('JungleAnimal' → 'Mammal'...) just to get one piece of behavior.
- This leads to the Brittle Base Class Problem.
  → change in a parent class can unexpectedly break child classes in subtle ways.

Behavior as Interfaces (Traits)

A Compositional Approach

1. Model data with simple ‘structs‘ (like in C). They hold data, nothing else.
2. Define behaviors as interfaces (called ‘traits‘ or ‘typeclasses‘).
3. Compose behaviors onto your data structures by implementing these interfaces for them.

Flexibility

This avoids rigid hierarchies. You don’t inherit the ”jungle”; you just implement the ‘EatsBananas‘ interface for your ‘Gorilla‘ data structure.

Making Impossible States Impossible with ADTs

Modern type systems use **Algebraic Data Types** (ADTs), where a type can be one of several variants, each holding different data.

The Compiler's Guarantee: Exhaustiveness

Control flow constructs like 'match' (in F*, Rust) or 'induction' (in Lean) are exhaustive. The compiler will produce an error if you forget to handle a case. This prevents entire classes of bugs.

Example: A Rust Enum with Varied Data

```
enum WebEvent {
    PageLoad,                   // Variant with no data
    KeyPress(char),             // Variant with a tuple
    Click { x: i64, y: i64 },   // Variant with a struct
}
```

C++: ‘std::variant‘ and ‘std::visit‘

```cpp
#include <variant>
#include <string>

struct GetRequest { std::string path; };
struct PostRequest { std::string path, body; };
struct DeleteRequest { std::string path; };

using Request = std::variant<GetRequest, PostRequest, DeleteRequest>;

void handle_request(const Request& req) {
    std::visit([](auto&& arg) {
        using T = std::decay_t<decltype(arg)>;
        if constexpr (std::is_same_v<T, GetRequest>) {
            // handle Get
        } else if constexpr (std::is_same_v<T, PostRequest>) {
            // handle Post
        }
        // No compile error for forgetting DeleteRequest.
    }, req);
}
```

The Danger

This code compiles without complaint, silently ignoring the unhandled ‘DeleteRequest‘. This is a bug that exhaustive pattern matching would catch.

From Enforcing Rules to Proving Theorems

The Key Insight

A strong type system doesn't just describe data; it enforces rules about how that data can be used.

- RAII (and 'unique_ptr' in C++) enforce a form of single ownership. Note: While you **can** have this in C++, it sucks, Rust does it better.
- Exhaustive pattern matching enforces that all program states are handled.

The Next Step: F* and Lean4

What if we could make the rules even more powerful? This is where formal methods begin. The powerful type systems in languages like F* and Lean4 allow them to function as proof assistants.

The Challenge of Secure Cryptography

The Stakes are Higher

In standard software, we rely on testing and code reviews. For cryptography, bugs are more subtle and the consequences are far more severe.

Catastrophic Consequences

A single, tiny bug in a cryptographic implementation can compromise an entire system's security (e.g., Heartbleed).

Why Testing Is Not Enough

"In general, I'd say that anybody who designs his kernel modules for C++ is either
(a) looking for problems
(b) a C++ bigot that can't see what he is writing is really just C anyway
(c) was given an assignment in CS class to do so.
Feel free to make up (d)."
— Linus Torvalds (2007-09-06)

Testing is Insufficient

Cryptographic algorithms operate on an enormous space of keys, nonces, and inputs. It is impossible for testing to cover every case.

- You can test that encryption works for *some* inputs...
- ...but you can't prove it's secure for *all* of them.

Real-World Example: The Heartbleed Bug (2014)

What Was It?

A critical bug in the popular OpenSSL library. It was **not** a flaw in the cryptography itself, but a simple programming error in the implementation of the TLS "Heartbeat" extension.

- A client could send a "heartbeat" message to a server to keep a connection alive.
- The message included a payload and its length.
- The client could lie, claiming the payload was much larger than it actually was.

The Vulnerability: A Missing Bounds Check

OpenSSL would allocate a buffer for the response based on the *claimed* length, copy the (small) payload into it, and then copy and return whatever happened to be next in memory to fill the rest of the claimed length. This leaked private keys, passwords, and other sensitive data.

How Can We Prove Away Timing Attacks?

The Problem: Leaky Code

If the execution path depends on a secret, it leaks information.

```
// Vulnerable: timing depends on secret
int isEqual(byte secret[], byte input[]) {
    for (int i=0; i<16; i++) {
        if (secret[i] != input[i]) {
            return 0; // Exits early!
        }
    }
    return 1;
}
```

By timing how long this takes to return, an attacker can guess the secret byte by byte.

The Solution: Constant-Time Proofs

We write code where the sequence of instructions is the same regardless of secret values. Then we **prove** this property using the type system.

```
// Constant-time: always takes same time
int isEqual_const(byte secret[], byte input[]) {
    int result = 0;
    for (int i=0; i<16; i++) {
        result |= secret[i] ^ input[i];
    }
    return result == 0;
}
```

A language like F* can formally prove that a function like `isEqual_const` has no branches that depend on secret data.

Dependent Types: Giving Types More Power

A **dependent type** is a type that depends not just on other types, but on values.

The Core Idea

This allows you to encode program logic and invariants directly into the type system, moving runtime checks into compile-time guarantees.

A Simple Analogy

In Rust, `Vec<i32>` is a vector of integers, but `Vec<42>` makes no sense because types can't depend on values. With dependent types, a type like 'Vector Int 42' (a vector of 42 integers) is possible and powerful.

From C++ Templates to Dependent Types

In C++, `std::array<int, 10>` is a type that depends on the *value* 10. Dependent types take this much further.

Example

```
// In C, you pass a pointer and length, and check at runtime.
void process_array(int* data, size_t len) {
  if (len != 3) { /* handle error */ }
}

// With dependent types, the check moves to compile time.
// (Hypothetical C++-like syntax)
template<typename T, size_t N> struct SizedBuffer { T* data; };

void process_3_elements(SizedBuffer<int, 3> buffer);

SizedBuffer<int, 3> buf1;
SizedBuffer<int, 4> buf2;

process_3_elements(buf1); // OK!
// process_3_elements(buf2); // COMPILE ERROR!
```

F*: A Language for Provably Correct Code

F* is a functional programming language with dependent types that also acts as a proof assistant.

Refined Types in F*

You can create types with built-in propositions. For example, a natural number (`nat`) is an integer that is proven to be non-negative.

```
// A type for natural numbers (non-negative integers)
type nat = x:int{x >= 0}
```

The compiler will reject any code that could possibly violate this property.

Remember: A **type** is a way for us to classify values and express **constraints** on them, and a **type system** allows us to REASON about them at compile time.

Types as Theorems: The Curry-Howard Correspondence

The Big Idea: Propositions are Types

In this paradigm, a logical proposition (a statement that can be true or false) is represented as a type. A program that has that type is a **proof** of the proposition.

Example: A Theorem in a Type

This function signature is not just a contract; it is a *provable theorem*.

```
(* For any type 'a', and any natural numbers 'n' and 'm',
   the function 'append' takes a vector of 'a's of length 'n'
   and a vector of 'a's of length 'm', and is proven to return
   a new vector of 'a's of length 'n + m'. *)
val append : #a:Type -> n:nat -> m:nat -> vec a n -> vec a m -> vec a (n + m)
```

How is this proven?

When the logic gets complex, the developer writes a proof script using **tactics** to guide the compiler's reasoning (this will have to do with how "**strong**" the type system is...).

Project Everest: A Verified HTTPS Ecosystem

The Goal

To build a fully verified, high-performance HTTPS software stack from the ground up, replacing libraries like OpenSSL with components that are provably correct and secure.

A Major Collaborative Effort (feat. Dr. Nadim Kobeissi™)

Project Everest is a joint effort between several major research institutions:

- Microsoft Research
- INRIA (French National Institute for Research in Digital Science and Technology)
- Carnegie Mellon University
- University of Edinburgh, and others.

Why This Matters for C/C++ Developers

Everest produces verified C code that can be used directly in any systems project.

The Everest Stack: HACL* (1)

A Verified Cryptographic Library

HACL* (High-Assurance Cryptographic Library) is a library of modern cryptographic primitives written and verified in F*.

- **Primitives:** ChaCha20, Poly1305, Curve25519, AES-GCM, SHA-2, SHA-3, etc.
- **Guarantees:** Each is formally proven to be:
  - Memory Safe: No buffer overflows.
  - Functionally Correct: Matches its specification (e.g., RFC).
  - Side-Channel Resistant: Constant-time execution.

Figure: C++ programmers when people ask them how secure their code is

The Everest Stack: KreMLin (2)

A Verified Compiler for High-Assurance C

KreMLin is a compiler that translates a subset of F* (called Low*) into readable, efficient, and dependency-free C code.

F* (Low*) Source Code

```
// Verified F* implementation
let add_mod_32 (a:uint32) (b:uint32)
  : uint32 =
  (a + b) % (pow2 32)
```

Generated C Code

```
// Efficient, portable C
uint32_t add_mod_32(uint32_t a, uint32_t b)
{
  return a + b;
}
```

The generated C code inherits all the safety guarantees proven in F*!

The Everest Stack: MiTLS (3)

A Verified TLS Implementation

MiTLS is a verified reference implementation of the TLS 1.2 and 1.3 protocols, also written in F*.

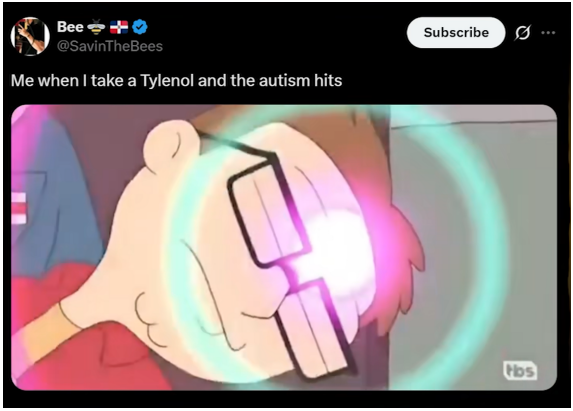
Beyond Primitives: Verifying the State Machine

Most TLS vulnerabilities are not in the crypto algorithms themselves, but in the complex logic of the protocol's state machine. MiTLS proves that the implementation of this state machine is free of logical flaws that could lead to security compromises.

The Everest Stack: How It Works

The Workflow from Proof to Production

1. **Write in F***: Implement an algorithm with a precise dependent type.
2. **Prove in F***: Use tactics to write a formal proof that the implementation meets its specification.
3. **Compile to C**: The proven F* code is fed to the KreMLin compiler.
4. **Ship Verified C Code**: KreMLin produces standard `.c` and `.h` files.



A Schematic of the Stack

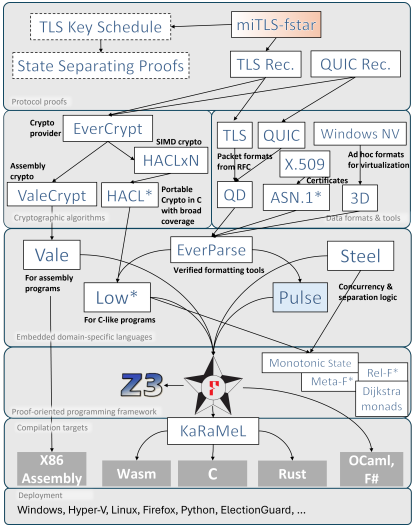


Figure: The verified components of Project Everest, from F* source code to compiled C.

Example: Proving Memory Safety in F*

F* Code for a Length-Indexed Byte Array

```
// F* type for a byte array with a specific length 'len'
type bytes (len:nat) = b:array u8{Array.length b = len}
```

A Verified Function Signature

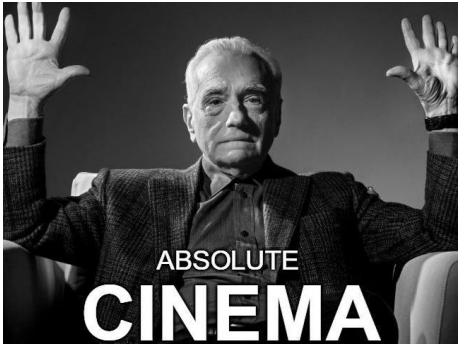
```
// This type signature *guarantees* no out-of-bounds access.
val aead_encrypt :
    key:bytes 32 ->         // 32-byte key
    iv:bytes 12 ->          // 12-byte IV
    plaintext:bytes len ->   // plaintext of length 'len'
    ciphertext:lbuffer len -> // output buffer of same length
    unit
```

Real-World Impact

Who Uses This Verified Code?

The C code generated by Project Everest is used in major, real-world projects, including:

- **Mozilla Firefox** → NSS, the cryptography library behind Firefox
- **The Linux Kernel** → Crypto subsystem, e.g. implementations for WireGuard VPN
- **The Windows Kernel** → 'bcrypt.sys', the core kernel cryptography module
- **Tezos Blockchain** → core cryptographic primitives for the consensus protocol
- **WireGuard VPN** → primary multiplatform cryptographic backend



Summary for a Systems Programmer

Dependent Types: What They Are

Think of them as C++ templates and `static_assert` on steroids. They let you bake value-dependent invariants (like buffer sizes and ranges) directly into the type itself.

Example: An F* Refined Type for a Buffer

```
// A type for a byte array that is PROVEN to have exactly 32 bytes.
type key_buffer = b:array u8{Array.length b = 32}

// A function using this type can't be called with a wrong-sized buffer.
// This prevents an entire class of errors at compile time.
val use_key : key_buffer -> unit
```

Summary for a Systems Programmer

Tactic-Based Proving: What It Is

An interactive conversation with the compiler to prove that your code satisfies its complex type. You provide high-level commands (**tactics**) to guide the proof.

Example: A Lean4 Tactic Proof with Explanation

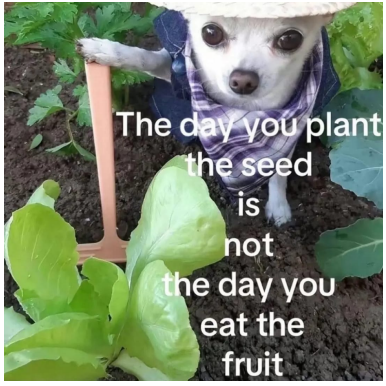
```
theorem and_comm (p q : Prop)
    : p AND q → q AND p := by  -- The goal is to prove q AND p
  intro h           -- Assume we have a proof `h` of `p AND q`
  apply And.intro   -- To prove `q AND p`, we must prove `q` & `p` separately
  · exact h.right   -- Goal 1: Prove `q`. We get it from the RHS of `h`.
  · exact h.left    -- Goal 2: Prove `p`. We get it from the LHS of `h`.
```

Summary for a Systems Programmer

F* and Project Everest → Ultimate Payoff

It starts with powerful type systems that see **types as provable theorems** (Dependent Types). When automation isn't enough, we guide the compiler with **proof scripts** (Tactics).

Such projects, use languages like **F*** to produce C code that is not just tested, but **formally proven** to be memory-safe, functionally correct, and secure against entire classes of side-channel attacks. This brings an unprecedented level of assurance to the software we rely on.



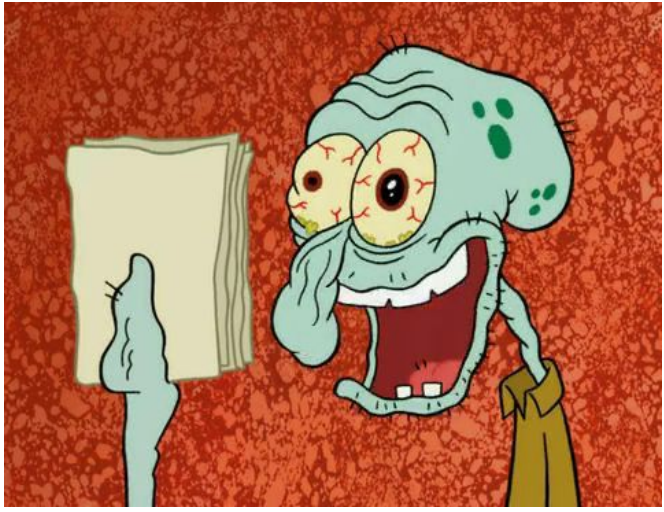
Further Readings & Cool Stuff

Further Readings & Cool Stuff (Serious)

- **Natural Number Game** (The classical introduction game for Lean): https://adam.math.hhu.de/
- **Project Everest: Provably Secure Communication Software** (website): https://project-everest.github.io/
- **Project Everest: Perspectives from Developing Industrial-grade High-Assurance Software** (Project Everest Team) May, 2025. https://project-everest.github.io/assets/everest-perspectives-2025.pdf
- **HACL*: A Verified Modern Cryptographic Library** (Zinzindohoue et al., Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security): https://dl.acm.org/doi/10.1145/3133956.3134043

The End

Questions?



Project Everest: Building a Provably Secure HTTPS Ecosystem with F*

An Introduction to Formal Methods in Cryptography

Daniel Dia
The Key Exchange (CMPS297AD/CMPS396AI)
American University of Beirut

October 20, 2025