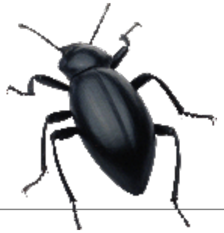
# Programming with Proofs for High-assurance Software

NIKHIL SWAMY

Senior Principal Researcher
Microsoft Research, Redmond

Computational Cybersecurity in Compromised Environments (C3E)
September 11, 2020

# High assurance software?




Microsoft Hyper-v

Microsoft SQL Azure

CloudBleed

...frastructure

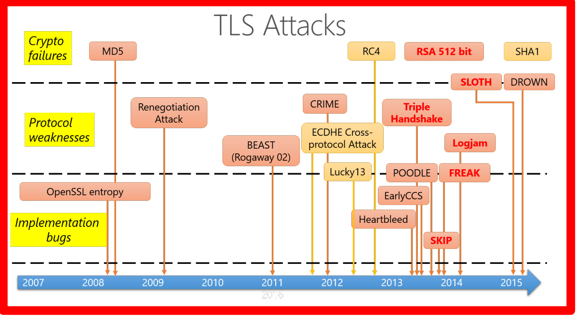
TLS 1.3

QUIC

WIREG...

FAST, MODERN, SE...

Secu...




Financial technology

Verifiable multi-party computations

Veritas

Wysteria

E-voting

ElectionGuard

**Formal proofs to the rescue**

Project Everest

https://project-everest.github.io

Project Everest

Papers People In the News Related Projects

Microsoft Inria Carnegie Mellon University Microsoft Research - Inria JOINT CENTRE

> `Project Everest aims to build and deploy a verified HTTPS stack`

We are a team of researchers and engineers from several organizations, including Microsoft Research, Carnegie Mellon University, INRIA, and the MSR-INRIA joint center.

Everest is a recursive acronym: It stands for the “Everest VERified End-to-end Secure Transport”.

# Project Everest

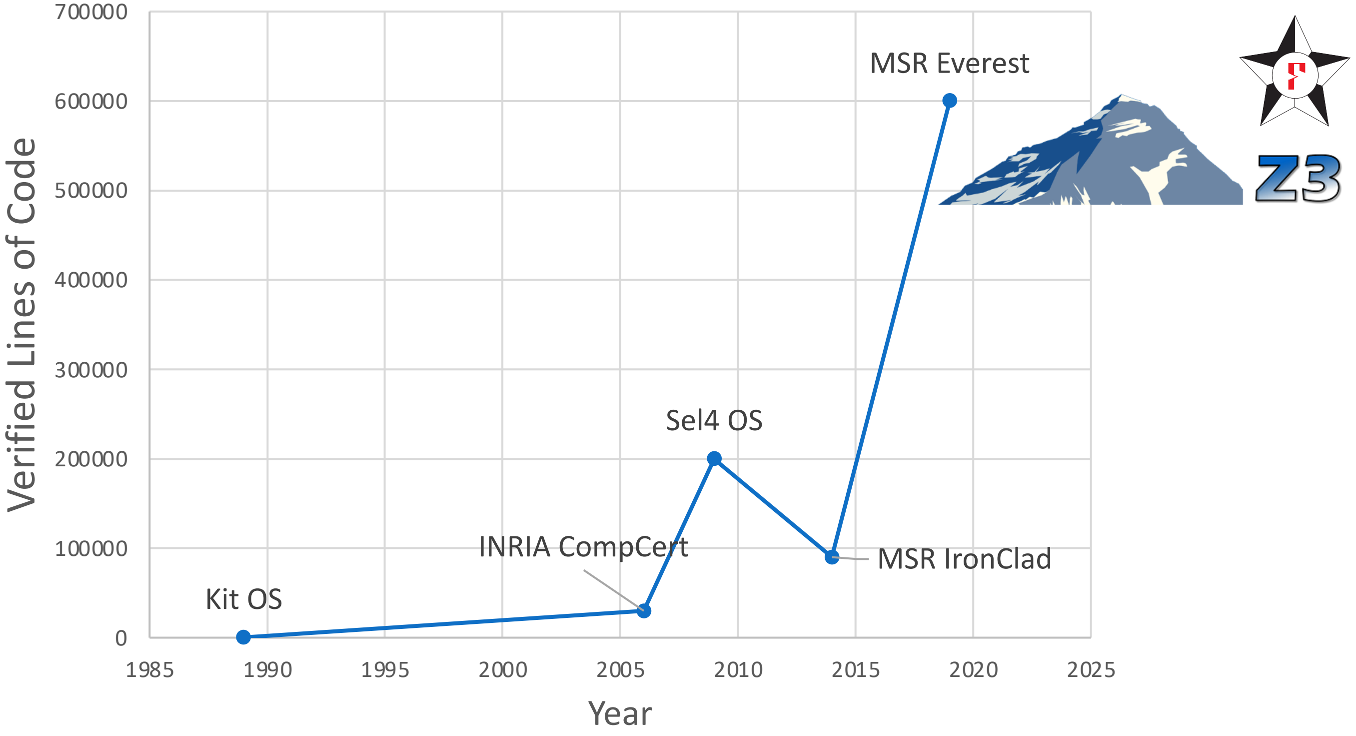
# Building and Deploying Verified Secure Communication Components

# Programming with proofs, at scale

Using F*, a new verification oriented programming language developed at MSR

600,000 lines of code and proof under continuous integration

100s of builds a week with checking proofs of functional correctness on an evolving systems



# Program proofs in F* for billions of unsuspecting users

Automated parsing untrusted data with proofs in Hyper-V/VMSwitch

Microsoft Hyper-v

Microsoft SQL Azure

Verified Merkle trees for Enterprise blockchains

Verified cryptography in the Linux kernel

TLS 1.3

QUIC

WireGuard
FAST, MODERN, SECURE VPN TUNNEL

Signal

CRYPTOGRAPHY

Quic transport,
MSQuic in Windows,
Verified crypto in Firefox,
mbedTLS,
Signal in Wasm,
Wireguard, …

Financial technology

Undisclosed 3rd parties rewriting core trading specifications and algorithms in F*

Verifiable multi-part computations

Veritas
Wysteria

Proven correct high performance verifiable key-value stores

ElectionGuard

Verified crypto for ElectionGuard e-Voting SDK

# What do we prove?

## Safety

Memory- and type-safety.  Mitigates buffer overruns, dangling pointers, code injections.

## Functional correctness

Our fast implementations behave precisely as our simpler specifications.

## Secrecy

Access to secrets, including crypto keys and private app data is restricted according to design.

## Cryptographic security

We bound the probability that an attacker may break any secrecy or integrity properties

Our specifications and implementations are written together, in one language (F*)
Drift between spec and implementation cannot happen.

# Incremental deployment of verified software

## Whole stack replacement with formally proven software is a great goal

- But, we need to incrementally evolve the current stack to get there

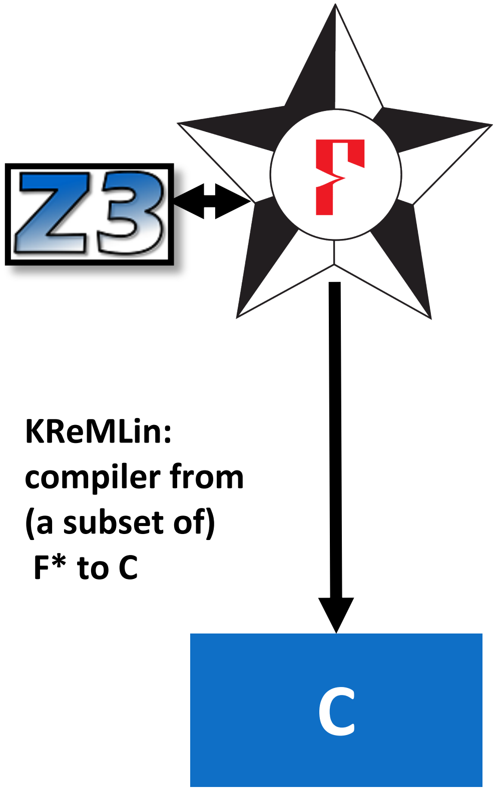
## Carefully identify components that underpin the security and correctness of high-value systems

- Cryptographic components: primitives, constructions, and standardized protocols
- Components that mediate access across trust boundaries, e.g., system call boundaries, virtualization interfaces, …

## Program and prove drop-in replacements for those components

- Bring decades of research in formal proofs to the real world, improving tools, methodologies
- Harden existing systems with measurable benefits to overall system security

# F*: A first example




KReMLin:
compiler from
(a subset of)
F* to C

**F* implementation and specification**

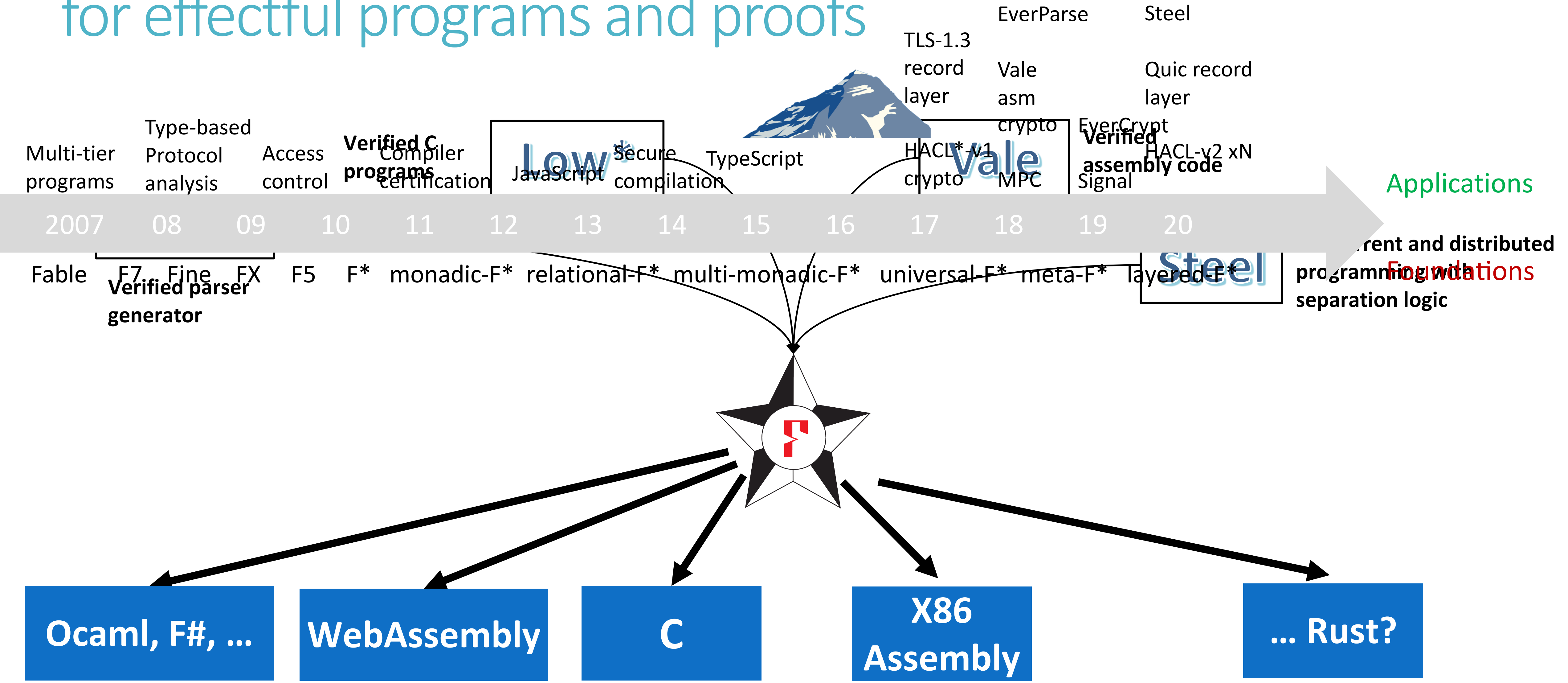
```
let multiply_by_9 (a:uint32) : Pure uint32
  (requires 9 * a <= MAX_UINT_32)
  (ensures λ result -> result == 9 * a)
  =
  let b = a << 3ul in
  a + b
```

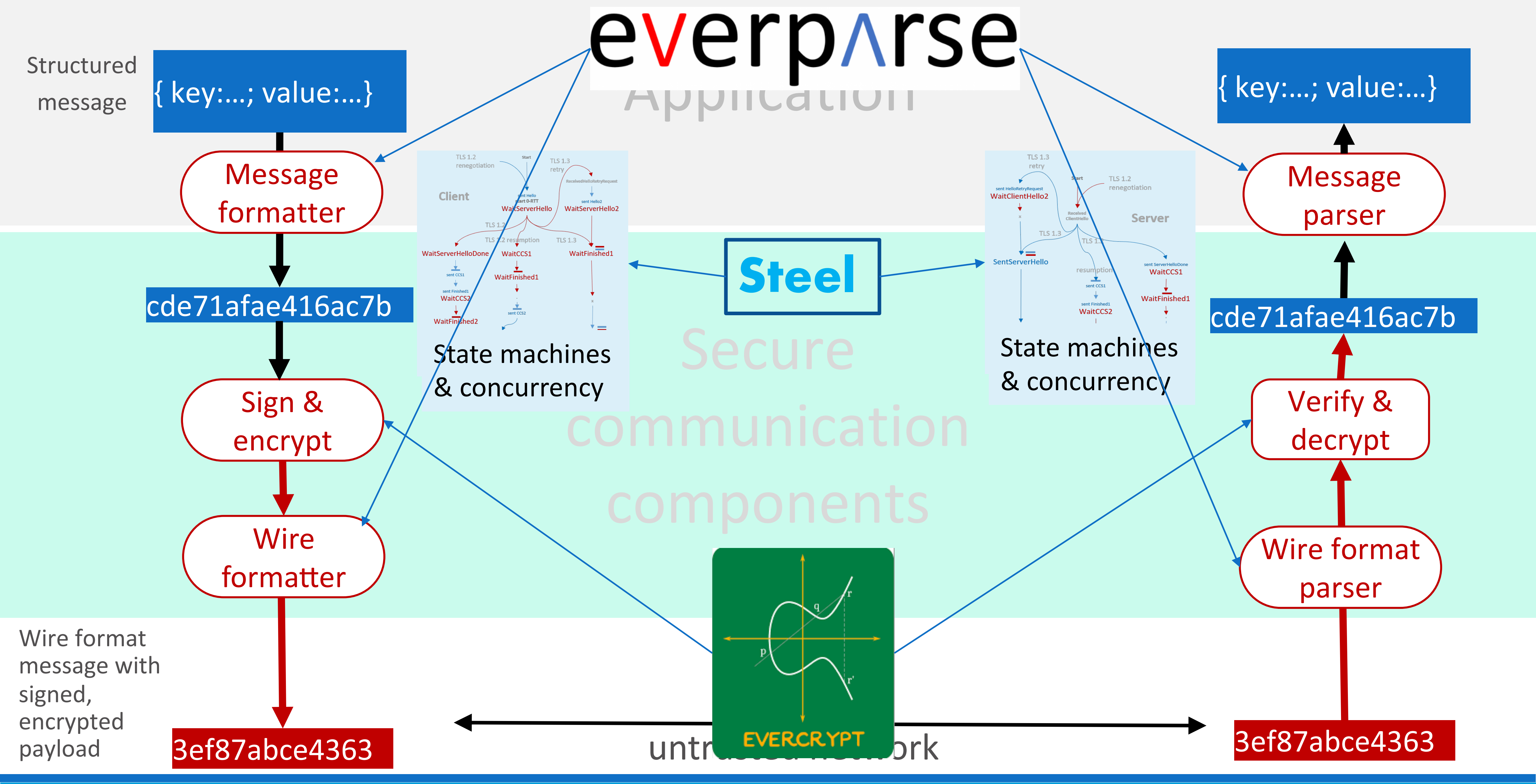
**Efficient C implementation**

Verification imposes no runtime performance overhead

```
uint32_t multiply_by_9(uint32_t a)
{
  uint32_t b = a << (uint32_t)3;
  return a + b;
}
```

# F*: A logical framework for effectful programs and proofs

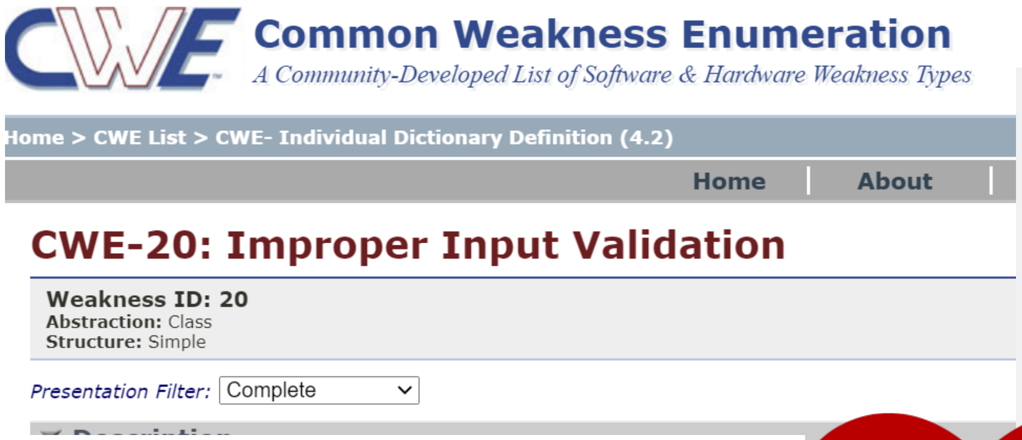



everparse
Application
Structured message
{ key:…; value:…}
Message formatter
cde71afae416ac7b
Sign & encrypt
Wire formatter
Wire format message with signed, encrypted payload
3ef87abce4363
State machines & concurrency
Steel
Secure communication components
State machines & concurrency
{ key:…; value:…}
Message parser
cde71afae416ac7b
Verify & decrypt
Wire format parser
3ef87abce4363
EVERCRYPT
untrusted network

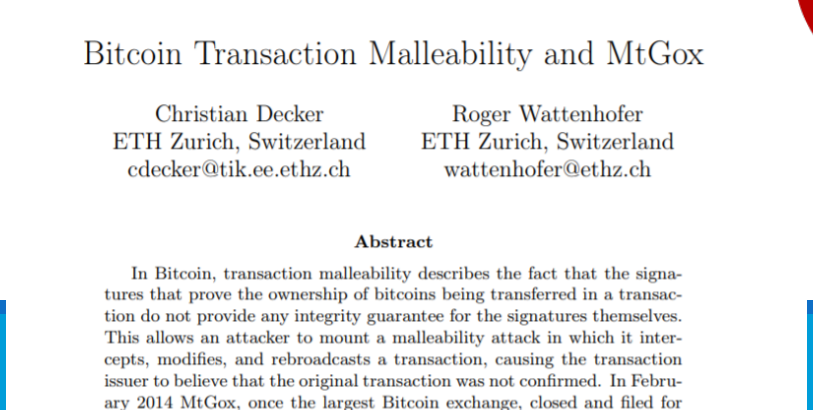
everparse

A Mathematically Proven, Efficient, Low-level Parser Generator

# Improper parsing of attacker-controlled input







Bitcoin Transaction Malleability and MtGox

Christian Decker
ETH Zurich, Switzerland
cdecker@tik.ee.ethz.ch

Roger Wattenhofer
ETH Zurich, Switzerland
wattenhofer@ethz.ch

Abstract

In Bitcoin, transaction malleability describes the fact that the signatures that prove the ownership of bitcoins being transferred in a transaction do not provide any integrity guarantee for the signatures themselves. This allows an attacker to mount a malleability attack in which it intercepts, modifies, and rebroadcasts a transaction, causing the transaction issuer to believe that the original transaction was not confirmed. In February 2014 MtGox, once the largest Bitcoin exchange, closed and filed for

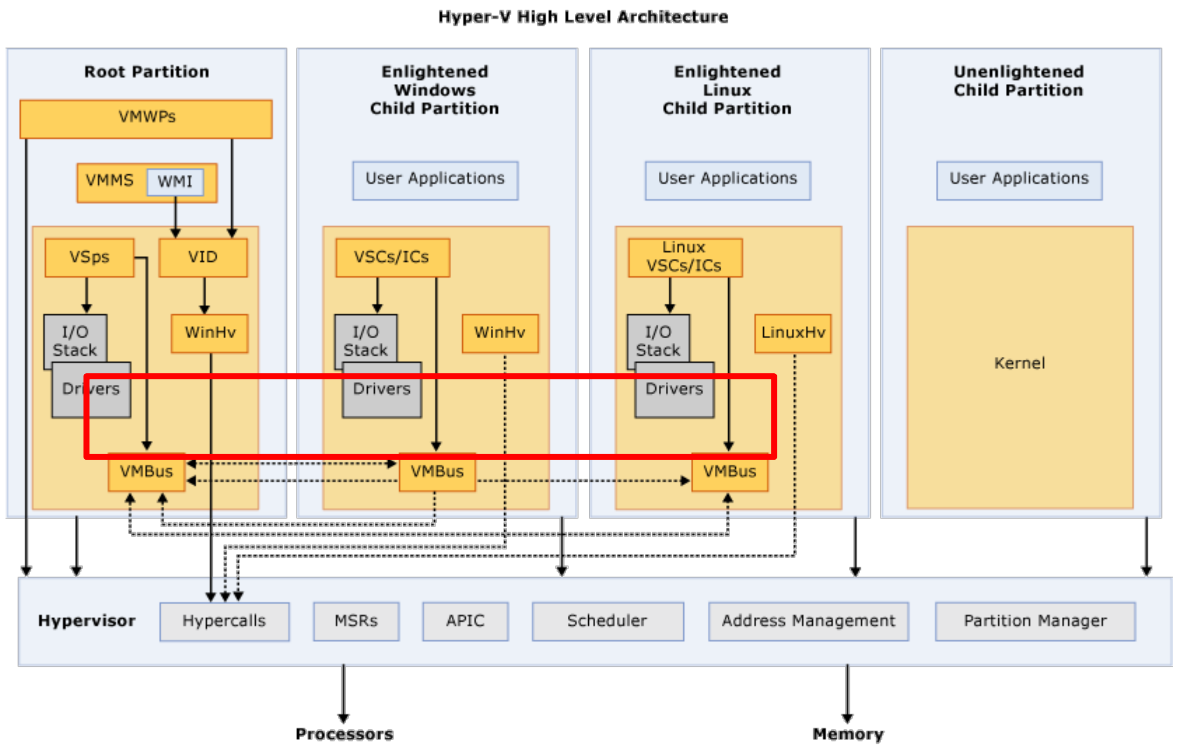
"The leakage was the result of a bug in an HTML parser chain Cloudflare uses to modify webpages as they pass through the service's edge servers. […]. When the parser was used in combination with three Cloudflare features […] it caused Cloudflare edge servers to leak pseudo random memory contents into certain HTTP responses."

# Microsoft Hyper-V: Traversing trust boundaries  deep in the software stack

Hyper-V is the core technology isolating virtual machines in the Azure cloud

Virtualized devices exposed to enlightened guests

Attack Surface: Untrusted guest VM can send malformed packets to host kernel, E.g. trying to overflow the packet buffer



# Data validation is challenging at the guest/host boundary

- Handwritten code to validate messages from untrusted guests

- Tricky to write for several reasons
  - Many variable-length structures and data-dependent unions
  - Avoiding arithmetic overflow
  - Layered protocols, with multiple headers to be parsed incrementally
  - Sometime dealing with shared memory … hard to be sure
    - Double fetches can lead to time-of-check/time-of-use bugs

- Legacy C code, hard to deploy even basic modern bounds-checking measures, e.g., C++ spans, Rust etc.,
  - Plus bounds checking comes with runtime overhead

# everparse

## A Mathematically Proven Low-level Parser Generator

Our long-term goal

- Abolish writing low-level binary format parsers by hand
- Instead, specify formats in a high-level declarative notation
- Auto-generate performant, verified low-level code to parse binary messages
- Integrate seamlessly with existing codebases in a variety of languages (C, C++, Rust, …)

With formal proofs that the code is:

- Memory safe (no access out of bounds, no use after free etc.)
- Arithmetically safe (no overflow/underflow)
- Functionally correct (that it parses exactly those messages that conform to the high-level spec)
- Free from double-fetches, so safe against time-of-check/time-of-use bugs

https://project-everest.github.io/everparse/

# everparse

## A Mathematically Proven Low-level Parser Generator

Our long-term goal

- Abolish writing low-level binary format parsers by hand
- Instead, specify formats in a high-level declarative notation
- Auto-generate performant, verified low-level code to parse binary messages
- Integrate seamlessly with existing codebases in a … of languages (C, C++, R…

Provably correct by construction:
Zero user proof effort

With formal proofs that the code is:

- Memory safe (no access out of bounds, no use after free etc.)
- Arithmetically safe (no overflow/underflow)
- Functionally correct (that it parses exactly those messages that conform to the high-level spec)
- Free from double-fetches, so safe against time-of-check/time-of-use bugs

https://project-everest.github.io/everparse/

## Starting from a high-level language of message formats

## EverParse auto-generates F* parsing code that is

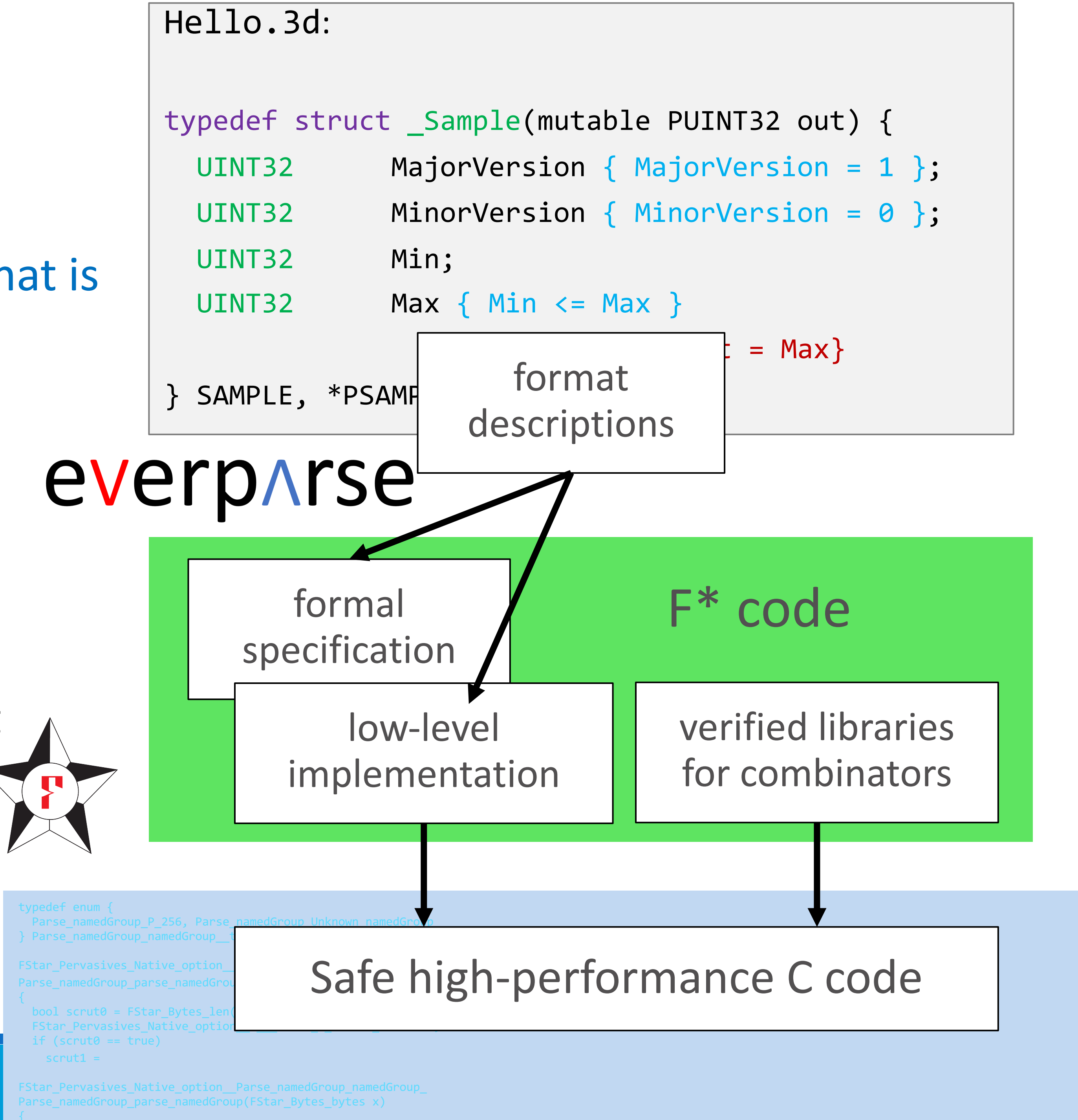
- Safe
- Correct
- Fast (zero-copy)

Correctness:

```
parse (serialize msg) = msg
valid msg ==> serialize (parse msg) = msg
```

Performance:

similar to or better than handwritten code



# Dependent Data Descriptions in 3D: A source language of message formats

Constraints and actions augmenting C data types

```
typedef struct _Sample(mutable PUINT32 out) {
  UINT32      MajorVersion { MajorVersion = 1 };
  UINT32      MinorVersion { MinorVersion = 0 };
  UINT32      Min;
  UINT32      Max { Min <= Max }
                  {:on-success *out = Max}
} SAMPLE, *PSAMPLE;
```

# Dependent Data Descriptions in 3D: A source language of message formats

Constraints and actions augmenting C data types

```
typedef union _MessageUnion {
    Init  init;
    Query query;
    Halt  halt;
} MessageUnion;
```

```
typedef struct _Message {
   UINT32 tag;
   MessageUnion message;
} Message;
```

# Dependent Data Descriptions in 3D: A source language of message formats

Constraints and actions augmenting C data types

```
casetype _MessageUnion(UINT32 tag) {
  switch(tag) {
  case INIT_MSG:
    Init  init;
  case QUERY_MSG:
    Query query;
  case HALT_MSG:
    Halt  halt;
  }
} MessageUnion;
```

```
typedef struct _Message {
   UINT32 tag;
   MessageUnion(tag) message;
} Message;
```

# A Sample 3D Specification for several variable-length structures

```
/*++
  |-----------DataOffset--------------------|----DataLength-----|

  |-------------PacketLength-------------------------------------------|

  |-----------HeaderLength---------------------|

  |---sizeof(this)--|

  |-----Offset2------|----DataLength2-----|
 --*/
entrypoint
typedef struct __SOME_PACKET (UINT32 PacketLength,
                              UINT32 HeaderLength,
                              mutable UINT32 *dataOffset,
                              mutable UINT32 *dataLength,
                              mutable UINT32 *offset2,
                              mutable UINT32 *length2)
  where (sizeof(this) <= HeaderLength &&
         HeaderLength <= PacketLength)
{
  UINT32                              DataOffs
     { sizeof(this) <= DataOffset }
     {:on-success
        *dataOffset = DataOffset;
        return true;
     };
  UINT32                              DataLeng
     { is_range_okay(PacketLength, DataOffset, DataLength) }
     {:on-success
        *dataLength = DataLength;
        return true;
     };
  UINT32                              FieldA
     { FieldA <= Bound_A };
  UINT32                              FieldB
     { FieldB <= Bound_B };
```

```
UINT32                          NumExtraElements
   { NumExtraElements <= MAX_EXTRA_ELEMENTS };
UINT32                          Offset2
   {:on-success
      *offset2 = Offset2;
      return true;
   };
                                DataLength2

                    ))

                   aderLength, Offset, DataLength2) &&
                   (this) &&
                   gth2 <= DataOffsset

   {:on-success
      *dataLength2 = DataLength2;
      return true;
   };
UINT32                          Reserved;
} SOME_PACKET;
```

- Constraints
- Actions with mutable outparameters
- Variable-length data at designated offsets
- Tagged unions
- …

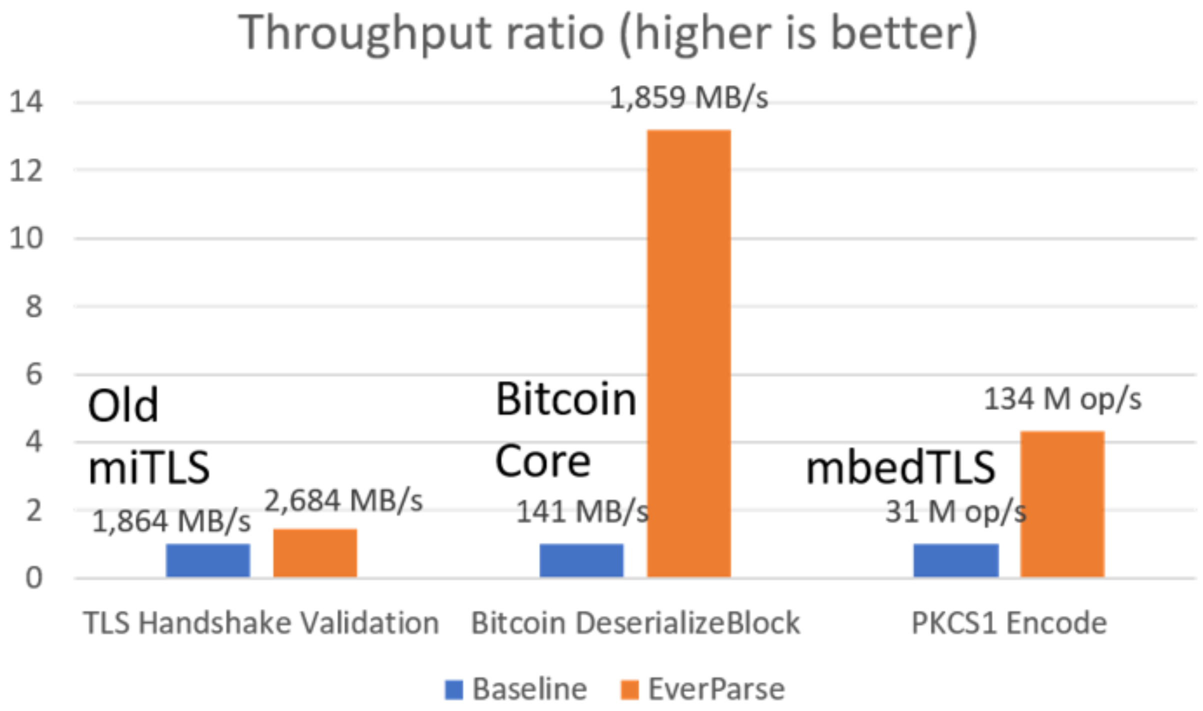
# Generated C code, after verification

- C code aims to be human-readable, human patchable
  - Propagates comments from source spec
  - Generates predictable descriptive names

```
BOOLEAN
CheckPacket(
    uint32_t ___PacketLength,
    uint32_t ___HeaderLength,
    uint32_t *dataOffset,
    uint32_t *dataLength,
    uint32_t *offset2,
    uint32_t *dataLength2,
    uint8_t  *base,
    uint32_t len);
```

# Some case studies

| | Data Types | Spec | F* LoC |
|---|---|---|---|
| TLS 1.2-1.3 | 315 | 1601 | 70k |
| Bitcoin | 6 | 31 | 2k |
| PKCS1 | 19 | 117 | 5k |
| LowParse | | | 33k |




- We can express real world formats
- We scale to large and complex schemas
- We produce high-performance code

# EverParse Takeaways: A Sweet Spot

- Good return on investment
  - Parsing bugs => security vulnerabilities exploitable from the attack surface
  - Focus defense efforts on parsing code

- EverParse: Push-button proofs and code-generation for low-level parsers
  - Strong mathematical guarantees of safety and correctness
  - Provably correct by construction: Zero user proof effort

- It works in Windows and Microsoft Azure today, securing the parsing of every packet that passes through the networking stack

# Industrial-grade verified cryptography at scale

# Efficient crypto requires a lot of customizations

## Poly1305: Uses the prime field with $p = 2^{130} - 5$

- Need 130 bits to represent a number
- Efficient implementations require custom bignum libraries to delay carries
- On X86: use 5 32-bit words, but using only 26 bits in each word
- On X64: use 3 64-bit words, but using only 44 bits in each word

## Curve25519: Uses the prime field with $p = 2^{255} - 19$

- On X64: use 5 64-bit words, but using only 51 bits per word

## OpenSSL has 12 unverified bignum libraries optimized for each case

# Many bugs in Curve25519 implementations

(C and assembly)

agl / curve25519-donna

Watch

Code | Issues 2 | Pull requests 7 | Projects 0 | Wiki | Insights

**Correct bounds in 32-bit code.**

```
The 32-bit code was illustrative of the tricks used in the original
curve25519 paper rather than rigorous. However, it has proven quite
popular.

This change fixes an issue that Robert Ransom found where outputs between
2^255-19 and 2^255-1 weren't correctly reduced in fcontract. This
appears to leak a small fraction of a bit of security of private keys.

Additionally, the code has been cleaned up to reflect the real-world
needs. The ref10 code also exists for 32-bit, generic C but is somewhat
slower and objections around the lack of qhasm availibility have been
raised.
```

master 1.3

Curve25519-donna

agl committed on Jun 9, 2014 1 parent

```
sv pack25519(u8 *o
{
  int i,j,b;
  gf m,t;
  FOR(i,16) t[i]=n
  car25519(t);
  car25519(t);
  car25519(t);
  FOR(j,2) {
    m[0]=t[0]-0xff
    for(i=1;i<15;i
      m[i]=t[i]-0x
      m[i-1]&=0xff
    }
    m[15]=t[15]-0x
    b=(m[15]>>16)&
    m[15]&=0xffff;
    sel25519(t,m,1-b);
  }
  FOR(i,16) {
    o[2*i]=t[i]&0xff;
    o[2*i+1]=t[i]>>8;
  }
}
```

TweetNaCl

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;`.

Ed25519 amd64 bug

gistfile1.md Raw

NaCl (asm)

While visiting 30c3, I attended the You-broke-the-Internet workshop on NaCl.

One thing mentioned in the talk was that auditing crypto code is a lot of work, and that this is one of the reasons why Ed25519 isn't included in NaCl yet (they promised a version including it for 2014). The speakers mentioned a bug in the amd64 assembly implementation of Ed25519 as an example of a bug that can only be found by auditing, not by randomized tests. This bug is caused by a carry being added in the wrong place, but since that carry is usually zero, the bug is hard to fint (occurs with probability 2^{-60} or so).

The TweetNaCl paper briefly mentions this bug as well:

> Partial audits have revealed a bug in this software ( `r1 += 0 + carry` should be `r2 += 0 + carry` in `amd64-64-24k` ) that would not be caught by random tests; this illustrates the importance of audits.

Searching for this string in the SUPERCOP source code turns up four matches:

```
crypto_scalarmult\curve25519\amd64-64\fe25519_mul.s
crypto_scalarmult\curve25519\amd64-64\fe25519_square.s
crypto_sign\ed25519\amd64-64-24k\fe25519_mul.s
crypto_sign\ed25519\amd64-64-24k\fe25519_square.s
```

So it appears like the `amd64-64` implementation of both Curve25519 and Ed25519 is affected.

It seems difficult to exploit this when used for key generation or signing since the attacker cannot influence the data. Key-exchange and signature verification might be a problem.

# 3 Bugs in OpenSSL implementation of Poly1305

OpenSSL Security Advisory [10 Nov 2016]

[openssl-dev] [openssl.org #4439] poly1305-x86.pl produces incorrect output

"These produce wrong results. The first example does so only on 32 bit, the other three also on 64 bit."

"I believe this affects both the SSE2 and AVX2 code. It does seem to be dependent on this input pattern."

"I'm probably going to write something to generate random inputs and stress all your other poly1305 code paths against a reference implementation."

recommend doing the same in your own test harness, to make sure there aren't others of these bugs lurking around.

| Algorithm | Portable C (HACL*) | Intel ASM (Vale) | Agile API (EverCrypt) |
| --- | --- | --- | --- |
| **AEAD** | | | |
| AES-GCM | | ✔ (AES-NI + CLMUL) | ✔ |
| Chacha20-Poly1305 | ✔ (+ AVX,AVX2) | | ✔ |
| **ECDH** | | | |
| Curve25519 | ✔ | ✔ (BMI2 + ADX) | |
| P-256 | ✔ | | |
| **Signatures** | | | |
| Ed25519 | ✔ | | |
| P-256 | ✔ | | |
| **Hashes** | | | |
| MD5 | ✔ | | ✔ |
| SHA1 | ✔ | | ✔ |
| SHA2-224,256 | ✔ | ✔ (SHAEXT) | ✔ |
| SHA2-384,512 | ✔ | | ✔ |
| SHA3 | ✔ | | |
| Blake2 | ✔ (+ AVX,AVX2) | | |
| **Key Derivation** | | | |
| HKDF | ✔ | ✔ (see notes below) | ✔ |
| **Ciphers** | | | |
| Chacha20 | ✔ (+ AVX,AVX2) | | |
| AES-128,256 | | ✔ (AES-NI + CLMUL) | |
| **MACS** | | | |
| HMAC | ✔ | ✔ (see notes below) | ✔ |
| Poly1305 | ✔ (+ AVX,AVX2) | ✔ (X64) | |

# EverCrypt

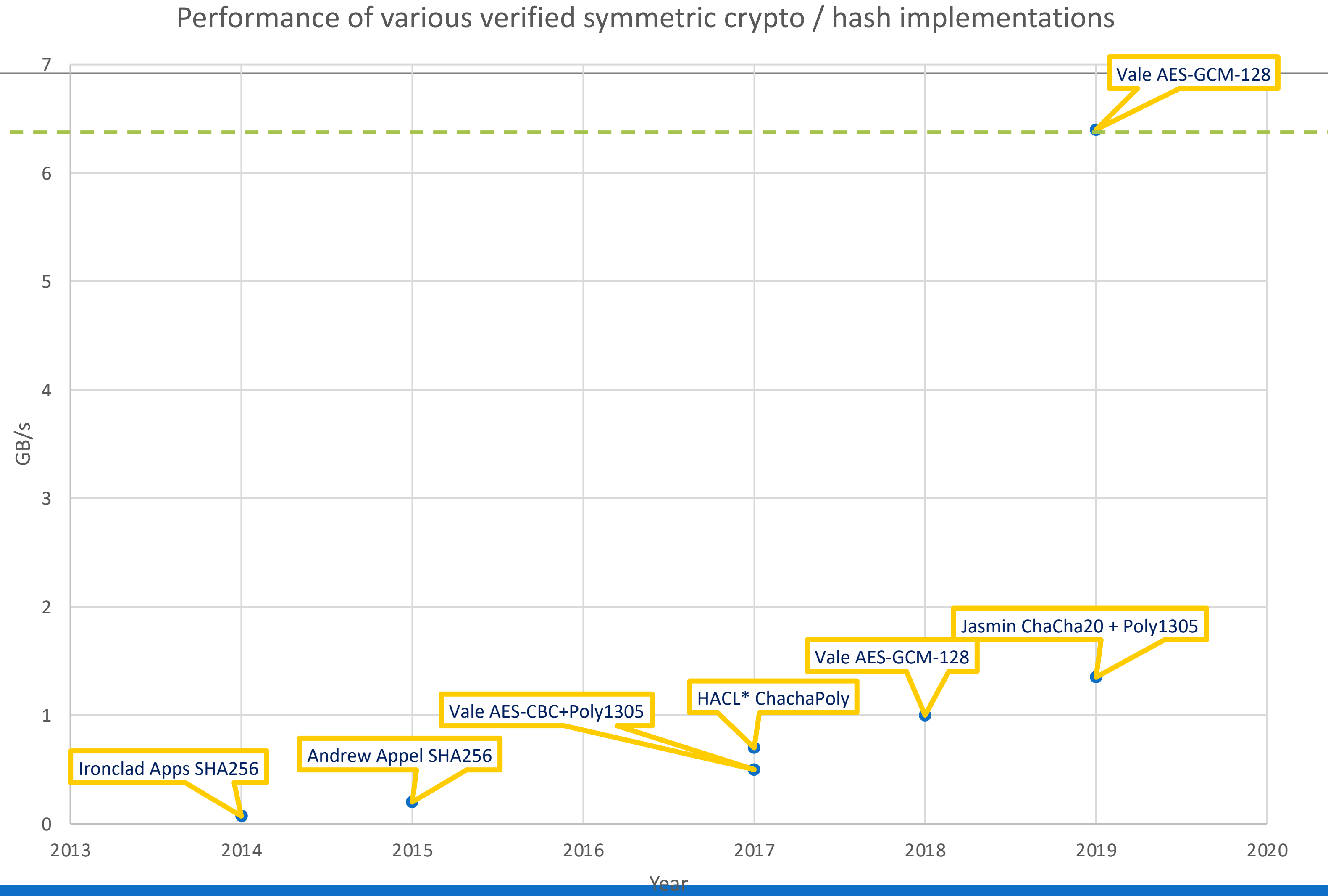
**A verified, no-excuses, industrial-grade cryptographic provider.**

**A replacement for:** OpenSSL, BCrypt, libsodium.
- A *collection* of algorithms **(exhaustive)**
- Easy-to-use API **(CPU auto-detection)**
- Several *implementations* **(multiplexing)**
- APIs grouped by *family* **(agility)**

Clients get **state-of-the art performance.**

- 140,000 lines of Low* and Vale
- 43,000 lines of C + 15,000 lines of ASM

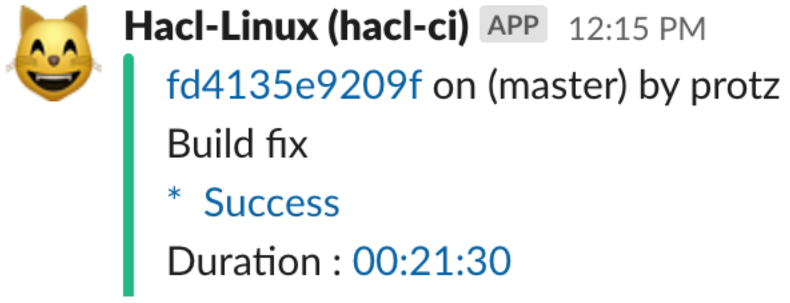
# Finally: speed **and** safety




Fastest OpenSSL assembly code

# A toolkit for scaling verification

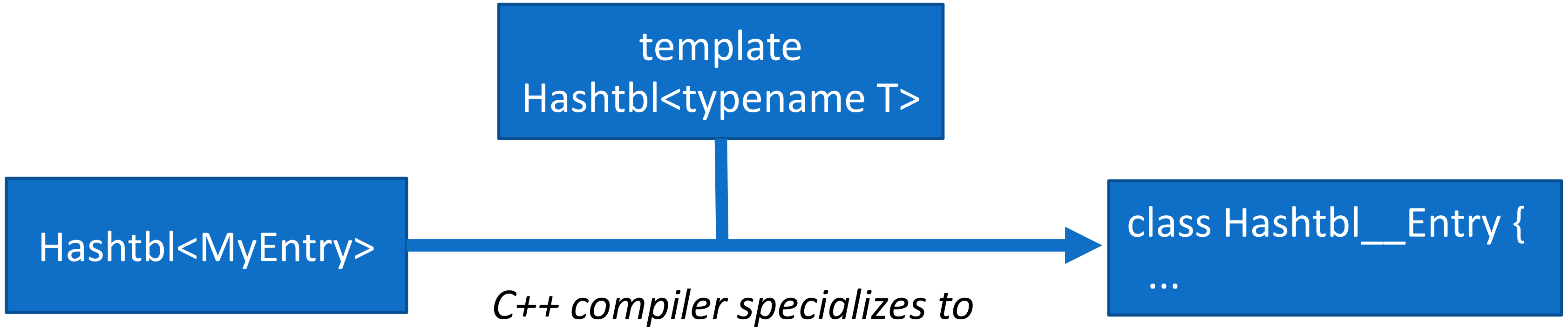
- CI, build, regressions: single greatest productivity improvement
- Understanding packaging & distribution for deployments
- Reducing complexity: a subset of Low* for cryptography (students)
- Hybrid style for more robust proofs (e.g. calc, tactics)
- External collaborations thanks to open-source




More importantly:

- **Many flavors of meta-programming to slash the proof-to-code ratio**

# Meta-programming in a nutshell

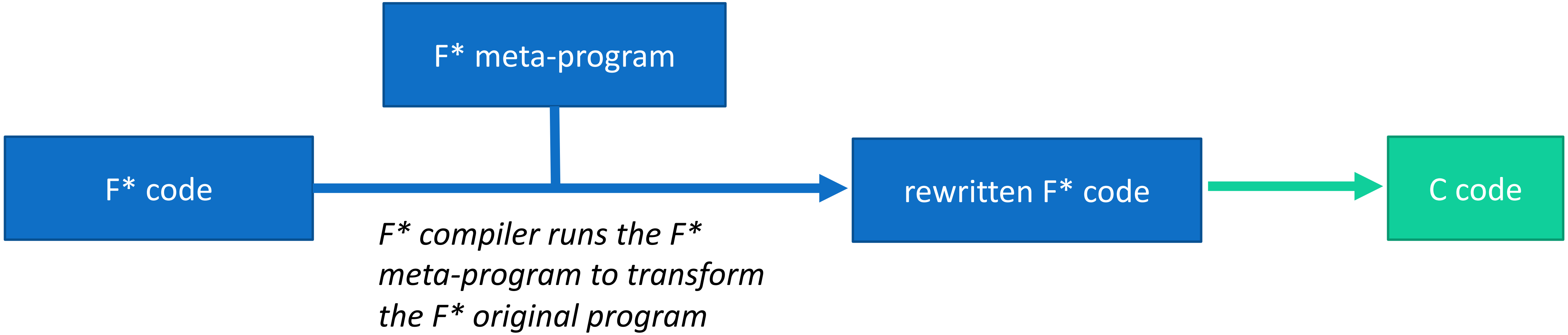



C++: Zero-cost abstraction, but limited expressive power

What we want:

- finer-grained control
- many flavors of meta-programming (partial evaluation, unrolling, rewriting)
- fully verified

# Meta-programming in F*




- Script the compiler
- Driving the generation of C code
- Any flavor of metaprogramming
- No increased trust boundary

# template HPKE<AEAD,DH,Hash>

Encode template specialization logic as a meta-program; **TCB unchanged**

- 1 generic implementation (write once)
- 3 existing DH implementations
- 11 existing hash implementations
- 5 existing AEAD implementations
- 165 possible combinations, we choose 14

**1565 lines of F* → 5820 lines of C code**

Any combination is valid: HPKE thus has 24 possible ciphersuites, and many more *implementation* combinations.
Individually verifying all these would be intractable. However, using the integrated HACL* library, we can build a *generic* implementation of HPKE in 800 lines of code, in a way that is abstract in the choice of its KEM, KDF and AEAD implementation. To instanti-

*proof to code ratio: 0.27*

# Meta-programming everywhere

- Integers: encode overloaded operators for 14 types of machine integers
- Algorithmic flavors (e.g. SHA, Blake, Curve)
- Type classes
- Functors for high-level crypto APIs that capture the essence of agility
- Loop unrolling in F*

My favorite:

- Vectorization: overloaded operators for vector types; write once, compile N times (CCS'20)

**Language improvements for the next order of magnitude.**

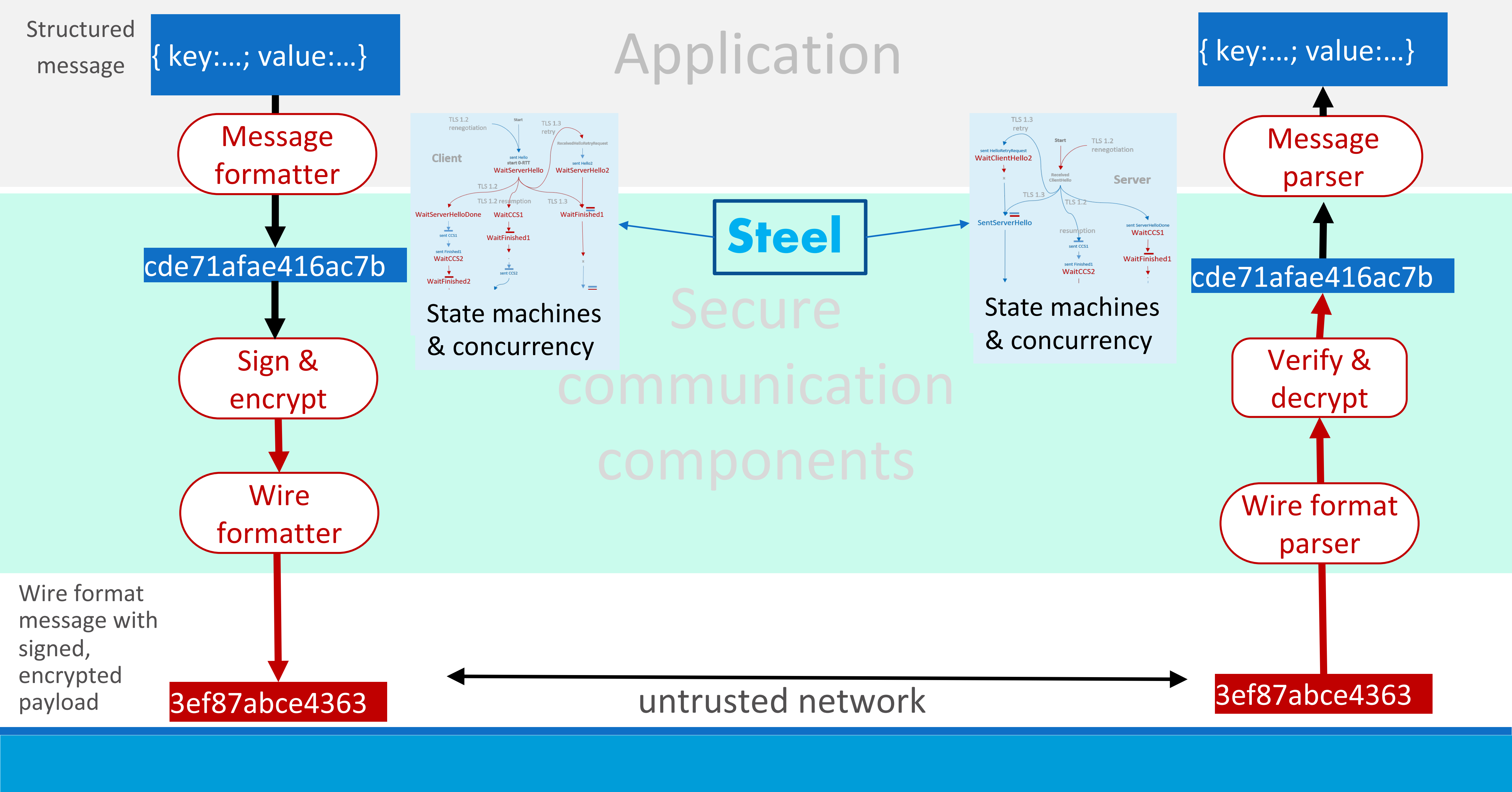
# Production deployments of Everest Verified Cryptography

All using Everest verified crypto

MSQuic integrates Everest TLS 1.3 and crypto

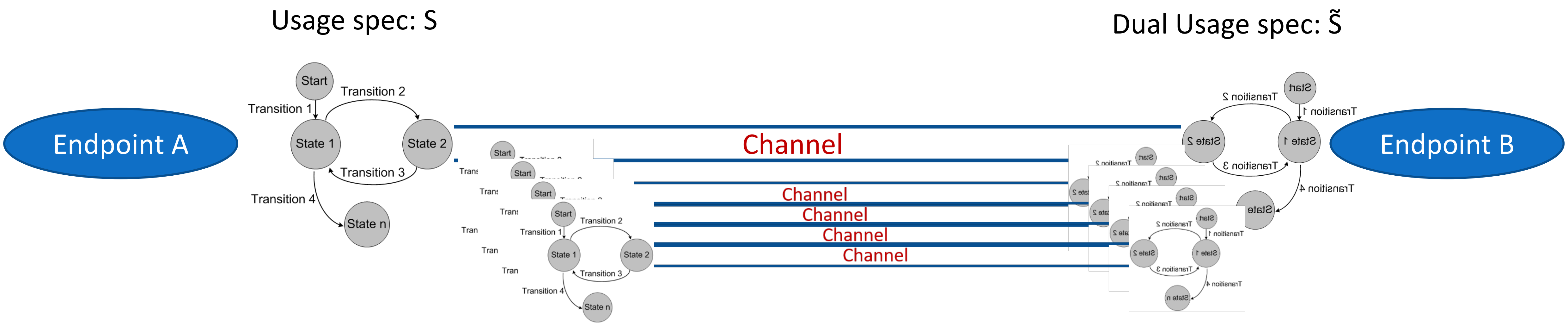
Using Everest crypto and verified Merkle trees in Azure Confidential Computing

# Layered abstractions for state, concurrency, and distribution in Steel



Structured message
{ key:…; value:…}
Application
Message formatter
cde71afae416ac7b
Sign & encrypt
Wire formatter
Wire format message with signed, encrypted payload
3ef87abce4363
Client
State machines & concurrency
Steel
Secure communication components
Server
State machines & concurrency
untrusted network
{ key:…; value:…}
Message parser
cde71afae416ac7b
Verify & decrypt
Wire format parser
3ef87abce4363

# Usage control on channels



# Session typed channels in Steel

```
let pingpong =
  x ⟵ Protocol.send ℤ;
  y ⟵ Protocol.recv (y:ℤ{y > x});
  Protocol.done

let client_server ()
  = let c = new_chan pingpong in
    par (client c) (server c)

let many (n:ℕ) = join (spawn n client_server)
```

```
let client (c:chan pingpong)
  = send c 17;
    let y = recv c in
    assert (y > 17);
    return ()

let server (c:chan pingpong)
  = let y = recv c in
    send c (y + 42);
    return ()
```

# Raising the level of abstraction in Steel

- Libraries for locks on built on top of primitive atomic instructions
- POSIX-style fork/join using structured parallelism & locks
- Libraries for message passing on channels built as an additional layer
- …

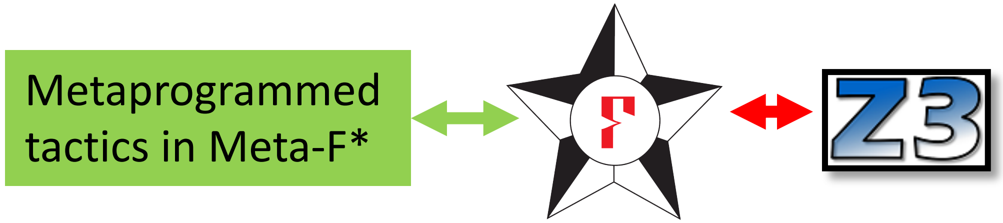
```
module Steel.Effect.Atomic
module Steel.SpinLock
module Steel.Primitive.ForkJoin
module Steel.Channel.Duplex
module Steel.Channel.BinaryFormatted
module Steel.Channel.Cryptographicall
module Steel.Channel.Secure
```

# SteelCore: A Foundational Concurrent Separation Logic Embedded in F*

Specify programs in concurrent separation logic

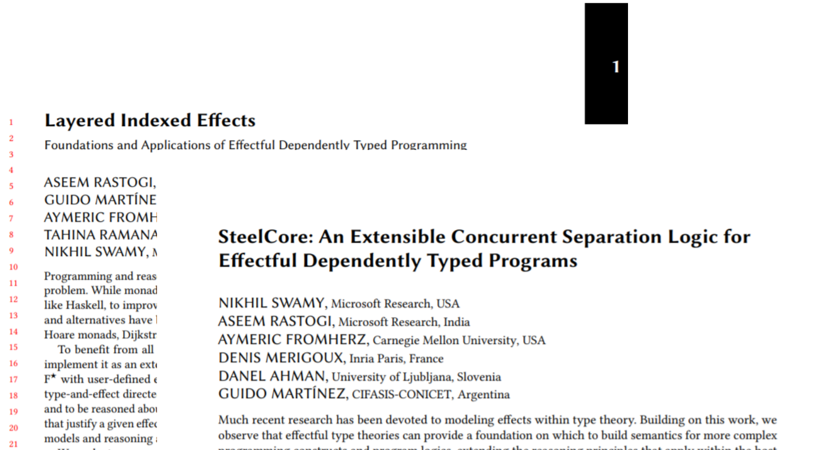
Verify programs using F* metaprograms, tactics and Z3

```
  { P } e { Q }
---------------------------- [Frame]
  { P * F } e { Q * F }
```




https://www.fstar-lang.org/papers/steelcore/

```
  { P1 } e1 { Q1 }
  { P2 } e2 { Q2 }
------------------------------------ [Par]
  { P1 * P2 } e1 || e2 { Q1 * Q2 }
```



1

**Layered Indexed Effects**

Foundations and Applications of Effectful Dependently Typed Programming

ASEEM RASTOGI,
GUIDO MARTÍNE
AYMERIC FROMH
TAHINA RAMANA
NIKHIL SWAMY,

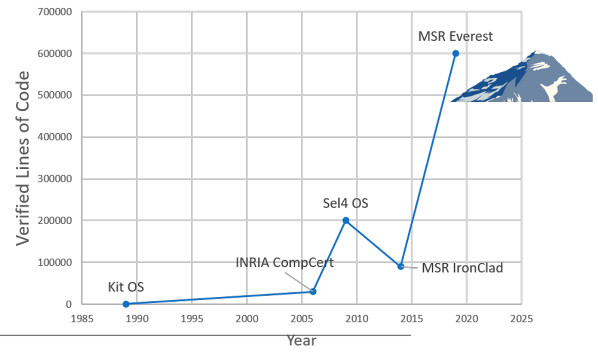
Programming and reas
problem. While monad
like Haskell, to improv
and alternatives have
Hoare monads, Dijkstr
To benefit from all
implement it as an ext
F* with user-defined
type-and-effect directe
and to be reasoned abo
that justify a given effe
models and reasoning

**SteelCore: An Extensible Concurrent Separation Logic for Effectful Dependently Typed Programs**

NIKHIL SWAMY, Microsoft Research, USA
ASEEM RASTOGI, Microsoft Research, India
AYMERIC FROMHERZ, Carnegie Mellon University, USA
DENIS MERIGOUX, Inria Paris, France
DANEL AHMAN, University of Ljubljana, Slovenia
GUIDO MARTÍNEZ, CIFASIS-CONICET, Argentina

Much recent research has been devoted to modeling effects within type theory. Building on this work, we observe that effectful type theories can provide a foundation on which to build semantics for more complex

# Takeaways




We verify & deploy reusable, critical software components at scale

- Fully verifying or hardening critical subsystems
- Achieving high performance & usability

We aim to lower the bar and scale proofs further by

- Proof and code generation from domain-specific languages
- Metaprogramming to specialize, partially evaluate and optimize code
- Raising the level of abstraction in libraries for state, concurrency & distribution

# Our ambitions

## Research

**Advance the state of the art in developing high-assurance critical systems**

Systems Programming, Cryptography, Security, Privacy, Scalable Proofs of Programs

## Applications

**Get formally-verified code in production**

**Deliver the strongest technical correctness, security & privacy guarantees to our customers**