F* and Meta-F*

Formal Verification, Language Extensibility, and Proof automation

Nik Swamy
Guido Martinez

As applied in Project Everest

https://fstar-lang.github.io
https://project-everest.github.io
TLS: Transport Layer Security

- Most widely deployed security? ½ Internet traffic (+40%/year)
- Web, cloud, email, VoIP, 802.1x, VPNs, ...
Threat model

Goal: A secure channel

connect(server,port);
send "GET…";
data = recv();
send "POST…";
...
accept(port);
request = recv();
send "<html>…";
order = recv();
...

20 years of attacks & fixes

Buffer overflows
Incorrect state machines
Lax certificate parsing
Weak or poorly implemented crypto
Side channels
Informal security goals
Dangerous APIs
Flawed standards

Mainstream implementations
OpenSSL, SChannel, NSS, …

As long as the adversary does not control the long-term credentials of the client and server, it cannot

• Inject forged data into the stream (authenticity)
• Distinguish the data stream from random bytes (confidentiality)
TLS 1.3: a new hope

Much discussions
IETF, Google, Mozilla, Microsoft, CDNs, cryptographers, network engineers, ...

Much improvements
- Modern design
- Fewer roundtrips
- Stronger security

New implementations required for all
- An early implementer and verified too!
- Find & fix flaws before it’s too late

RFC 8446: Aug 2018
Including many of our proposals

#4 log-based key separation
extended session hashes
(fixing attacks we found on 1.2)

#11 stream terminators
(eventually fixing an attack)

#14 downgrade resilience

#15 session ticket format

#17 simplified key schedule
pre-shared-key 0RTT

#18 PSK binding (fixing an attack)

Mentioning many formal models of the protocol, including our verified implementation of the record layer
Project Everest
Verified Secure Components in the TLS Ecosystem

- Strong verified security
- Widespread deployment
- Trustworthy, usable tools
- Growing expertise in high-assurance software development
- Open source
**Verification Tools and Methodology**

F*: A general purpose programming language and verification tool

kreMLin Compiler from (a subset of) F* to C

---

### Math spec in F*

poly1305_mac computes a polynomial in GF(2^{130} - 5), storing the result in tag, and not modifying anything else.

```f
val poly1305_mac: tag:nbytes 16 →
  len:u32 →
  msg:nbytes len{disjoint tag msg} →
  key:nbytes 32 {disjoint msg key ^ disjoint tag key} →
  ST unit

(requires (\lambda h \rightarrow msg \in h ^ key \in h ^ tag \in h))
(ensures (\lambda h0 _ h1 →
  let r = Spec.clamp h0.[sub key 0 16] in
  let s = h0.[sub key 16 16] in
  modifies {tag} h0 h1 ^
  h1.[tag] = Spec.mac_1305 (encode_bytes h0.[msg]) r s))
```

---

### Efficient C implementation

Verification imposes no runtime performance overhead.

```c
void poly1305_mac(uint8_t *tag, uint32_t len, uint8_t *msg, uint8_t *key)
{
  uint64_t tmp [10] = { 0 };
  uint64_t *acc = tmp
  uint64_t *r = tmp + (uint32_t)5;
  uint8_t s[16] = { 0 };
  Crypto_Symmetric_Poly1305_poly1305_init(r, s, key);
  Crypto_Symmetric_Poly1305_poly1305_process(msg, len, acc, r);
  Crypto_Symmetric_Poly1305_poly1305_finish(tag, acc, s);
}
```
Proof strategy (simplified)

Security spec

Protocol security proofs

Protocol specs

Implementation

Crypto assumptions

Secure authenticated channel

Definition 3.4.1 Let $F: K \times D \rightarrow R$ be a family of functions, and let $A$ be an algorithm that takes an oracle and returns a bit. We consider two games as described in Fig. 3.1. The $\text{adv}_A$ of $A$ is defined as

$$\text{Adv}_A^S(A) = \Pr[\text{Real}\rightarrow] - \Pr[\text{Rand}\rightarrow]$$

Network Working Group F. Rescorla
Internet-Draft RFE, Inc.
Obsoletes: 5226, 5245, 5256, 5266, 5277
1st approved
Updates: 5264
Intended standard: Standards Track
Expires: January 6, 2016

draft-ietf-tls-tls13-07

Abstract

This document specifies Version 1.3 of the Transport Layer Security (TLS) protocol. The TLS protocol provides communications security over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference documents.

AES is a pseudo-random function
What do we verify?

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
Everest in Action, so far

Production deployments of Everest Verified Cryptography

- **Windows**
  - WinQUIC: Delivered Everest TLS 1.3 and crypto stack to Windows Networking, in the latest Windows developer previews

- **Mozilla NSS**
  - Mozilla NSS runs Everest verified crypto for several core algorithms

- **Linux**
  - Everest verified crypto in the Linux kernel via WireGuard secure VPN
So what is this F* thing anyway?

A programming language
A proof assistant
A program verification tool
# Two camps of program verification tools

<table>
<thead>
<tr>
<th>Interactive proof assistants</th>
<th>Semi-automated verifiers of imperative programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coq, Isabelle, Agda, Lean, PVS, ...</td>
<td>air Dafny, FramaC, Why3, Verve, IronClad, miTLS, Vale</td>
</tr>
<tr>
<td>CompCert, seL4, Bedrock, 4 colors</td>
<td>gap</td>
</tr>
</tbody>
</table>

- **In the left corner**: Very expressive dependently-typed logics, but only purely functional programming.
- **In the right**: Effectful programming, SMT-based automation, but only first-order logic.
F*: Bridging the gap

- Functional programming language with effects
  - Like OCaml, Haskell, F#, ...
  - Compiles to OCaml or F#
  - A subset of F* compiled to C (with manual control over memory management)

- With an expressive core dependent type theory
  - Like Coq, Agda, Lean, ...

- Semi-automated verification using SMT
  - Like Dafny, Vcc, Liquid Haskell, ...

- In-language extensibility and proof automation using metaprograms
A first taste

- Write ML-like code

```ml
let rec factorial n =
  if n = 0 then 1
  else n * factorial (n - 1)
```

- Give it a specification, claiming that `factorial` is a total function from non-negative to positive integers.

```plaintext
val factorial : int \(\geq 0\) \rightarrow\) \(\text{Tot (int \(\geq 1\)}}
```

- Ask F* to check it

```
fstar factorial.fst
Verified module: Factorial
All verification conditions discharged successfully
```
F* builds a typing derivation of the form:

\[ \Gamma_{\text{prelude}} \vdash \text{let factorial } n = e : t \leftarrow \phi \]

- In a context \( \Gamma_{\text{prelude}} \) including definitions of F* primitives
- The program \( \text{let factorial } n = e \) has type \( t \), given the validity of a logical formula \( \phi \)
- \( \phi \) is passed to Z3 (an automated theorem prover/SMT solver) to check for validity
- If the check succeeds, then, from the metatheory of F*, the program is safe at type \( t \)
Beyond Pure Code

Effects

• Programmers model effects with monads
  • \texttt{st a = s \rightarrow a * s}

• \textit{F*} derives a WP calculus for use with that monad
  • Also known as \textit{Dijkstra} monads
  • \texttt{st\_post a = a * s \rightarrow prop}
  • \texttt{st\_pre = s \rightarrow prop}
  • \texttt{wp\_st a = st\_post a \rightarrow st\_pre}

• And a computation type for effectful terms
  • Computations indexed by their own WPs: \texttt{STATE a (wp : wp\_st a)}
Effectful programs with Hoare-style Specifications

```
let invert (r:ref int)
  : STExn unit
  (requires fun h0 -> True)
  (ensures fun h0 v h1 -> no_exn v ==> h0.[r] <> 0 \&\& h1.[r] = 1 / h0.[r])
  =
  let x = !r in
  if x = 0 then
    raise Division_by_zero
  else r := 1 / x
```
Effectful programs with Hoare-style Specifications

let invert (r: ref int)
  : STExn unit
  (requires fun h0 -> h0.[r] <> 0)
  (ensures fun h0 v h1 -> no_exn v \ h1.[r] = 1 / h0.[r])
=
  let x = !r in
  if x = 0 then
    raise Division_by_zero
  else r := 1 / x
Exploiting Expressiveness & Extensibility

Low*: A subset of F* that compiles to C

• Embed within F* a CompCert C-like memory model
  • Low*: An effect for stateful programs manipulating a C-like memory model
  • Programs in the Low* effect are extracted to C by KreMLin, F*‘s C backend

• Separate memory region for heaps and stacks
  • A region is a heterogeneous partial map
  • \texttt{region = addr -> option (a:Type & a)}

• With libraries modeling mutable arrays, pointers, structs, unions
And to support compilation to C, in nearly 1-1 correspondence, for auditability of our generated code

Designed to allow manipulating a C-like view of memory

```ocaml
let chacha20
  (len: uint32 {len ≤ blocklen}) (output: bytes {len = output.length})
  (key: keyBytes) (nonce: nonceBytes {disjoint [output; key; nonce]}) (counter: uint32):
  Stack unit (requires (λ m0 → output ∈ m0 ∧ key ∈ m0 ∧ nonce ∈ m0))
  (ensures (λ m0 _m1 → modifies output m0 m1)) = Erased specification
push_frame ();
let state = Buffer.create 0ul 32ul in
let block = Buffer.sub state 16ul 16ul in
chacha20_init block key nonce counter;
chacha20_update output state len;
pop_frame ();
```
let encrypt (i:id) (p:writer) (l:bytes) (l:plainLen) (p:plain 1 l)
: ST (cipher 1 l)
(requires (λ h0 ->
  l ≤ max_TLSPlain_text_fragment_length ∧
  sel h0 (ctr e.counter) < max_ctr)
(ensures (λ h0 c h1 ->
  modifies [e.log_region] h0 h1 ∧
  h1 \$HS.contains\$ (ctr e.counter) ∧
  sel h1 (ctr e.counter) == sel h0 (ctr e.counter) + 1 ∧
  (authId i =>
  let log = i log e.log in
  let ent = Entry 1 c p in
  let n = Seq.length (sel h0 log) in
  h1 \$HS.contains\$ log ∧
  witnessed (at_least n ent log) ∧
  sel h1 log == snoc (sel h0 log) ent)))) =

let h0 = get() in
let ctr = ctr e.counter in
HST.recall ctr; //lemma
let text = if safeId i then create_ 1 0 z else repr i l p in
let n = !ctr in
lemma_repr_bytes_values n; //lemma
let nb = bytes_of_int (AEAD.noncelen i) n in
let iv = AEAD.create_nonce e.aead nb in
lemma_repr_bytes_values (length text); //lemma
let c = AEAD.encrypt e.aead iv ad text in
if authId i then
  begin
  let ilog = i log e.log in
  HST.recall ilog; //lemma
  let ictr: ideal_ctr e.region i ilog = e.counter in
  testify_seq ictr;
  write_at_end ilog (Entry 1 c p);
  HST.recall ictr; //lemma
  increment_seq ictr;
  HST.recall ictr //lemma
  end
else
  ctr := n + 1;
  c
But SMT-based proofs can go awry

- E.g., when using theories like non-linear arithmetic

```ocaml
let lemma_carry_limb_unrolled (a0 a1 a2:nat) : Lemma
    (requires T)
    (ensures (a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44))
              = a0 + p44 * a1 + p88 * a2)) =

let open FStar.Math.Lemmas in
let z = a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44)) in
distributivity_add_right p88 a2 ((a1 + a0 / p44) / p44); (* argh! *)
pow2_plus 44 44;
lemma_div_mod (a1 + a0 / p44) p44;
distributivity_add_right p44 ((a1 + a0 / p44) % p44) (p44 * ((a1 + a0 / p44) / p44)); (* argh! *)
assert (p44 * ((a1 + a0 / p44) % p44) + p88 * ((a1 + a0 / p44) / p44) = p44 * (a1 + a0 / p44));
distributivity_add_right p44 a1 (a0 / p44); (* argh! *)
lemma_div_mod a0 p44
```

Forced to write very detailed proof terms when SMT fails
And can be at a low level of abstraction

• Lots of boilerplate to define parsers/formatters and prove them mutually inverse
Domain-specific languages, ad hoc proof automation, extensibility

Allow programmers to:

- Customize how programs are typechecked, producing domain-specific VCs
- Metaprogram programs and their proofs
- Define their own equivalence preserving program transformations
- ...
Domain-specific languages, ad hoc proof automation, extensibility

By treating the F* object language as its own metalanguage

Building on *elaborator reflection* a great idea by

- David Christiansen and Edwin Brady (Idris)
- Leonardo de Moura (Lean)
A passive compiler pipeline

- Program.fst
- Parsing & Desugaring
- Typechecker
- Extraction aka Code generation
- Higher-order Unification
- Normalizer
- SMT Encoding
- Program.ml
Scripting components with a metaprogram

Program.fst

Parsing & Desugaring → Typechecker → Extraction aka Code generation

Higher-order Unification → Normalizer → SMT Encoding

MetaProgram.fst

Program.ml
Scripting a language implementation from within the language

Provide a API to compiler internals for F* (meta)programs to reflect and/or construct
- Syntax of terms
- Typechecking environment
- Typing derivations
- ...

F* compiler runs F* metaprograms to build and typecheck other F* programs
From F* to Meta-F*,
In three easy steps

1. Metaprogramming as a computational effect of *proof-state transformers*

2. Primitive operations to manipulate proof-states, using trusted compiler primitives

3. Reflecting on syntax: quotation and unquotation
Proof-state: A collection of typed holes

A hole is a missing program fragment in a context

\[ \Gamma \vdash \alpha_0 : \tau \]

(aka a *goal*)

The proof-state is a collection of pending holes

\[ \{ \Gamma_i \vdash \alpha_i : \tau_i \} \]

+ some internal persistent state

(e.g., unionfind graph of the unifier)
Metaprograms are proofstate transformers

```haskell
let meta a = proofstate → Dv (either (a × proofstate) error)
```

- Uses an existing F* effect for non-termination: \( Dv \)
- The type of the state is an abstract type: proofstate
- error is the type of exceptions

State + Exception + Non-termination monad
Step 2

**Primitive operations on** **proofstate**

• Every typing rule (read backwards) provided as a proofstate primitive

\[
\frac{\Gamma, x:t_1 \vdash ?_1 : t_2}{\Gamma \vdash ?_0 : (x:t_1 \rightarrow t_2)} \quad \text{[TC-Abs]} \quad \text{where} \quad ?_0 := \lambda x : t_1. ?_1
\]

**intro**: unit -> **Meta** binder

```haskell
intro () (\Gamma \vdash ?_o : (x : t_1 \rightarrow t_2) :: rest) =
  Inl (x, (\Gamma, x:t_1 \vdash ?_1 : t_2 :: rest))

where fresh x and ?_o := fun (x : t_1) -> ?_1
```

intro () _ = raise (Failure “Goal is not an arrow”)
Step 3

Reflecting on syntax

- Locally nameless abstract syntax of F* terms provided as a datatype to metaprograms

- Quotation: Builds the syntax of a term `$\lambda 0 + 1 : \text{term}$`

- Unquotation
  - Typechecks syntax at a given type

```ocaml
val unquote : a : Type -> x : term -> Meta a
```
Putting it together

Entrypoint: Decorate an implicit with a metaprogram

```coq
let id : (a:Type) -> a -> a = 
  _ by (                        [\_ \_ \_ : (a:Type) \rightarrow a \rightarrow a]
    let a = intro () in        [a:Type \_ \_ : a \rightarrow a]
    let x = intro () in        [a:Type, x:a \_ \_ : a]
    hyp x                      []
  )
```
And can be at a low level of abstraction

- Lots of boilerplate to define parsers/formatters and prove them mutually inverse

Remember this?
Metaprogramming mutually inverse parsers and formatters

```ocaml
let parser (t:Type) = bytes → option t
let serializer (#t:Type) (p:parser t) =
  s : (x:t → b:bytes) \{
    (∀ x. p (s x) = Some x) ∧
    (∀ b. match p b with
      | Some x → s x = b
      | _ → T)}
```

Where, the types capture that parser/serializer are mutual inverses
Putting it together

Entrypoint: Decorate an assertion with a metaprogram

let f (x:nat) =
  if x > 1 then
    assert (x * x > x)
    by tac;
  ...

\( x: \text{nat}, h: (x > 1) \vdash _\_ : (x * x > x) \)

- Assertions produce a goal in a context including control flow hypotheses
- Tackled by the metaprogram \texttt{tac}
SMT: Just one of F*’s tactic primitives

val smt: unit -> Meta unit

let f (x:nat) =
  if x > 1 then
    assert (x * x > x)
    by (smt());
  ...

\[ x: \text{nat}, h: (x > 1) \vdash \_ : (x * x > x) \]

- Assertions produce a goal in a context including control flow hypotheses
- Tackled by the metaprogram tac
But SMT-based proofs can go awry

- E.g., when using theories like non-linear arithmetic

```coq
let lemma_carry_limb_unrolled (a0 a1 a2:nat) : Lemma
  (requires T)
  (ensures (a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44))
            = a0 + p44 * a1 + p88 * a2)) =

let open FStar.Math.Lemmas in
let z = a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44)) in
distributivity_add_right p88 a2 ((a1 + a0 / p44) / p44); (* argh! *)
pow2_plus 44 44;
lemma_div_mod (a1 + a0 / p44) p44;
distributivity_add_right p44 ((a1 + a0 / p44) % p44) (p44 * ((a1 + a0 / p44) / p44)); (* argh! *)
assert (p44 * ((a1 + a0 / p44) % p44) + p88 * ((a1 + a0 / p44) / p44) = p44 * (a1 + a0 / p44));
distributivity_add_right p44 a1 (a0 / p44); (* argh! *)
lemma_div_mod a0 p44
```

Forced to write very detailed proof terms when SMT fails
SMT + Tactics for more automated, robust proofs

Key lemma in poly1305, a MAC algorithm, doing multiplication in the prime field $2^{130} - 5$

```ocaml
let poly_multiply (n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 h1 h2 hh : int) : Lemma
  (requires p > 0 ∧ r1 ≥ 0 ∧ n > 0 ∧ 4 × (n × n) = p + 5 ∧ r = r1 × n + r0 ∧ h = h2 × (n × n) + h1 × n + h0 ∧ s1 = r1 + (r1 / 4) ∧ r1 % 4 = 0 ∧ d0 = h0 × r0 + h1 × s1 ∧ d1 = d2 = h2 × r0 ∧ hh = d2 × (n × n) ∧
  (ensures (h × r) % p = hh % p) =
let r14 = r1 / 4 in
let h_r_expand = (h2 × (n × n) + h1 × n + h0) in
let hh_expand = (h2 × r0) × (n × n) + (h0 × (5 × r14)) × n + (h0 × r0 + h1 × (5 × r14)) × n in
let b = (h2 × n + h1) × r14 in
modulo_addition_lemma hh_expand p b;
assert (h_r_expand = hh_expand + b × (n × n) by (canon_semiring int_csr; smt()))
```

- A reflective metaprogram to canonicalize terms in commutative semirings
- Simplifies goal into a form that smt can then solve using linear arithmetic only
- Prior manual proof required 41 steps of explicit rewriting lemmas (!)
Language extension with native metaprograms

• By default, metaprograms are interpreted on F*’s normalizer

• But, F* is implemented in F* and all F* programs can be compiled to OCaml
Language extension with native metaprograms

- Typeclass resolution in F* is implemented entirely in "user-land" with a metaprogram
- As a language implementor this is great! Push language feature requests back to users
- But, F* is implemented in F* and all F* programs can be compiled to OCaml
- Metaprograms too: 10x perf
Some takeaways

• Freedom of expression
  • Tools for large-scale, full program verification need arbitrary expressive power

• Proof automation, Expressiveness, Control
  • SMT is great, but not a panacea: Eventually hit a complexity/undecidability wall

• Careful combination of tactics and SMT improve automation relative to either SMT or tactics alone

• Meta-F*: Self-scripting a PL implementation with reflective metaprogramming; tactics are just a special case