

Meta-F*

Language Extensibility, Metaprogramming and Proof automation

Nik Swamy MSR Redmond



https://fstar-lang.github.io/ https://project-everest.github.io/

Carnegie Mellon

University

Research

Microsoft[®]

MSR Redmond

- Barry Bond
- Chris Hawblitzel
- Qunyan Magnus
- Kiran Muthabatulla
- Jonathan Protzenko
- Tahina Ramananandro
- Nikhil Swamy
- Gustavo Varo

• MSR Cambridge

- Antoine Delignat-Lavaud
- Cédric Fournet
- Christoph M. Wintersteiger
- Santiago Zanella-Béguelin
- MSR India
 - Aseem Rastogi

• INRIA Paris

- Danel Ahman
- Kenji Maillard
- Benjamin Beurdouche
- Karthikeyan Bhargavan
- Victor Dumitrescu
- Cătălin Hriţcu
- Marina Polubelova
- CMU (Pittsburgh)
 - Jay Bosamiya
 - Aymeric Fromherz
 - Bryan Parno
- Edinburgh
 - Markulf Kohlweiss

2016 -- present

- Interns, open-source contributors, visitors, alumns
- Guido Martinez
- Zoe Paraskevopoulou
- Yao Li
- Joonwon Choi
- Clément Pit-Claudel
- Nick Giannarakis
- Niklas Grimm
- Anita Gollamudi
- Nadim Kobeissi
- Matteo Maffei
- Asher Manning
- Monal Narasimhamurthy
- Gordon Plotkin
- Perry Wang
- Jean-Karim Zinzindohoue



TLS: Transport Layer Security



- Most widely deployed security?
 ¹/₂ Internet traffic (+40%/year)
- Web, cloud, email, VoIP, 802.1x, VPNs, ...

TLS: Transport Layer Security Goal: A secure channel



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: Transport Layer Security Goal: A secure channel

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, ...

<u>2</u> . Attacks on TLS		_	•	
<u>2.1</u> . SSL Stripping	anto	fail:	anto	fail
2.2. STARTTLS Command Injection Attack (CVE-2011-0411)	noto	fail	anto	fail
<u>2.3</u> . BEAST (CVE-2011-3389)	goto	fail	noto	fail
<u>2.4</u> . Padding Oracle Attacks	goto	110	goto	Fall
<u>2.5</u> . Attacks on RC4	yoco	Tall;	goqo	Tall
<u>2.6</u> . Compression Attacks: CRIME, TIME, and BREACH	goto	tail;	gotio	tai
<u>2.7</u> . Certificate and RSA-Related Attacks	goto	fails	gota	fail
<u>2.8</u> . Theft of RSA Private Keys	goto	fails	goto	fail
<u>2.9</u> . Diffie-Hellman Parameters	aoto	fails	doto	fail
<u>2.10</u> . Renegotiation (CVE-2009-3555)	noto	f 291 9	not o	fail
<u>2.11</u> . Triple Handshake (CVE-2014-1295)	anto		doto	fail
<u>2.12</u> . Virtual Host Confusion	goco	fait,	goco	Call
<u>2.13</u> . Denial of Service	goto	Tail;	goto	Tai
<u>2.14</u> . Implementation Issues	goto	fail;	goto	fai

The Washington Post

Switch

'FREAK' flaw undermines security for Apple and Google users, researchers discover



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

Network Working Group Internet-Draft Obsoletes: 5077, 5246, 5746 (if approved) Updates: 4492 (if approved) Intended status: Standards Track Expires: September 23, 2016 E. Rescorla RTFM, Inc. March 22, 2016

Table of Contents

- 1.1. Conventions and Terminology
- 1.2. Major Differences from TLS 1.2
- 2. Goals
- 3. Goals of This Document
- 4. Presentation Language
- 4.1. Basic Block Size
- 4.2. Miscellaneous 4.3. Vectors

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

The Transport Layer Security (TLS) Protocol Version 1.3

- #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

Verification Tools and Methodology



Verification Tools and Methodology



Proof strategy (simplified)





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 runs Everest verified crypto for several core algorithms



Everest verified crypto in the Linux kernel via WireGuard secure VPN

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





So what is this F* thing anyway?

- A programming language
- A proof assistant
- A program verification tool

Two camps of program verification tools

Interactive proof assistants			Semi-automated verifiers of imperative programs		
Coq,	CompCert,	air	Dafny,	Verve,	
Isabelle,	seL4,		FramaC,	IronClad,	
Agda,	Bedrock,		Why3	miTLS	
Lean, PVS,	4 colors	gap		Vale	

- 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

- 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)

- 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, ...

- 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, ...

- 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

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.

val factorial: n:int{n >= 0} -> Tot (i:int{i >= 1})

• Ask F* to check it

fstar factorial.fst
Verified module: Factorial
All verification conditions discharged successfully

fstar factorial.fst
Verified module: Factorial
All verification conditions discharged successfully

F* builds a typing derivation of the form:

```
\Gamma_{\text{prelude}} \vdash \texttt{let factorial n} \texttt{=} \texttt{e}: t \Leftarrow \phi
```

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

- Programmers model effects with monads
 - st a = s -> a * s

- Programmers model effects with monads
 - st a = s -> a * s
- F* derives a WP calculus for use with that monad

- Programmers model effects with monads
 - st a = s -> a * s
- F* derives a WP calculus for use with that monad
 - Also known as *Dijkstra* monads
 - st_post a = a * s -> prop
 - st_pre = s -> prop
 - wp_st a = st_post a -> st_pre

- Programmers model effects with monads
 - st a = s -> a * s
- F* derives a WP calculus for use with that monad
 - Also known as *Dijkstra* monads
 - st_post a = a * s -> prop
 - st_pre = s -> prop
 - wp_st a = st_post a -> st_pre
- And a computation type for effectful terms
 - Computations indexed by their own WPs: STATE a (wp : wp_st a)

Effectful programs with Hoare-style Specifications

```
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 -> 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
 - region = addr -> option (a:Type & a)
- With libraries modeling mutable arrays, pointers, structs, unions



Designed to allow manipulating a C-like view of memory

Low*

1	let chacha20
2	$(len: uint32{len \leq blocklen}) (output: bytes{len = output.length})$
3	(key: keyBytes) (nonce: nonceBytes{disjoint [output; key; nonce]}) (counter: uint32) :
4	Stack unit (requires ($\lambda m0 \rightarrow output \in m0 \land key \in m0 \land nonce \in m0$))
5	(ensures ($\lambda \mod m1 \rightarrow modifies_1 \text{ output } m0 \mod 1$)) =
6	push_frame ();
7	let state = Buffer.create 0ul 32ul in
8	let block = Buffer.sub state 16ul 16ul in
9	chacha20_init block key nonce counter;
10	chacha20_update output state len;
11	pop_frame ()



Designed to allow manipulating a C-like view of memory





Designed to allow manipulating a C-like view of memory

I ow* COMPILED 1 et chacha20 void chacha20 ($(len: uint32{len \leq blocklen}) (output: bytes{len = output.length})$ uint32_t len, uint8_t *output, (key: keyBytes) (nonce: nonceBytes{disjoint [output; key; nonce]}) (counter: uint32) : uint8_t *key, uint8_t *nonce, uint32_t counter) 3 3 Stack unit (requires ($\lambda \text{ m0} \rightarrow \text{output} \in \text{m0} \land \text{key} \in \text{m0} \land \text{nonce} \in \text{m0}$)) 4 4 5 5 (ensures ($\lambda \mod m1 \rightarrow \text{modifies}_1 \text{ output } m0 \mod 1$)) = push frame (); 6 6 let state = Buffer.create 0ul 32ul in7 → uint32 t state[32] = $\{0\}$; Stack allocation 8 let block = Buffer.sub state 16 μ 16 μ in uint32 t *block = state + 16; 8 9 9 chacha20_init block key nonce counter; chacha20_init(block, key, nonce, counter); 10 chacha20 update output state len; 10 chacha20_update(output, state, len); 11 pop_frame () 11 }



Designed to allow manipulating a C-like view of memory

low* COMPILED 1 et chacha20 void chacha20 ($(len: uint32{len \leq blocklen}) (output: bytes{len = output.length})$ uint32_t len, uint8_t *output, (key: keyBytes) (nonce: nonceBytes{disjoint [output; key; nonce]}) (counter: uint32) : uint8_t *key, uint8_t *nonce, uint32_t counter) 3 3 Stack unit (requires ($\lambda \text{ m0} \rightarrow \text{output} \in \text{m0} \land \text{key} \in \text{m0} \land \text{nonce} \in \text{m0}$)) 4 4 5 5 (ensures ($\lambda \mod m1 \rightarrow \text{modifies}_1 \text{ output } m0 \mod 1$)) = push frame (); 6 6 let state = Buffer.create 0ul 32ul in7 \rightarrow uint32 t state[32] = { 0 }; Stack allocation 8 let block = Buffer.sub state 16 μ 16 μ in \rightarrow uint32 t \ast block = state + 16; Pointer arithmetic 9 chacha20_init block key nonce counter; chacha20_init(block, key, nonce, counter); 9 10 chacha20_update output state len; 10 chacha20_update(output, state, len); 11 pop_frame () 11


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

low* COMPILED 1 et chacha20 void chacha20 ($(len: uint32{len \leq blocklen}) (output: bytes{len = output.length})$ uint32_t len, uint8_t *output, (key: keyBytes) (nonce: nonceBytes{disjoint [output; key; nonce]}) (counter: uint32) : uint8_t *key, uint8_t *nonce, uint32_t counter) 3 3 Stack unit (requires ($\lambda m0 \rightarrow output \in m0 \land key \in m0 \land nonce \in m0$)) Erased 4 4 5 5 (ensures ($\lambda \mod m1 \rightarrow \text{modifies}_1 \text{ output } m0 \mod 1$)) = specification push frame (); 6 6 let state = Buffer.create 0ul 32ul in7 \rightarrow uint32 t state[32] = { 0 }; Stack allocation 8 let block = Buffer.sub state 16 μ 16 μ in \rightarrow uint32 t *block = state + 16; Pointer arithmetic -9 chacha20_init block key nonce counter; 9 chacha20_init(block, key, nonce, counter); 10 chacha20_update output state len; 10 chacha20_update(output, state, len); 11 pop_frame () 11 }

```
let encrypt (#i:id) (e:writer i) (ad:bytes) (l:plainLen) (p:plain i l)
  : ST (cipher i 1)
       (requires (\lambda h0 \rightarrow
         1 \leq \max TLSPlaintext fragment length \wedge
         sel h0 (ctr e.counter) < max ctr))</pre>
        (ensures (\lambda h0 c h1 \rightarrow
         modifies [e.log region] h0 h1 \wedge
         h1 `HS.contains` (ctr e.counter) ∧
         sel h1 (ctr e.counter) == sel h0 (ctr e.counter) + 1 \wedge
         (authId i \Rightarrow
          let log = ilog e.log in
          let ent = Entry l c p in
          let n = Seq.length (sel h0 log) in
          h1 `HS.contains` log ∧
          witnessed (at least n ent log) \wedge
          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 l 0z 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 = ilog e.log in
    HST.recall ilog; //lemma
    let ictr: ideal ctr e.region i ilog = e.counter in
    testify seqn ictr;
    write at end ilog (Entry l c p);
    HST.recall ictr; //lemma
    increment seqn ictr;
    HST.recall ictr //lemma
    end
  else
    ctr \coloneqq n + 1;
  С
```

Dependently typed specs in the style of Hoare Type Theory / Coq

```
Dafny-ish proofs mostly
automated by SMT, with a few
well-chosen user-supplied
lemmas
```

But SMT-based proofs can go awry

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

```
let lemma_carry_limb_unrolled (a0 a1 a2:nat) : Lemma
    (requires \top)
    (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

emacs@NSWAMY-SURFBK		×
File Edit Options Buffers Tools Help		
<pre>let cipherSuiteBytes cs =</pre>	<pre>> let parseCipherSuite b =</pre>	^
match cs with	match b.[0ul], b.[1ul] with	
UnknownCipherSuite b1 b2 → twobytes (b1,b2)	$ $ (0x00z, 0x00z) \rightarrow Correct(NullCipherSuite)	
NullCipherSuite → twobytes (0x00z, 0x00z)		
	(0x13z, 0x01z) → Correct (CipherSuite13 AES128_GCM SHA256)	
CipherSuite13 AES128_GCM SHA256 → twobytes (0x13z, 0x01z)	(0x13z, 0x02z) → Correct (CipherSuite13 AES256_GCM SHA384)	
CipherSuite13 AES256_GCM SHA384 → twobytes (0x13z, 0x02z)	(0x13z, 0x03z) → Correct (CipherSuite13 CHACHA20_POLY1305 SHA256)	
CipherSuite13 CHACHA20_POLY1305 SHA256 → twobytes (0x13z, 0x03z)	(0x13z, 0x04z) → Correct (CipherSuite13 AES128_CCM SHA256)	
CipherSuite13 AES128_CCM SHA256 → twobytes (0x13z, 0x04z)	(0x13z, 0x05z) → Correct (CipherSuite13 AES128_CCM8 SHA256)	
CipherSuite13 AES128_CCM8 SHA256 → twobytes (0x13z, 0x05z)		
	(0x00z, 0x01z) → Correct(CipherSuite Kex_RSA None (MACOnly MD5))	
CipherSuite Kex_RSA None (MACOnly MD5) → twobytes (0x00z, 0x01z)	(0x00z, 0x02z) → Correct(CipherSuite Kex_RSA None (MACOnly SHA1))	
CipherSuite Kex_RSA None (MACOnly SHA1) → twobytes (0x00z, 0x02z)	(0x00z, 0x3Bz) → Correct(CipherSuite Kex_RSA None (MACOnly SHA256))	
CipherSuite Kex_RSA None (MACOnly SHA256) → twobytes (0x00z, 0x3Bz)	(0x00z, 0x04z) → Correct(CipherSuite Kex_RSA None (MtE (Stream RC4_128) MD5))	
CipherSuite Kex_RSA None(MtE (Stream RC4_128) MD5) → twobytes (0x00z, 0x04z)	(0x00z, 0x05z) → Correct(CipherSuite Kex_RSA None (MtE (Stream RC4_128) SHA1))	
CipherSuite Kex_RSA None(MtE (Stream RC4_128) SHA1) → twobytes (0x00z, 0x05z)		
	(0x00z, 0x0Az) → Correct(CipherSuite Kex_RSA None (MtE (Block TDES_EDE_CBC) SHA1))	
CipherSuite Kex_RSA None(MtE (Block TDES_EDE_CBC) SHA1) → twobytes (0x00z, 0x0Az)	<pre> (0x00z, 0x2Fz) → Correct(CipherSuite Kex_RSA None (MtE (Block AES128_CBC) SHA1))</pre>	
CipherSuite Kex_RSA None(MtE (Block AES128_CBC) SHA1) \rightarrow twobytes (0x00z, 0x2Fz)	(0x00z, 0x35z) → Correct(CipherSuite Kex_RSA None (MtE (Block AES256_CBC) SHA1))	
CipherSuite Kex_RSA None(MtE (Block AES256_CBC) SHA1) → twobytes (0x00z, 0x35z)	(0x00z, 0x3Cz) → Correct(CipherSuite Kex_RSA None (MtE (Block AES128_CBC) SHA256))	
CipherSuite Kex_RSA None(MtE (Block AES128_CBC) SHA256) \rightarrow twobytes (0x00z, 0x3Cz)	(0x00z, 0x3Dz) → Correct(CipherSuite Kex_RSA None (MtE (Block AES256_CBC) SHA256))	
CipherSuite Kex_RSA None(MtE (Block AES256_CBC) SHA256) → twobytes (0x00z, 0x3Dz)		
	(**************************************	
(*************************************	(0x00z, 0x0Dz) → Correct(CipherSuite Kex_DH (Some DSA) (MtE (Block TDES_EDE_CBC) SHA1))	
CipherSuite Kex_DH (Some DSA) (MtE (Block TDES_EDE_CBC) SHA1) → twobytes (0x00z, 0x0Dz)	(0x00z, 0x10z) → Correct(CipherSuite Kex_DH (Some RSASIG) (MtE (Block TDES_EDE_CBC) SHA1))	
CipherSuite Kex_DH (Some RSASIG) (MtE (Block TDES_EDE_CBC) SHA1) → twobytes (0x00z, 0x10z)	(0x00z, 0x13z) → Correct(CipherSuite Kex_DHE (Some DSA) (MtE (Block TDES_EDE_CBC) SHA1))	
CipherSuite Kex_DHE (Some DSA) (MtE (Block TDES_EDE_CBC) SHA1) → twobytes (0x00z, 0x13z)	(0x00z, 0x16z) → Correct(CipherSuite Kex_DHE (Some RSASIG) (MtE (Block TDES_EDE_CBC) SHA1))	
CipherSuite Kex_DHE (Some RSASIG) (MtE (Block TDES_EDE_CBC) SHA1) → twobytes (0x00z, 0x16z)		
	(0x00z, 0x30z) → Correct(CipherSuite Kex_DH (Some DSA) (MtE (Block AES128_CBC) SHA1))	
CipherSuite Kex_DH (Some DSA) (MtE (Block AES128_CBC) SHA1) → twobytes (0x00z, 0x30z)	(0x00z, 0x31z) → Correct(CipherSuite Kex_DH (Some RSASIG) (MtE (Block AES128_CBC) SHA1))	
CipherSuite Kex_DH (Some RSASIG) (MtE (Block AES128_CBC) SHA1) → twobytes (0x00z, 0x31z)	↓ (0x00z, 0x32z) → Correct(CipherSuite Kex_DHE (Some DSA) (MtE (Block AES128_CBC) SHA1))	~
1**- n1.fst Ton 118 (Fundamental -2)	1**- n1.fst 52% 139 (Fundamental -2)	
let inverse cinherSuite (x:cinherSuite)	let inverse cinherSuite' (x:lbytes 2)	~
: emma		
(requires (- (UnknownCipherSuite? x)))	(ensures (let y = parseCipherSuite x in	

```
(ensures (let y = cipherSuiteBytes x in
```

```
Some? y \implies parseCipherSuite (Some?.v y) = Correct x))
```

= ...

(ensures (let y = parseCipherSuite x in Correct? y /\ not (UnknownCipherSuite? (Correct?._0 y)) ==> Some? (cipherSuiteBytes (Correct?._0 y)) /\ x == Some?.v (cipherSuiteBytes (Correct?._0 y)))) = 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.ml

A passive compiler pipeline



Scripting components with a metaprogram



Scripting components with a metaprogram



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 $\Gamma \vdash ?_0 : \tau$ (aka a goal)

The proof-state is a collection of pending holes

$$\{\Gamma_i \vdash ?_i : \tau_i\}$$

+ some internal persistent state (e.g., unionfind graph of the unifier)

Metaprograms are proofstate transformers

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

Metaprogramming as a user-defined effect

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

```
(* Monad *)

let return a x : meta a = \lambda ps \rightarrow Inl (x, ps)

let bind a b (f:meta a) (g: a \rightarrow meta b) : meta b =

\lambda ps \rightarrow match f ps with

| Inl (x, ps') \rightarrow g x ps'

| Inr err \rightarrow Inr err

(* Actions *)

let get () : meta ps = \lambda ps \rightarrow Inl (ps, ps)

let raise a err : meta a = \lambda ps \rightarrow Inr (err, ps)
```

- Standard definitions of return, bind, get, raise
- Exceptions reset the state

Metaprogramming as a user-defined effect

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

```
(* Monad *)
let return a x : meta a = \lambda ps \rightarrow Inl (x, ps)
let bind a b (f:meta a) (g: a \rightarrow meta b) : meta b =
  \lambda ps \rightarrow match f ps with
           | Inl (x, ps') \rightarrow g x ps'
           | Inr err \rightarrow Inr err
(* Actions *)
let get () : meta ps = \lambda ps \rightarrow Inl (ps, ps)
let raise a err : meta a = \lambda ps \rightarrow Inr (err, ps)
(* Ask F* to derive an effect from meta *)
new_effect { META : a:Type → Effect
                with repr = meta; bind; return; raise; get }
```

Metaprogramming as a user-defined effect

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

```
Conspicuously, no put action
```

The proofstate is a compiler internal data structure.

A metaprogram can read it but cannot modify it arbitrarily

Step 2 Primitive operations on proofstate Every typing rule (read backwards) provided as a proofstate primitive

 $\vdash ?_o: (x:t_1 \rightarrow t_2) :: rest) =$ Inl (x, (Γ , x:t_1 $\vdash ?_1: t_2$:: rest))
where fresh x and ?_0 := fun (x:t_1)->?_1
intro () _ = raise (Failure "Goal is not an arrow")

Step 2 Primitive operations on proofstate Every typing rule (read backwards) provided as a proofstate primit

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

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

$$\vdash ?_o: (x:t_1 \rightarrow t_2) :: rest) =$$
Inl (x, (Γ , x:t_1 $\vdash ?_1: t_2$:: rest))
where fresh x and ?_0 := fun (x:t_1)->?_1
intro () _ = raise (Failure "Goal is not an arrow")

Step 2 Primitive operations on proofstate Every typing rule (read backwards) provided as a proofstate primitive

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

intro : unit -> Meta binder $\vdash ?_o : (x:t_1 \rightarrow t_2) :: rest) =$ Inl (x, (Γ , $x:t_1 \vdash ?_1 : t_2 :: rest$)) where fresh x and $?_0 := fun (x:t_1) ->?_1$ intro () _ = raise (Failure "Goal is not an arrow") Step 2 Primitive operations on proofstate xx and ?0??00?0 := fun (xx: t1tt t11t1) -> ?1??11?1 $xx_{i} (\Gamma, xx: t1 tt t11 t1 \vdash ?1??11?1 : t2tt t22t2 ::: rest))$ $\vdash \Gamma? : x: t_{1} : t_{1} : t_{1} : t_{2} : t_{2} :: rest) =$ $E F = Y_{0} : Y_{0}$

intro : unit -> Meta binder where fresh and ?₀ := fun (x:t₁)->?₁ where fresh x and ?₀ := fun (x:t₁)->?₁ intro () _ = raise (Failure "Goal is not an arrow") Step 2 Primitive operations on proofstate xx and ?0??00?0 := fun (xx: t1ttt11t1) -> ?1??11?1 $xx_i (\Gamma, xx: t1 ttt11 t1 \vdash ?1??11?1 : t2ttt22t2 ::: rest))$ $\vdash \Gamma? : x: t_1 ttt11 t1 \vdash ?1??11?1 : t2ttt22t2) :: rest) =$ $Erery o : (xx: t1ttt11t1 \rightarrow t2ttt22t2) :: rest) =$

intro : unit -> Meta binder intro () _ = raise (Failure "Goal is not an arrow") 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

УF	e term =	
	T_Var	: v:bv → term
	T_BVar	: v:bv → term
	T_FVar	: v:fv → term
	Т_Арр	: hd:term → a:argv → term
	T_Abs	: bv :binder \rightarrow body:term \rightarrow term
	T_Arrow	: bv :binder \rightarrow c:comp \rightarrow term
	T_Type	: unit → term
	T_Refine	: bv:bv → ref:term → term
	T_Const	: vconst \rightarrow term
	T_Uvar	: $\mathbb{Z} \rightarrow \text{ctx_uvar_and_subst} \rightarrow \text{term}$
	T_Let	: recf:bool → bv:bv → def:term → body:term → term
	T_Match	: scrutinee:term \rightarrow brs:(list branch) \rightarrow term
	T_Ascribe	edT : e:term \rightarrow t:term \rightarrow tac:option term \rightarrow term
	T_Ascribe	edC : e:term \rightarrow c:comp \rightarrow tac:option term \rightarrow term

Step 3 Reflecting on syntax

- Locally nameless abstract syntax of F* terms provided as a datatype to metaprograms
 type term =
- Quotation: Builds the syntax of a term
 (0 + 1) : term

T_Var : v:bv \rightarrow term T BVar : v:bv \rightarrow term T_FVar : v:fv \rightarrow term T_App : hd:term \rightarrow a:argv \rightarrow term T Abs : by:binder \rightarrow body:term \rightarrow term T Arrow : by:binder \rightarrow c:comp \rightarrow term T Type : unit \rightarrow term T Refine : $bv:bv \rightarrow ref:term \rightarrow term$ T Const : vconst \rightarrow term T Uvar : $\mathbb{Z} \rightarrow \text{ctx}$ uvar and subst $\rightarrow \text{term}$: recf:bool \rightarrow by:by \rightarrow def:term \rightarrow body:term \rightarrow term T Let T Match : scrutinee:term \rightarrow brs:(list branch) \rightarrow term T AscribedT : e:term \rightarrow t:term \rightarrow tac:option term \rightarrow term T AscribedC : e:term \rightarrow c:comp \rightarrow tac:option term \rightarrow term

Step 3 Reflecting on syntax

- Locally nameless abstract syntax of F* terms provided as a datatype to metaprograms
 type term =
- Quotation: Builds the syntax of a term
 (0 + 1) : term
- Unquotation
 - Typechecks syntax at a given type

```
T_Var
              : v:bv \rightarrow term
T BVar
              : v:bv \rightarrow term
T FVar
              : v:fv \rightarrow term
T_App
              : hd:term \rightarrow a:argv \rightarrow term
T Abs
              : by:binder \rightarrow body:term \rightarrow term
T Arrow : by:binder \rightarrow c:comp \rightarrow term
           : unit → term
T Type
T Refine : bv:bv \rightarrow ref:term \rightarrow term
T Const : vconst \rightarrow term
T Uvar : \mathbb{Z} \rightarrow \text{ctx} uvar and subst \rightarrow \text{term}
              : recf:bool \rightarrow by:by \rightarrow def:term \rightarrow body:term \rightarrow term
T Let
T Match : scrutinee:term \rightarrow brs:(list branch) \rightarrow term
T AscribedT : e:term \rightarrow t:term \rightarrow tac:option term \rightarrow term
T AscribedC : e:term \rightarrow c:comp \rightarrow tac:option term \rightarrow term
```

```
val unquote: a:Type -> x:term -> Meta a
```

Putting it together

Putting it together

 $[\cdot \vdash _:(aa:_{Type}) \rightarrow aa \rightarrow aa]$

Putting it together () in [aa:Type ⊢_:aa →aa]

 $[- [aa:Type) \rightarrow aa \rightarrow aa]$

let id : (a:Type) -> a -> a =
 let a = intro a:Type
$$\vdash _: a \rightarrow a$$
]
intro () in [a:Type, x: a $\vdash _: a$]
 hyp x
 []
)

Putting it together

- () in [aa:Type, xx:aa ⊢_:aa]
- () **in** [aa:Type \vdash :aa \rightarrow aa]

$$[\cdot \vdash _ :(aa:_{Type}) \rightarrow aa \rightarrow aa]$$

Putting it together

() **in** [aa:Type, $xx:aa \vdash aa$]

() **In** [aa:Type \vdash :aa \rightarrow aa]

[]

$$[\cdot \vdash _ :(aa:_{Type}) \rightarrow aa \rightarrow aa]$$

And can be at a low level of abstraction

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

Semacs@NSWAMY-SURFBK	
File Edit Options Buffers Tools Help	
<pre>let cipherSuiteBytes cs =</pre>	<pre>> let parseCipherSuite b =</pre>
match cs with	match b.[0ul], b.[1ul] with
UnknownCipherSuite b1 b2 → twobytes (b1,b2)	$(0x00z, 0x00z) \rightarrow \text{Correct(NullCipherSuite)}$
NullCipherSuite → twobytes (0x00z, 0x00	
	(0x13z, 0x01z) → Correct (CipherSuite13 AES128_GCM SHA256) Kenender
CipherSuite13 AES128_GCM SHA256 → twobytes (0x13z, 0x01	z) (0x13z, 0x02z) → Correct (CipherSuite13 AES256_GCM SHA384)
CipherSuite13 AES256_GCM SHA384 → twobytes (0x13z, 0x02	z) ($0x13z$, $0x03z$) → Correct (CipherSuite13 CHACHA20_POLY1305 SHA256) 4
CipherSuite13 CHACHA20_POLY1305 SHA256 → twobytes (0x13z, 0x03	z) (0x13z, 0x04z) → Correct (CipherSuite13 AES128_CCM SHA256)
CipherSuite13 AES128_CCM SHA256 → twobytes (0x13z, 0x04	z) (0x13z, 0x05z) → Correct (CipherSuite13 AES128_CCM8 SHA256)
CipherSuite13 AES128_CCM8 SHA256 → twobytes (0x13z, 0x05	z)
	(0x00z, 0x01z) → Correct(CipherSuite Kex_RSA None (MACOnly MD5))
CipherSuite Kex_RSA None (MACOnly MD5) → twobytes (0x00z, 0x01	z) (0x00z, 0x02z) → Correct(CipherSuite Kex_RSA None (MACOnly SHA1))
CipherSuite Kex_RSA None (MACOnly SHA1) → twobytes (0x00z, 0x02	z) (0x00z, 0x3Bz) → Correct(CipherSuite Kex_RSA None (MACOnly SHA256))
CipherSuite Kex_RSA None (MACOnly SHA256) → twobytes (0x00z, 0x3B	z) (0x00z, 0x04z) → Correct(CipherSuite Kex_RSA None (MtE (Stream RC4_128) MD5))
CipherSuite Kex_RSA None(MtE (Stream RC4_128) MD5) → twobytes (0x00z, 0x04	z) (0x00z, 0x05z) → Correct(CipherSuite Kex_RSA None (MtE (Stream RC4_128) SHA1))
CipherSuite Kex_RSA None(MtE (Stream RC4_128) SHA1) → twobytes (0x00z, 0x05	
	☐ [(0X00z, 0X0Az) → Correct(CipherSuite Kex_RSA None (Mtt (Block IDES_EDE_CBC) SHAI))
CipherSuite Kew_RSA None(MtE (Block IDES_EDE_CBC) SHAI) → twobytes (0x00z, 0x0A	z) (0X00z, 0X2Fz) → Correct(CipherSuite Kex_RSA None (Mtt (Block AESI28_CBC) SHAI))
CipherSuite Kex_RSA None(MtE (Block AES128_CBC) SHA1) → twobytes (0x00z, 0x2Fz) (0x00z, 0x35z) → Correct(CipherSuite Kex_RSA None (Mtt (Block AES256_CBC) SHAI))
CipherSuite Kex_RSA None(MtE (Block AES256_CBC) SHAI) → twobytes (0x00z, 0x35z) ($(00002, 00302)$) \rightarrow Correct(CipherSuite Kex_RSA None (Mtt (Block AES128_CBC) SHA256))
CipherSuite Kex_RSA None(MTE (Block AES128_CBC) SHA256) → twobytes (0x00z, 0x3cz) (0X00Z, 0X3DZ) → Correct(LipnerSuite Kex_KSA None (MTE (Block AES256_CBC) SHA256))
CipnerSuite Kex_KSA None(MtE (Block AES256_LBC) SHA256) → twodytes (0x002, 0x302)
/**********	(Average Average)) , connect(GinbarSuite Key DH (Come DSA) (Mtt (Diack TDE) (DA1))
(CiphonSuito Koy DH (Some DSA) (MtE (Plack TDES EDE (PC) SHA1)) turbutos (200	$(0.0002, 0.0002) \rightarrow \text{Connect}((1))$ (Some DSA) (inte (block DDS_EDE_COC) Shall)
[CipherSuite Key_DH (Some DSA) (ME (Block TDES_EDE_DEC) SHAI) - twoDytes (0x00	$2, 0x002$ $(0x002, 0x002, 0x002) \neq connect(CliphenSuite Key_DM (Some KSASIG) (Mite (Block HDES_EDE_CDE) Shall)$
[CipherSuite Kex_DH (Some RSA310) (ME (Block TDES_EDE_DEC) SHAI) → twodytes (0x00	$2, 0x102$ $(0x002, 0x132) \rightarrow Connect(CliphenSuite Key_DHE (Some DSA) (MEC (Block TDES_EDE_EDE) STALL)$
[CipherSuite Key_DHE (Some DSA) (ME (Block TDES_LDE_DEC) SHAI) - twobytes (0x00	2, 0x12,) [(0x002, 0x102) + correct(cipilersuite kex_bit (some ksx510) (htt (biotk fbt5_tbt_cbc) shAr))
Cipitel Suite Kex_Dire (Solie KSASIG) (Hite (BIOCK TDES_EDE_EDE) STAT) + CWODYLES (0.000	(QVQQ7 QVQQ7) > Correct(CinherSuite Key DH (Some DSA) (MtE (Rlock AES128 CBC) SHA1))
CipherSuite Key DH (Some DSA) (MtF (Rlock AFS128 (RC) SHA1) → twohytes (0x007	$(0,0007,0,0317) \rightarrow Connect(ChipherSuite Key DH (Some BSAS(G) (ME (Block AESI28 (BC) SHA1)))$
CipherSuite Key DH (Some RSASTG) (ME (Block AES128 (BC) SHA1) \rightarrow twobytes (0x002	(0,0,0,1) $(0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0$
1**- p1.fst Top L18 (Fundamental -2)	1**- p1.fst 52% L139 (Fundamental -2)
	^
<pre>let inverse_cipherSuite (x:cipherSuite)</pre>	<pre>let inverse_cipherSuite' (x:lbytes 2)</pre>
: Lemma	: Lemma
(requires (¬ (UnknownCipherSuite? x)))	(ensures (let y = parseCipherSuite x in

(requires (¬ (UnknownCipherSuite? x)))

```
(ensures (let y = cipherSuiteBytes x in
```

Some? $y \implies parseCipherSuite (Some?.v y) = Correct x))$

. . .

Correct? y /\ not (UnknownCipherSuite? (Correct?._0 y)) ==>

Some? (cipherSuiteBytes (Correct?._0 y)) /\ x == Some?.v (cipherSuiteBytes (Correct?._0 y)))) = Metaprogramming mutually inverse parsers and formatters

```
type color = | Red | Blue | Green | Yellow
type pallette = nlist 18 (color × u8)
let ps : (p:parser pallette & serializer p) =
    _ by (gen_parser_serializer (`pallette))
```

Metaprogramming mutually inverse parsers and formatters

```
type color = | Red | Blue | Green | Yellow
type pallette = nlist 18 (color × u8)
let ps : (p:parser pallette & serializer p) =
    _____by (gen_parser_serializer (`pallette))
```

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

```
let parser (t:Type) = bytes \rightarrow option t
let serializer (#t:Type) (p:parser t) =
  s : (x:t \rightarrow b:bytes) {
    (\forall x. p (s x) = Some x) \land
    (\forall b. match p b with
        | Some x \rightarrow s x = b
        | _ \rightarrow \top)}
```

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: nat, h: (x > 1) \vdash : (x * x > x)
```

- Assertions produce a goal in a context including control flow hypotheses
- Tackled by the metaprogram 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());
```

...

 $x: nat, h: (x > 1) \vdash : (x * x > x)$

- 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

```
let lemma_carry_limb_unrolled (a0 a1 a2:nat) : Lemma
    (requires \top)
                                                                                                 Remember
    (ensures (a0 % p44 + p44 * ((a1 + a0 / p44) % p44) + p88 * (a2 + ((a1 + a0 / p44) / p44))
                                                                                                 this?
             = 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 $2^{130} - 5$

```
let poly multiply (n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 h1 h2 hh : \mathbb{Z})
   : Lemma
     (requires p > 0 \land r1 \ge 0 \land n > 0 \land 4 \times (n \times n) == p + 5 \land r == r1 \times n + r0 \land
                  h == h2 \times (n \times n) + h1 \times n + h0 \wedge s1 == r1 + (r1 / 4) \wedge r1 \% 4 == 0 \wedge n
                  d0 == h0 × r0 + h1 × s1 \land d1 == h0 × r1 + h1 × r0 + h2 × s1 \land
                  d2 == h2 \times r0 \wedge hh == d2 \times (n \times n) + d1 \times n + d0)
     (ensures (h \times r) \% p == hh \% p) =
let r14 = r1 / 4 in
let h r expand = (h2 \times (n \times n) + h1 \times n + h0) \times ((r14 \times 4) \times n + r0) in
let hh expand = (h2 \times r0) \times (n \times n) + (h0 \times (r14 \times 4) + h1 \times r0 + h2 \times r0)
                     (5 \times r14)) \times n + (h0 \times r0 + h1 \times (5 \times r14)) in
let b = (h2 \times n + h1) \times r14 in
modulo addition lemma hh expand p b;
assert (h r expand == hh expand + b × (n × n × 4 + (\$-\$5))
     by (canon semiring int csr;
           smt())
```

SMT + Tactics for more automated, robust proofs

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

```
let poly_multiply (n p r h r0 r1 h0 h1 h2 s1 d0 d1 d2 h1 h2 hh : \mathbb{Z})
  : Lemma
     (requires p > 0 \land r1 \ge 0 \land n > 0 \land 4 \times (n \times n) == p + 5 \land r == r1 \times n + r0 \land
                h == h2 \times (n \times n) + h1 \times n + h0 \wedge s1 == r1 + (r1 / 4) \wedge r1 \% 4 == 0 \wedge
                d0 == h0 \times r0 + h1 \times s1 \wedge d1 =:
                d2 = h2 \times r0 \wedge hh = d2 \times (n = \bullet A reflective metaprogram to canonicalize
    (ensures (h \times r) \% p == hh \% p) =
                                                         terms in commutative semirings
let r14 = r1 / 4 in
let h_r expand = (h2 × (n × n) + h1 × n + h0)
let hh_expand = (h2 \times r0) \times (n \times n) + (h0 \times (n \times n))
                                                         Simplifies goal into a form that smt can
                  (5 \times r14)) \times n + (h0 \times r0 + l)
                                                         then solve using linear arithmetic only
let b = (h2 \times n + h1) \times r14 in
modulo_addition_lemma hh_expand p b;
assert (h_r_expand == hh_expand + b × (n × n
    by (canon_semiring int_csr;
                                                        Prior manual proof required 41 steps of
         smt())
                                                         explicit rewriting lemmas (!)
```

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

 But, F* is implemented in F* and all F* programs can be compiled to OCaml



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

 But, F* is implemented in F* and all F* programs can be compiled to OCaml

• Metaprograms too: 10x perf



• Typeclass resolution in F* is implemented entirely in "user-land" with a metaprogram

 But, F* is implemented in F* and all F* programs can be compiled to OCaml

• Metaprograms too: 10x perf



- 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

and all F* programs can be compiled to OCaml

• Metaprograms too: 10x perf



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

- 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

- 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

- 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

Project Everest Verified Secure Components in the TLS Ecosystem

• SMT, metaprogramming and tactics for scalable, robust program proof

> https://fstar-lang.github.io https://project-everest.github.io/



