Combining tactics, normalization, and SMT solving to verify systems software

Chris Hawblitzel

F*: expressive and automated verification

```
type nat10 = x:int{0 <= x /\ x < 10}
type nat20 = x:int{0 <= x /\ x < 20}

let f (x:nat20) = ...
let g (x:nat10) = f x
```

ask Z3 to prove:

(0 <= x /\ x < 10) ==>
(0 <= x /\ x < 20)

- higher-order logic
- tactics (as of 2017)
- full dependent types (as of 2015)
  - interpreter in type checker (computation on terms)

```
let f (b:bool):(if b then int else bool)
  = if b then 5 else true

let x:int = f true
```

F* interpreter (not Z3):

(if true then int else bool) → int

Using F* to verify systems software

(an F user's perspective)*

- Introductory examples
  - Bytes and words via normalization + SMT
  - Parsers/printers via tactics + SMT
    - EverParse library (USENIX Security 2019)
- Everest project and EverCrypt
  - Example cryptography: SHA, Poly1305
  - Poly1305 math via tactics + SMT
- Assembly language in Vale/F*
  - Efficient verification conditions via normalization + SMT

Demo: bytes and words via normalization + SMT

```
let nat8 = n:nat{n < 0x100}

let rec bytes_to_nat_i (s:seq nat8) (i:nat{i <= length s}) : nat =
  if i = 0 then 0
  else s.[i - 1] + 0x100 * bytes_to_nat_i s (i - 1)

let rec bytes_to_nat (s:seq nat8) : nat =
  bytes_to_nat_i s (length s)

let demo_norm (s:seq nat8) : Lemma
  (requires
    length s == 8 /\
    (forall (i:nat).{:pattern s.[i]} i < 8 ==> s.[i] = 0x12 * i))
  (ensures bytes_to_nat s == 0x00122436485a6c7e)
  =
  norm_spec
    [zeta; iota; primops; delta_only [`%bytes_to_nat_i]]
    (bytes_to_nat_i s 8)
```

Demo: parsers/printers via tactics + SMT

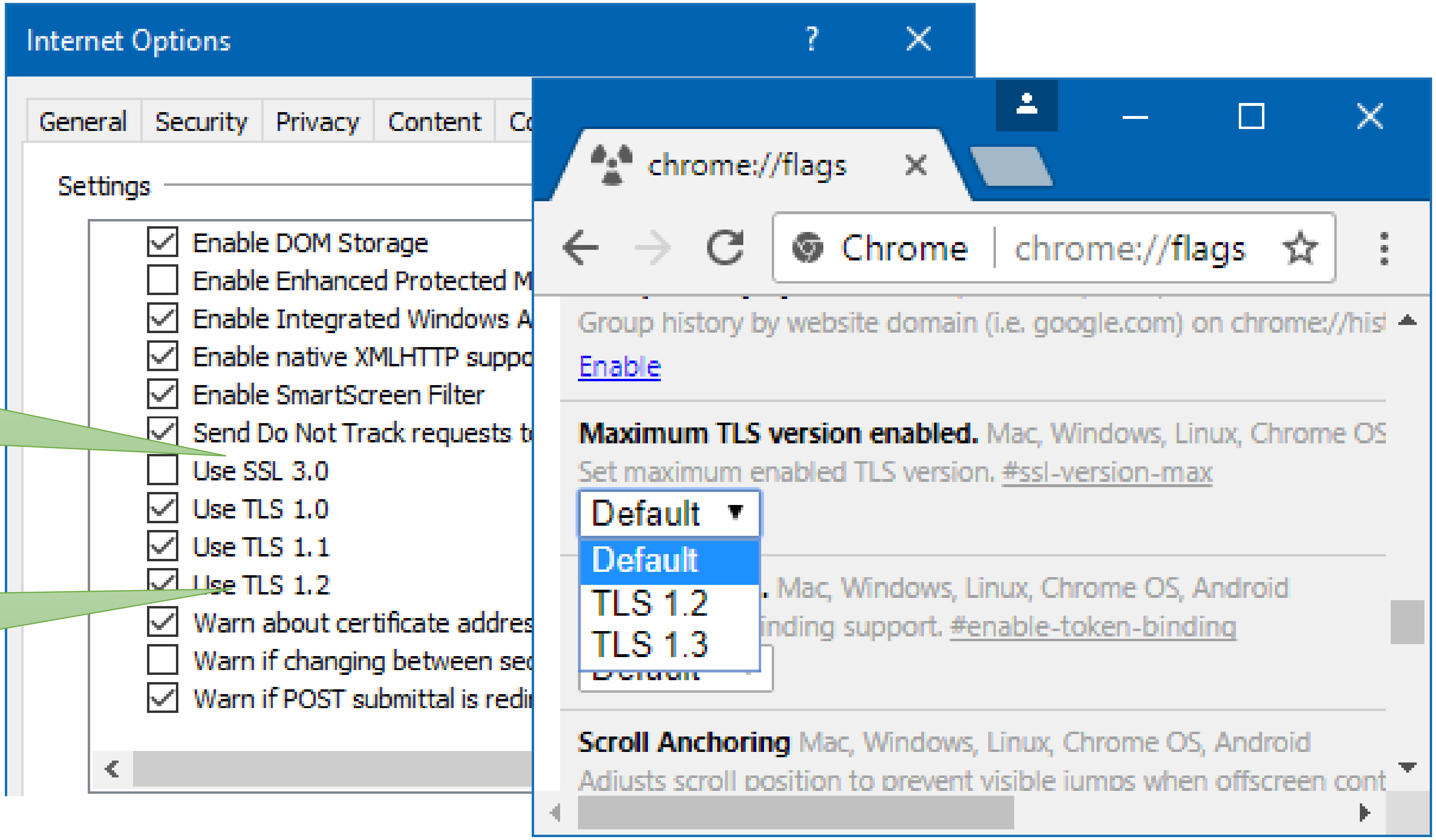
```
type color = | Red | Green | Blue

let make_value (t:term) : Tac unit =
  if term_eq t (`int) then exact (`0) else
  if term_eq t (`bool) then exact (`false) else
  if term_eq t (`color) then exact (`Red) else
  fail "oops"

let i:int = _ by (make_value (`int))
let c:color = _ by (make_value (`color))
```

Demo: parsers/printers via tactics + SMT

```
noeq type print_parse (a:Type) =
| PrintParse :
   print: (a -> int) ->
   parse: (int -> a) ->
   round_trip: (v:a -> Lemma (ensures parse (print v) == v)) ->
   print_parse a

let print_parse_bool : print_parse bool = …

let print_color (v:color) : int = match v with | Red -> 0 | Green -> 1 | Blue -> 2
let parse_color (p:int) : color = match p with | 0 -> Red | 1 -> Green | _ -> Blue
let lemma_color (v:color) : Lemma (ensures parse_color (print_color v) == v) = ()
let print_parse_color : print_parse color = PrintParse print_color parse_color lemma_color

let make_print_parse (t:term) : Tac unit =
  if term_eq t (`bool) then exact (`print_parse_bool) else
  if term_eq t (`color) then exact (`print_parse_color) else
  fail "oops"

let test:print_parse color = _ by (make_print_parse (`color))
```

Secure communication

confidentiality, integrity, authentication



Internet Options

Settings

☑ Enable DOM Storage
☐ Enable Enhanced Protected M
☑ Enable Integrated Windows A
☑ Enable native XMLHTTP suppo
☑ Enable SmartScreen Filter
☑ Send Do Not Track requests t
☐ Use SSL 3.0
☑ Use TLS 1.0
☑ Use TLS 1.1
☑ Use TLS 1.2
☑ Warn about certificate addres
☐ Warn if changing between se
☑ Warn if POST submittal is redi

Group history by website domain (i.e. google.com) on chrome://hist

Enable

Maximum TLS version enabled. Mac, Windows, Linux, Chrome OS

Set maximum enabled TLS version. #ssl-version-max

Default ▾

- Default
- TLS 1.2
- TLS 1.3

. Mac, Windows, Linux, Chrome OS, Android

inding support. #enable-token-binding

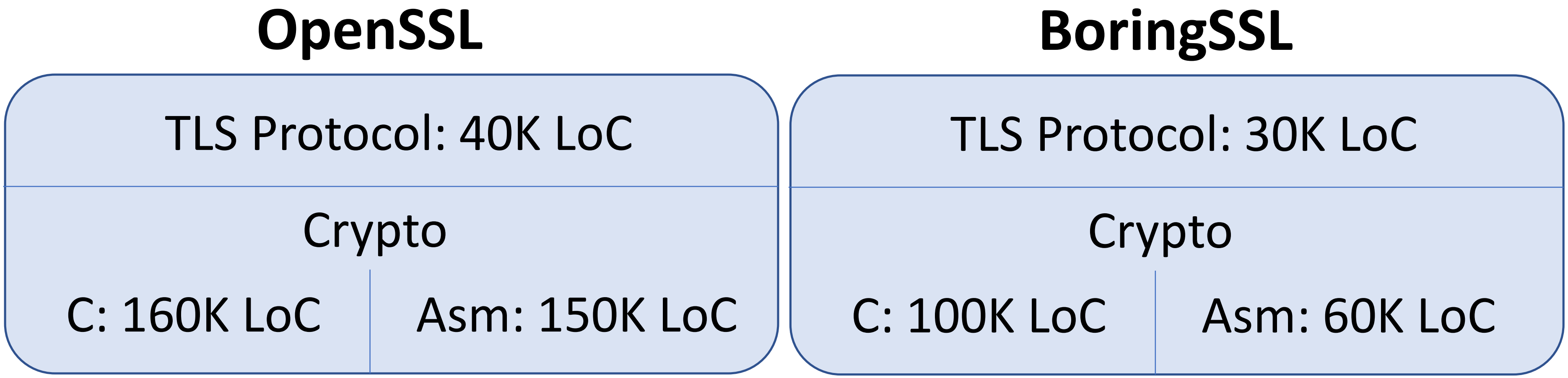
Scroll Anchoring Mac, Windows, Linux, Chrome OS, Android

Adjusts scroll position to prevent visible jumps when offscreen cont

SSL: **S**ecure **S**ockets **L**ayer

TLS: **T**ransport **L**ayer **S**ecurity

TLS standards, some implementations



OpenSSL

TLS Protocol: 40K LoC

Crypto

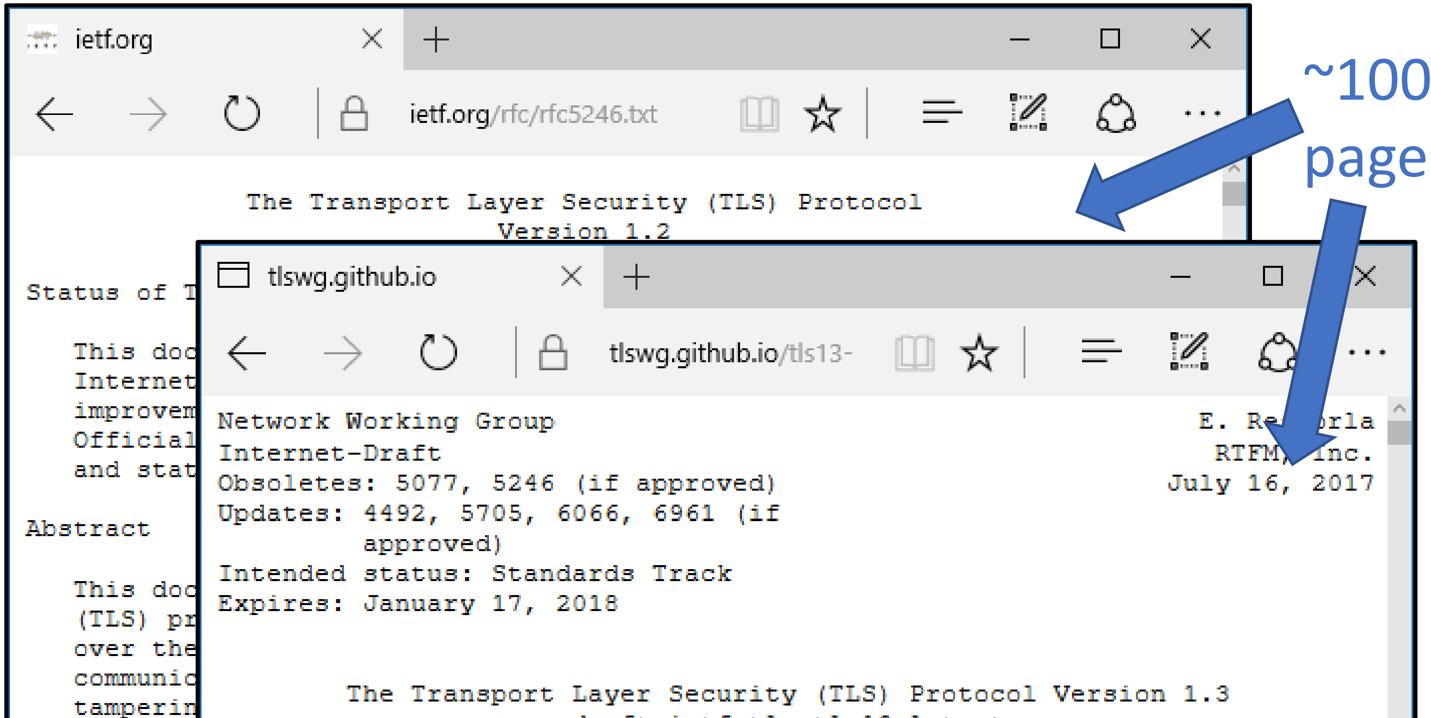
C: 160K LoC | Asm: 150K LoC

BoringSSL

TLS Protocol: 30K LoC

Crypto

C: 100K LoC | Asm: 60K LoC



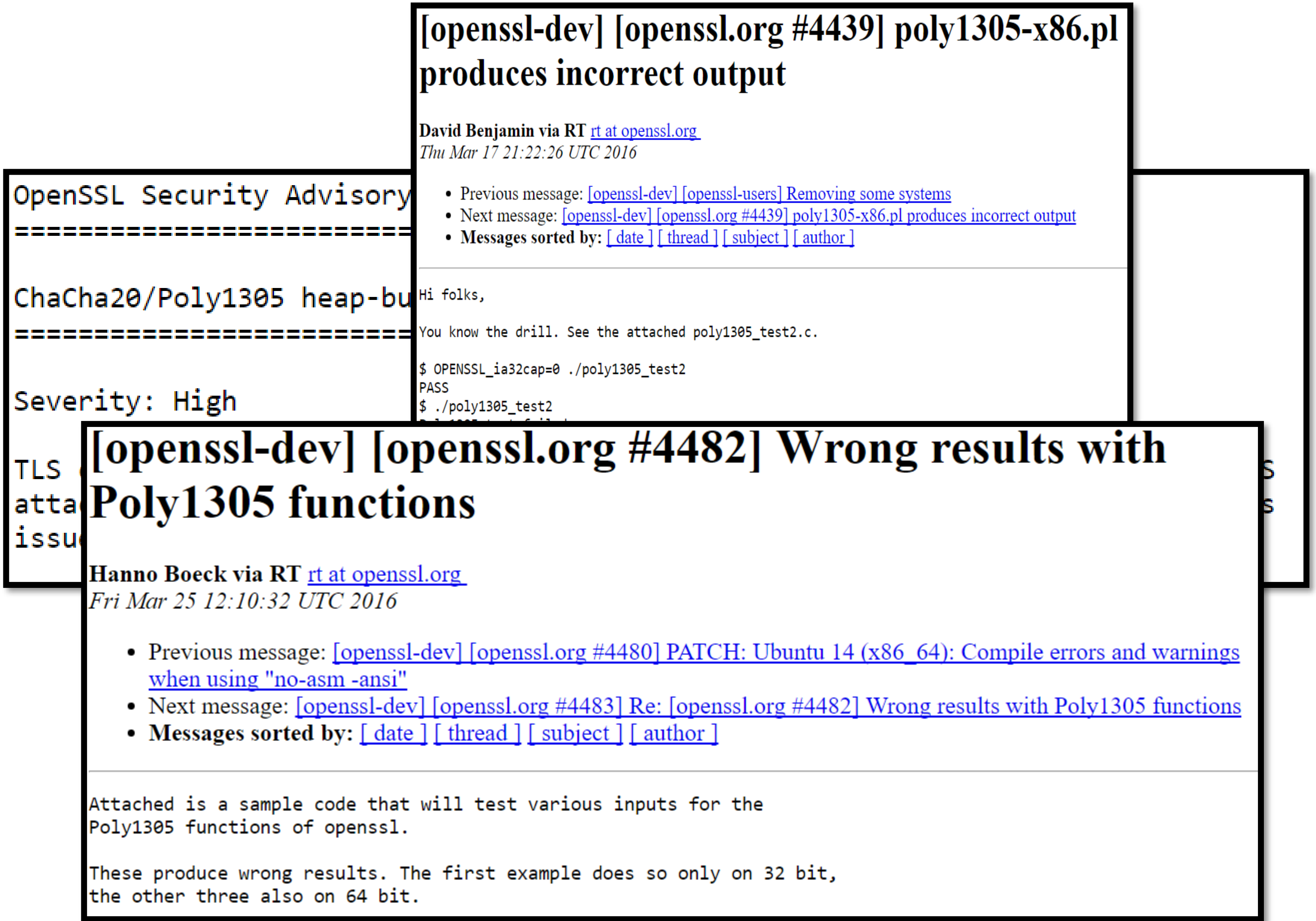
```
          The Transport Layer Security (TLS) Protocol
                          Version 1.2

Status of T

   This doc
   Internet
   improvem
   Official
   and stat

Abstract

   This doc
   (TLS) pr
   over the
   communic
   tamperin
```

```
Network Working Group                                    E. Re    rla
Internet-Draft                                           RTFM, Inc.
Obsoletes: 5077, 5246 (if approved)                   July 16, 2017
Updates: 4492, 5705, 6066, 6961 (if
         approved)
Intended status: Standards Track
Expires: January 17, 2018


        The Transport Layer Security (TLS) Protocol Version 1.3
```

~100 pages

Lines-of-Code measured with SLOCCount

Crypto implementation bugs



```
OpenSSL Security Advisory
==========================

ChaCha20/Poly1305 heap-bu
==========================

Severity: High

TLS
atta
issu
```

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

David Benjamin via RT rt at openssl.org
Thu Mar 17 21:22:26 UTC 2016

- Previous message: [openssl-dev] [openssl-users] Removing some systems
- Next message: [openssl-dev] [openssl.org #4439] poly1305-x86.pl produces incorrect output
- **Messages sorted by:** [date] [thread] [subject] [author]

```
Hi folks,

You know the drill. See the attached poly1305_test2.c.

$ OPENSSL_ia32cap=0 ./poly1305_test2
PASS
$ ./poly1305_test2
```

[openssl-dev] [openssl.org #4482] Wrong results with Poly1305 functions

Hanno Boeck via RT rt at openssl.org
Fri Mar 25 12:10:32 UTC 2016

- Previous message: [openssl-dev] [openssl.org #4480] PATCH: Ubuntu 14 (x86_64): Compile errors and warnings when using "no-asm -ansi"
- Next message: [openssl-dev] [openssl.org #4483] Re: [openssl.org #4482] Wrong results with Poly1305 functions
- **Messages sorted by:** [date] [thread] [subject] [author]

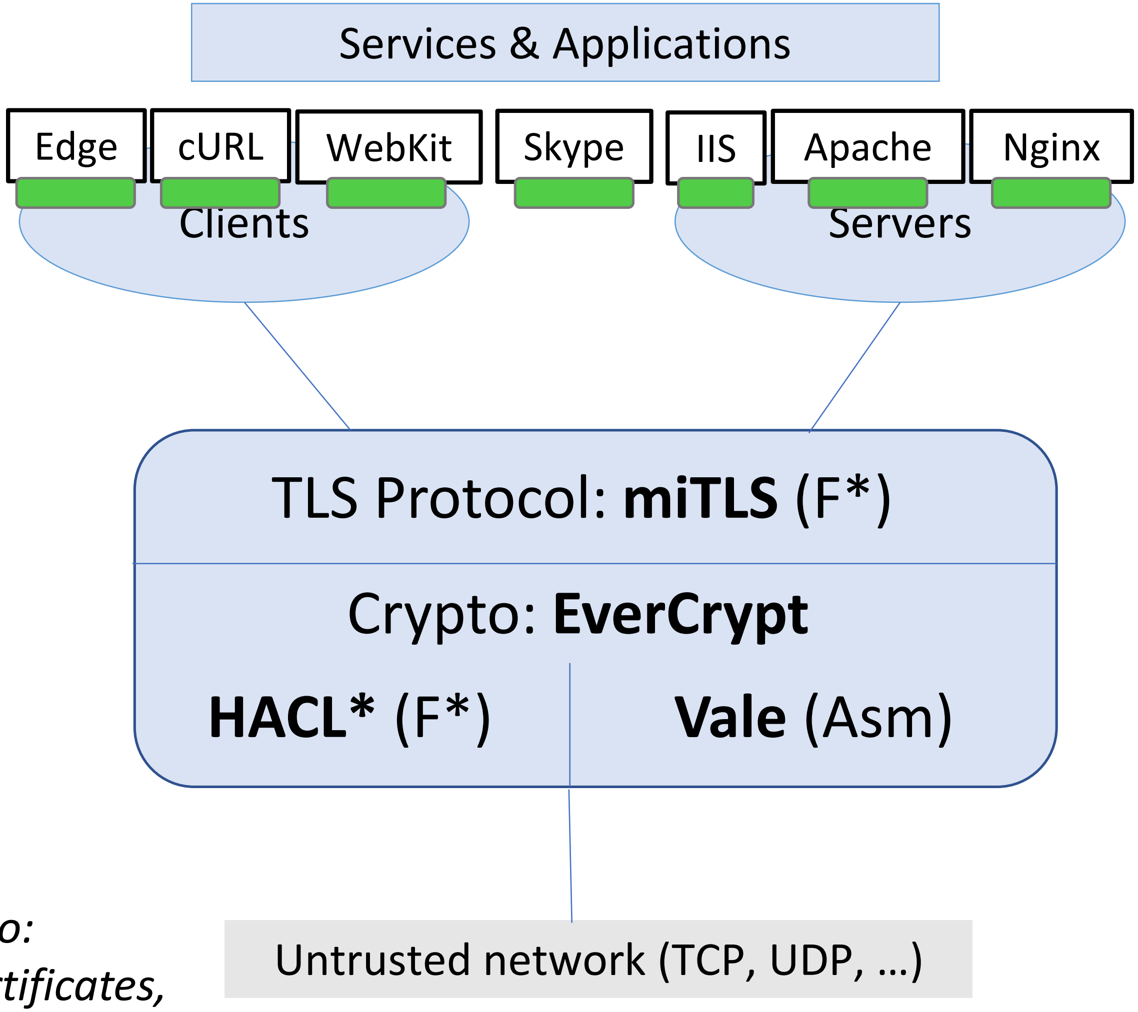
```
Attached is a sample code that will test various inputs for the
Poly1305 functions of openssl.

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

Everest: verified components for the HTTPS ecosystem

Goals:

- Strong verified security
- Trustworthy, usable tools
- Widespread deployment

5-year project (2016-2021)



Services & Applications

Edge | cURL | WebKit | Skype | IIS | Apache | Nginx

Clients

Servers

TLS Protocol: **miTLS** (F*)

Crypto: **EverCrypt**

HACL* (F*)

Vale (Asm)

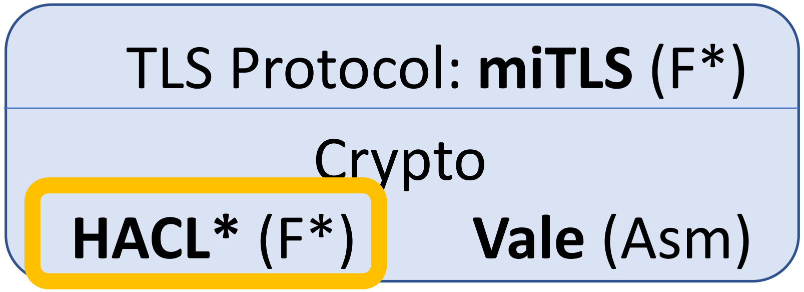
Untrusted network (TCP, UDP, …)

also:
certificates,
properties of HTTPS,
…

Verifying cryptography

- Popular algorithms
  - symmetric (shared key): **AES**, **ChaCha20**, …
  - hashes and MACs: **SHA**, **HMAC**, **Poly1305**, …
    - combined symmetric+MAC (AEAD): **AES-GCM**, …
  - public key and signatures: **RSA**, **Elliptic curve**, …
- Verification goals:
  - safety
  - implementation meets specification
  - avoid side channels

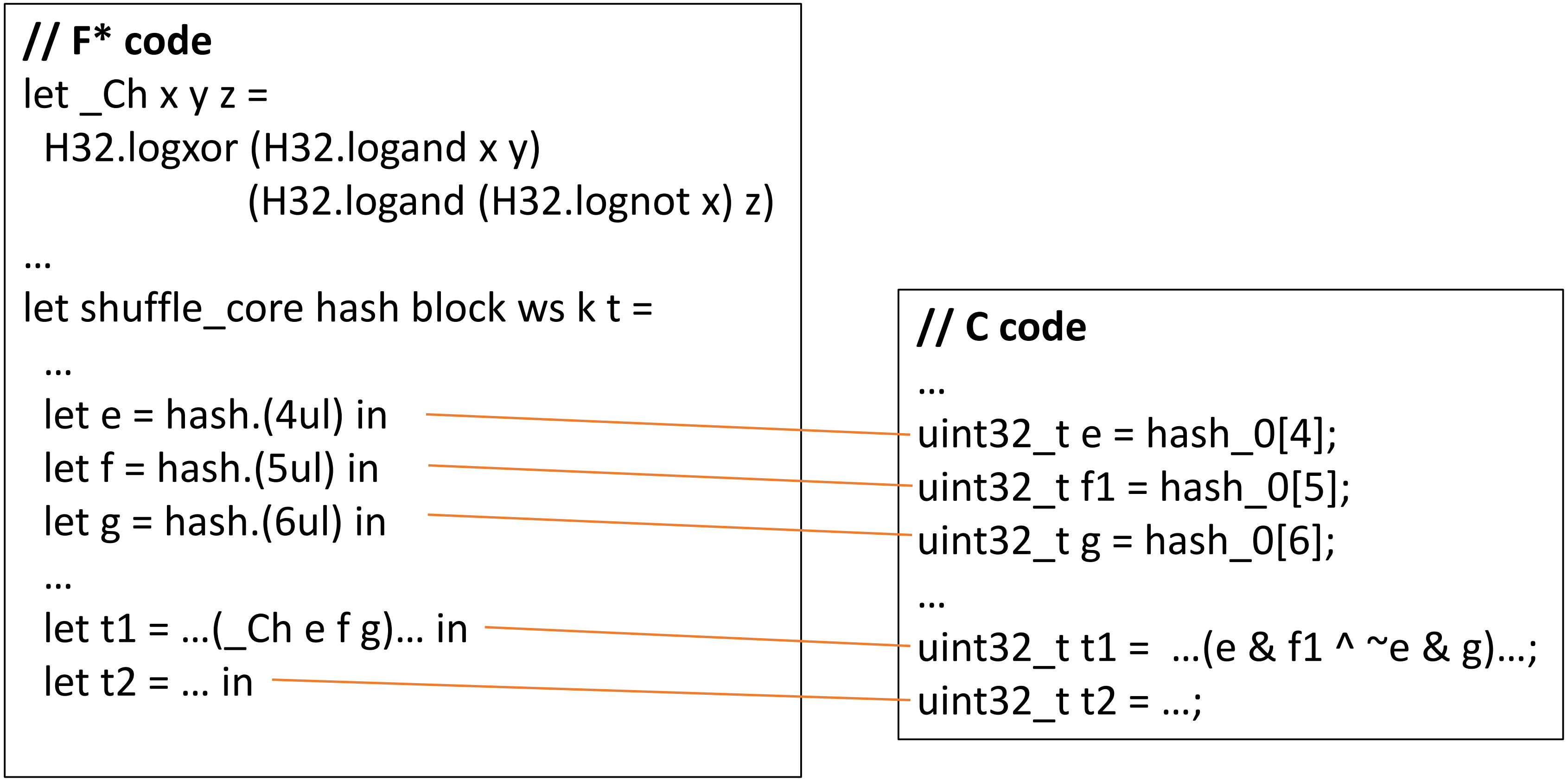
HACL* SHA example


TLS Protocol: **miTLS** (F*)

Crypto

HACL* (F*)

Vale (Asm)


// F* code

```
let _Ch x y z =
  H32.logxor (H32.logand x y)
             (H32.logand (H32.lognot x) z)
…
let shuffle_core hash block ws k t =
  …
  let e = hash.(4ul) in
  let f = hash.(5ul) in
  let g = hash.(6ul) in
  …
  let t1 = …(_Ch e f g)… in
  let t2 = … in
```

// C code

```
…
uint32_t e = hash_0[4];
uint32_t f1 = hash_0[5];
uint32_t g = hash_0[6];
…
uint32_t t1 =  …(e & f1 ^ ~e & g)…;
uint32_t t2 = …;
```

Example algorithm: Poly1305 MAC

```
// pseudocode for poly1305 inner loop

bigint p := $2^{130}$ - 5;
bigint h := 0;
uint128 r := ...derived from key...;

while(...) {
    uint128 data := ...next 16 data bytes...;
    h := h + data;
    h := h * r;
    h := h mod p;
}
```

Example algorithm: Poly1305 MAC

...

h := h mod p; // $p = 2^{130} - 5$

...

$395 = 4 * 10^2 - 5$

$p = 4*(2^{64})^2 - 5$

$\boxed{3}\boxed{01}$ mod 99
= 202 mod 99
= 103 mod 99
= 4 mod 99

= (3+1) mod 99

301 mod 95
= 206 mod 95
= 111 mod 95
= 16 mod 95

= (5*3+1) mod 95

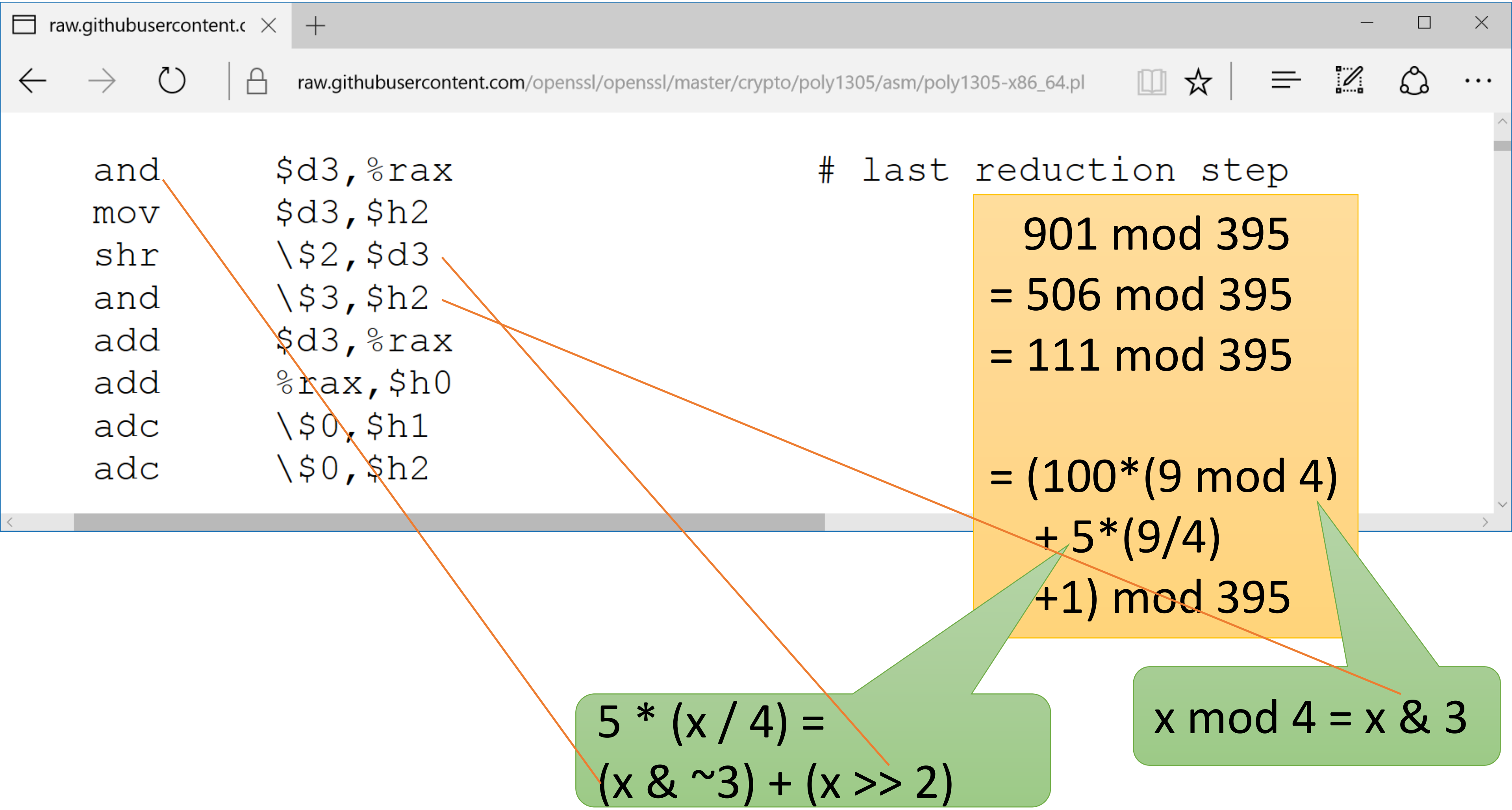
901 mod 395
= 506 mod 395
= 111 mod 395

= (100*(9 mod 4)
+ 5*(9/4)
+1) mod 395

5 * (x / 4) =
(x & ~3) + (x >> 2)

x mod 4 = x & 3

Example algorithm: Poly1305 MAC



raw.githubusercontent.com/openssl/openssl/master/crypto/poly1305/asm/poly1305-x86_64.pl

```
	and	$d3,%rax		# last reduction step
	mov	$d3,$h2
	shr	\$2,$d3
	and	\$3,$h2
	add	$d3,%rax
	add	%rax,$h0
	adc	\$0,$h1
	adc	\$0,$h2
```

901 mod 395
= 506 mod 395
= 111 mod 395

= (100*(9 mod 4)
+ 5*(9/4)
+1) mod 395

5 * (x / 4) =
(x & ~3) + (x >> 2)

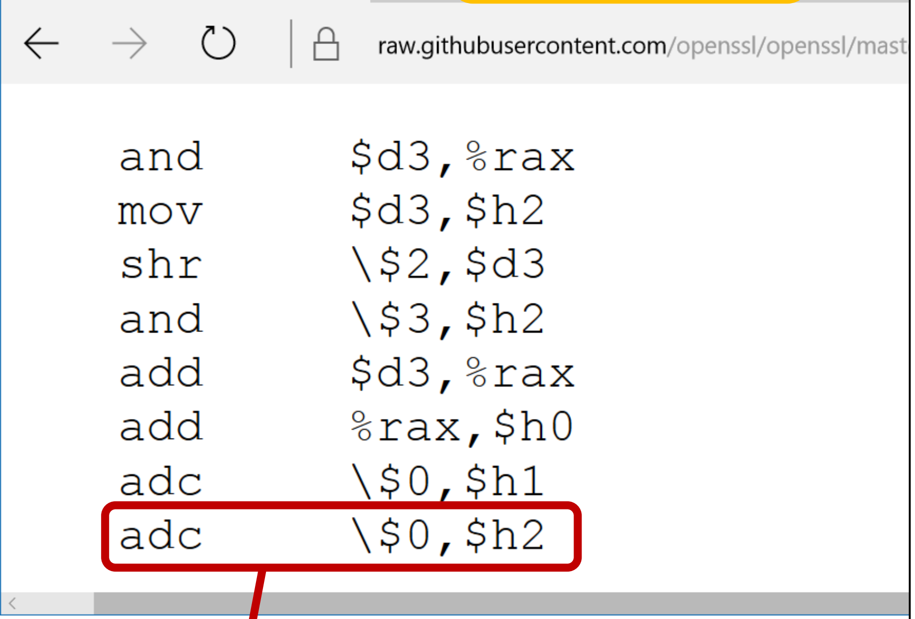
x mod 4 = x & 3

Vale Poly1305

TLS Protocol: **miTLS** (F*)

Crypto

HACL* (F*) | **Vale** (Asm)



```
and     $d3,%rax
mov     $d3,$h2
shr     \$2,$d3
and     \$3,$h2
add     $d3,%rax
add     %rax,$h0
adc     \$0,$h1
adc     \$0,$h2
```

Bug! This carry was originally missing!

```
procedure poly1305_reduce()
   ...
{
   ...
   And64(rax, d3);
   Mov64(h2, d3);
   Shr64(d3, 2);
   And64(h2, 3);
   Add64Wrap(rax, d3);
   Add64Wrap(h0, rax);
   Adc64Wrap(h1, 0);
   Adc64Wrap(h2, 0);
   ...
}
```

Vale Poly1305

```
procedure poly1305_reduce() returns(ghost hOut:int)
  let
    n := 0x1_0000_0000_0000_0000;
    p := 4 * n * n - 5;
    hIn := (n * n) * d3 + n * h1 + h0;
    d3 @= r10; h0 @= r14; h1 @= rbx; h2 @= rbp;
  modifies
    rax; r10; r14; rbx; rbp; efl;
  requires
    d3 / 4 * 5 < n;
    rax == n - 4;
  ensures
    hOut % p == hIn % p;
    hOut == (n * n) * h2 + n * h1 + h0;
    h2 < 5;
{
  lemma_BitwiseAdd64();
  lemma_poly_bits64();
  And64(rax, d3)…Adc64Wrap(h2, 0);
  ghost var h10 := n * old(h1) + old(h0);
  hOut := h10 + rax + (old(d3) % 4) * (n * n);
  lemma_poly_reduce(n, p, hIn, old(d3), h10, rax, hOut);
}
```

```
And64(rax, d3);
Mov64(h2, d3);
Shr64(d3, 2);
And64(h2, 3);
Add64Wrap(rax, d3);
Add64Wrap(h0, rax);
Adc64Wrap(h1, 0);
Adc64Wrap(h2, 0);
```

Vale Poly1305

```
procedure poly1305_reduce() returns(ghost hOut:int)
  let
    n := 0x1_0000_0000_0000_0000;
    p := 4 * n * n - 5;
    hIn := (n * n) * d3 + n * h1 + h0;
    d3 @= r10; h0 @= r14; h1 @= rbx; h2 @= rbp;
  modifies
    rax; r10; r14; rbx; rbp; efl;
  requires
    d3 / 4 * 5 < n;
    rax == n - 4;
  ensures
    hOut % p == hIn % p;
    hOut == (n * n) * h2 + n * h1 + h
    h2 < 5;
{
  lemma_BitwiseAdd64();
  lemma_poly_bits64();
  And64(rax, d3)…Adc64Wr  h2, 0);
  ghost var h10 := n * old( 1) + old(h0);
  hOut := h10 + rax + (o d(d3) % 4) * (n * n);
  lemma_poly_reduce(n, p, hIn, old(d3), h10, rax, hOut);
}
```

```
val lemma_poly_reduce (n p h h2 h10 c hh:int) :
  Lemma
  (requires
    p > 0 /\
    n * n > 0 /\
    h >= 0 /\ h2 >= 0 /\
    4 * (n * n) == p + 5 /\
    h2 == h / (n * n) /\
    h10 == h % (n * n) /\
    c == (h2 / 4) + (h2 / 4) * 4 /\
    hh == h10 + c + (h2 % 4) * (n * n))
  (ensures
    h % p == hh % p)
```

Demo: canonizer example

```
let demo_canonizer (a b c d e x:int) : Lemma
  (requires x == d * e)
  (ensures
    (a * (b * c) + (2 * d) * e == e * d + c * (b * a) + x)
  )
  =
  assert_by_tactic
    (a * (b * c) + (2 * d) * e == e * d + c * (b * a) + x)
    (fun _ -> canon_semiring int_cr)
```

Demo: Poly1305 via canonizer

(https://github.com/project-everest/hacl-star/blob/fstar-master/vale/code/crypto/poly1305/x64/Vale.Poly1305.Math.fst)

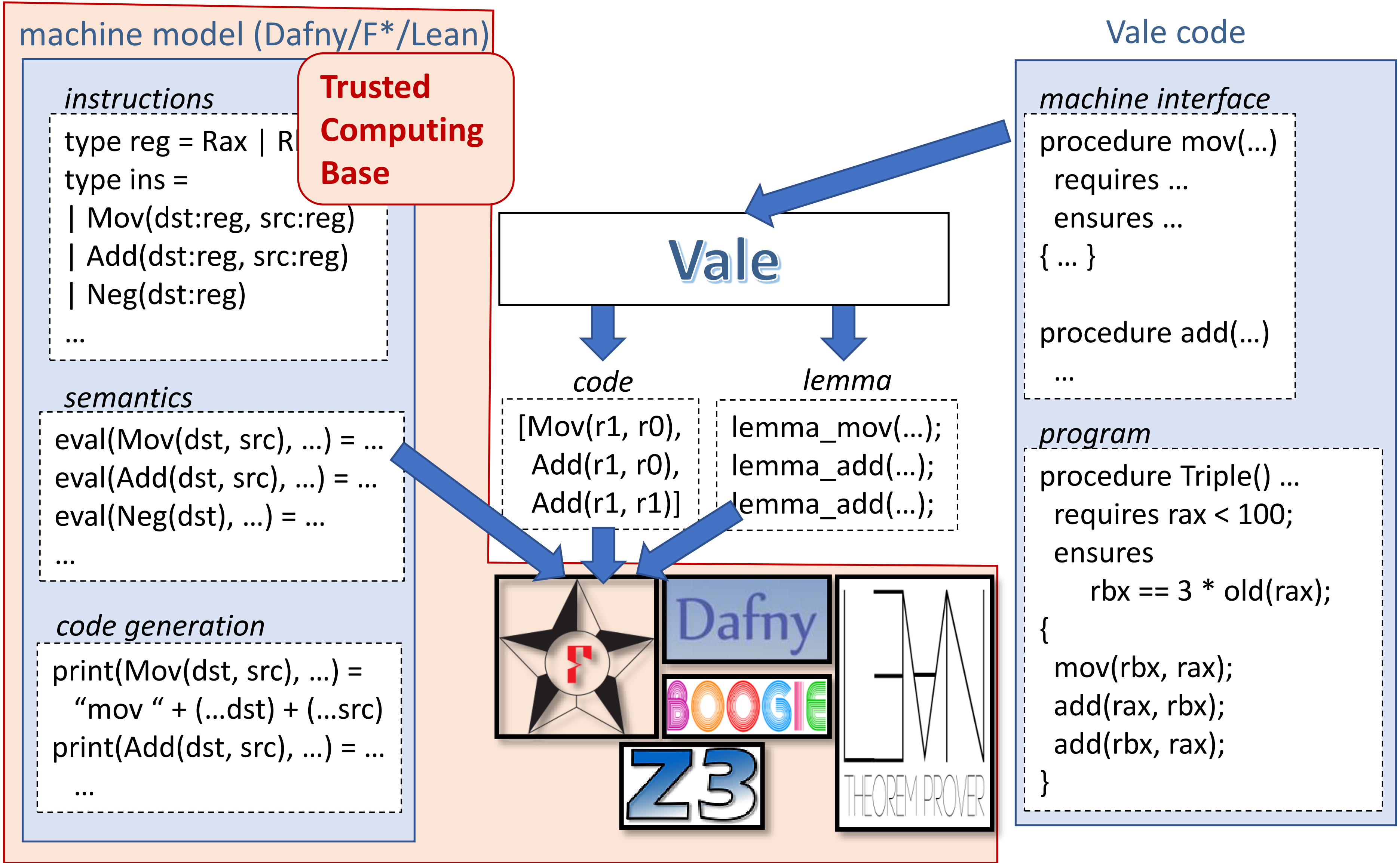
```
let lemma_poly_reduce (n:int) (p:int) (h:int) (h2:int) (h10:int) (c:int) (hh:int) =
  let h2_4 = h2 / 4 in
  let h2_m = h2 % 4 in
  let h_expand = h10 + (h2_4 * 4 + h2_m) * (n * n) in
  let hh_expand = h10 + (h2_m) * (n * n) + h2_4 * 5 in
  lemma_div_mod h (n * n);
  modulo_addition_lemma hh_expand p h2_4;
  assert_by_tactic (h_expand == hh_expand + h2_4 * (n * n * 4 + (-5)))
    (fun _ -> canon_semiring int_cr);
  ()
```

Using F* to verify systems software

(an F user's perspective)*

- Introductory examples
  - Bytes and words via normalization + SMT
  - Parsers/printers via tactics + SMT
    - EverParse library (USENIX Security 2019)
- Everest project and EverCrypt
  - Example cryptography: SHA, Poly1305
  - Poly1305 math via tactics + SMT
- Assembly language in Vale/F*
  - Efficient verification conditions via normalization + SMT

Vale: extensible, automated assembly language verification



machine model (Dafny/F*/Lean)

Trusted Computing Base

instructions

```
type reg = Rax | R
type ins =
| Mov(dst:reg, src:reg)
| Add(dst:reg, src:reg)
| Neg(dst:reg)
…
```

semantics

```
eval(Mov(dst, src), …) = …
eval(Add(dst, src), …) = …
eval(Neg(dst), …) = …
…
```

code generation

```
print(Mov(dst, src), …) =
  “mov “ + (…dst) + (…src)
print(Add(dst, src), …) = …
  …
```

Vale

code

```
[Mov(r1, r0),
 Add(r1, r0),
 Add(r1, r1)]
```

lemma

```
lemma_mov(…);
lemma_add(…);
lemma_add(…);
```

Vale code

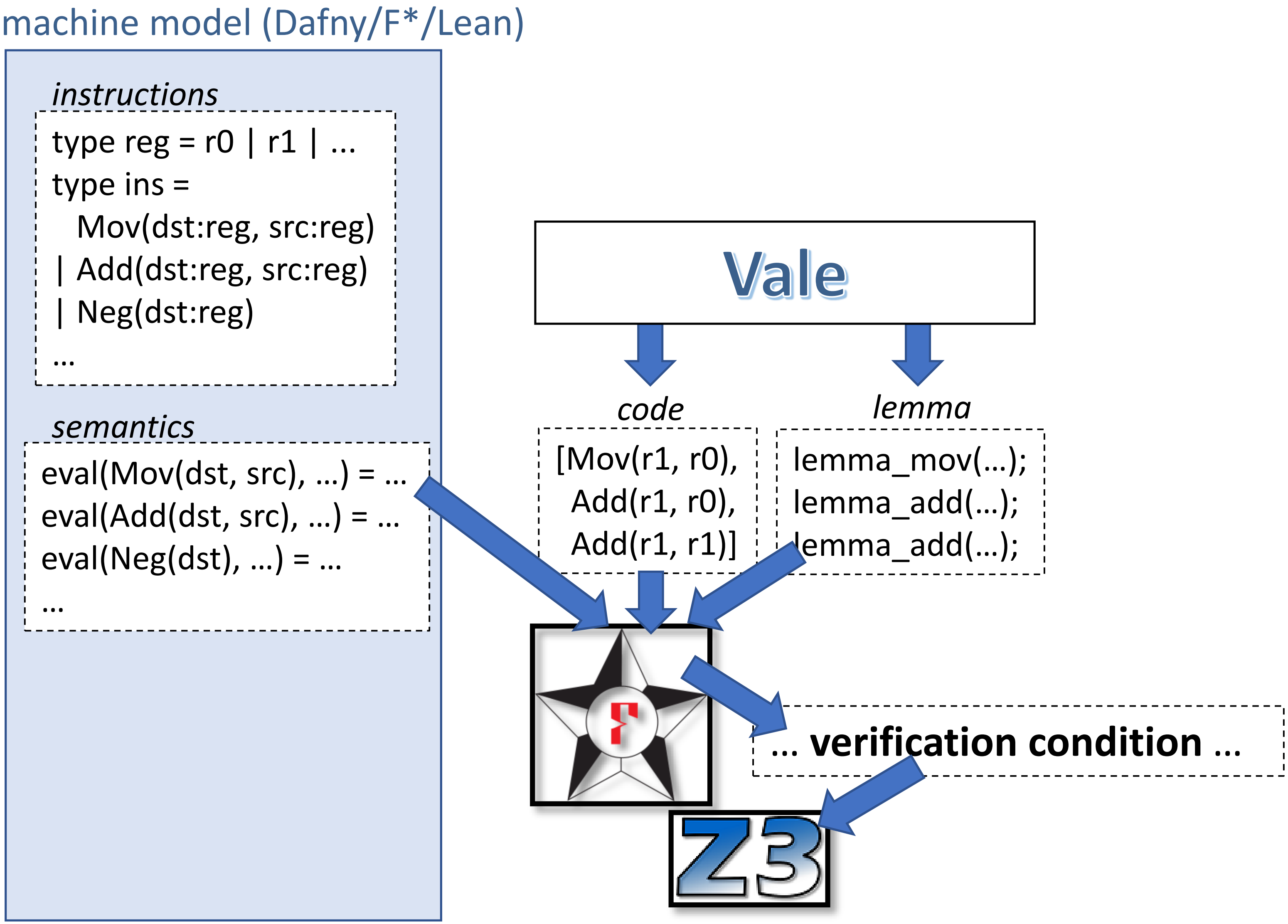
machine interface

```
procedure mov(…)
  requires …
  ensures …
{ … }

procedure add(…)
  …
```

program

```
procedure Triple() …
  requires rax < 100;
  ensures
      rbx == 3 * old(rax);
{
  mov(rbx, rax);
  add(rax, rbx);
  add(rbx, rax);
}
```

Vale: extensible, automated assembly language verification



machine model (Dafny/F*/Lean)

instructions

```
type reg = r0 | r1 | ...
type ins =
   Mov(dst:reg, src:reg)
| Add(dst:reg, src:reg)
| Neg(dst:reg)
...
```

semantics

```
eval(Mov(dst, src), ...) = ...
eval(Add(dst, src), ...) = ...
eval(Neg(dst), ...) = ...
...
```

Vale

code

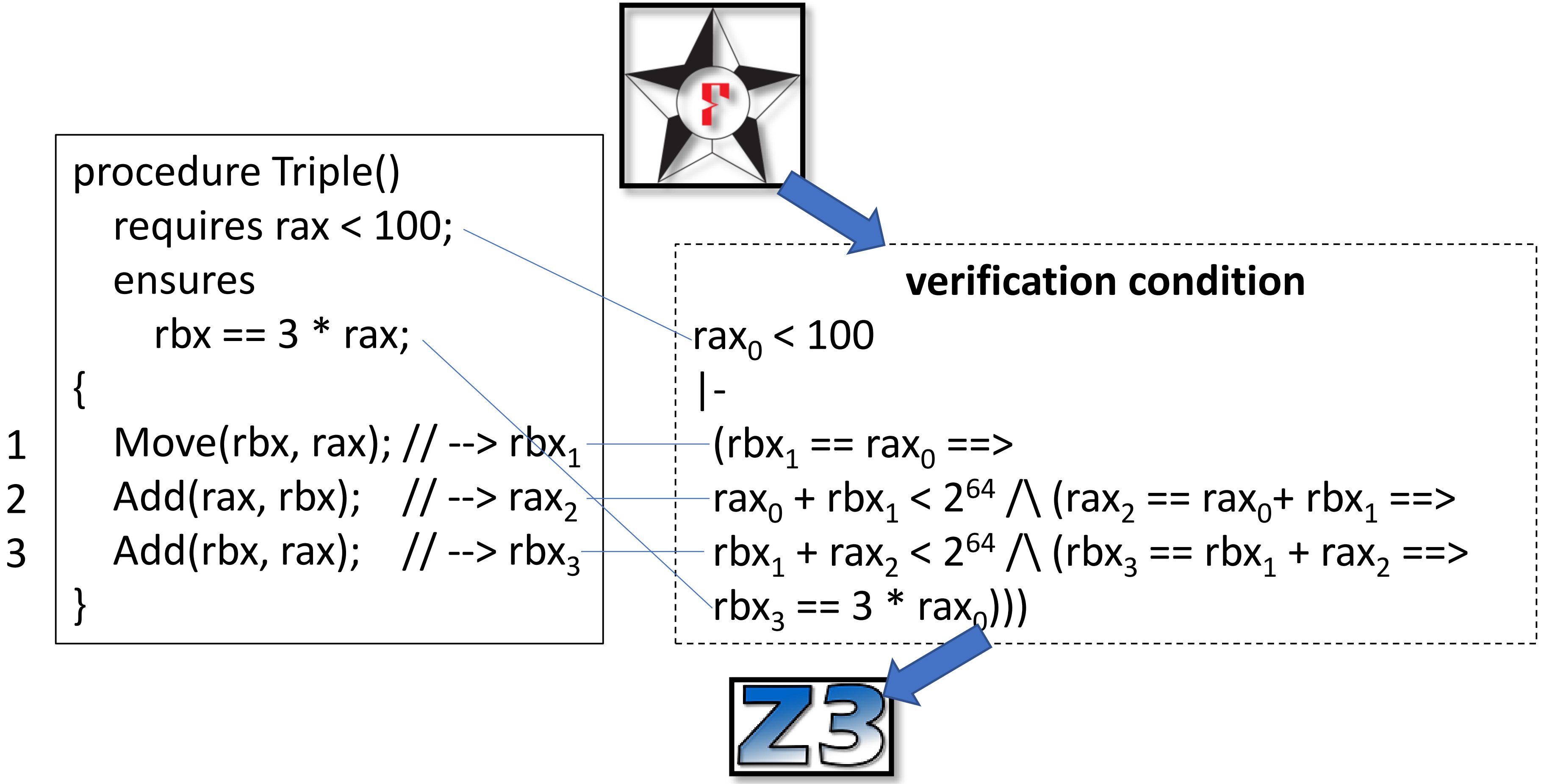
```
[Mov(r1, r0),
  Add(r1, r0),
  Add(r1, r1)]
```

lemma

```
lemma_mov(...);
lemma_add(...);
lemma_add(...);
```

... verification condition ...

Verification condition



```
procedure Triple()
   requires rax < 100;
   ensures
      rbx == 3 * rax;
{
   Move(rbx, rax); // --> rbx_1
   Add(rax, rbx);    // --> rax_2
   Add(rbx, rax);    // --> rbx_3
}
```

verification condition

$rax_0 < 100$

$|-$

$(rbx_1 == rax_0 ==>$

$rax_0 + rbx_1 < 2^{64} /\backslash (rax_2 == rax_0 + rbx_1 ==>$

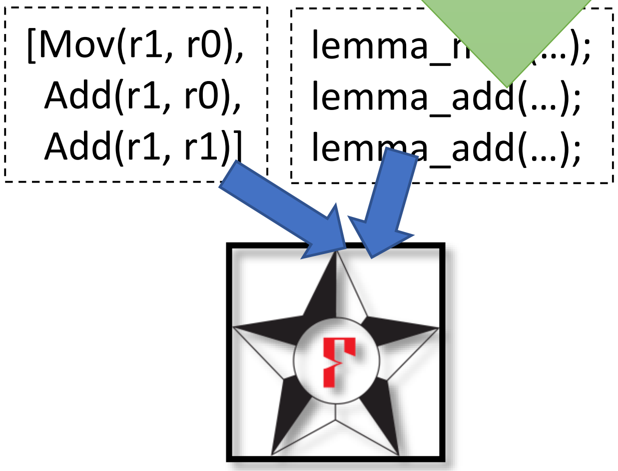
$rbx_1 + rax_2 < 2^{64} /\backslash (rbx_3 == rbx_1 + rax_2 ==>$

$rbx_3 == 3 * rax_0)))$

States, lemmas

```
s1 : state
s2 : state
type state = {
    ok:bool;
    regs:regs;
    flags:nat64;
    mem:mem;
}
```

```
lemma_add (...)...
  requires ...
     s1.ok /\
     valid_operand s1 dst /\
     valid_operand s1 src /\
     (   eval_operand s1 dst
      + eval_operand s1 src  ) < 2^64
  ensures ...
     s2.ok /\
     s2 == (...framing... s1) /\
     eval_operand s2 dst ==
               (  eval_operand s1 dst,
               + eval_operand s1 src)
```



```
[Mov(r1, r0),
  Add(r1, r0),
  Add(r1, r1)]
```

```
lemma_r   ...);
lemma_add(...);
lemma_add(...);
```

Ugh! Default SMT query looks awful!

verification condition we want:

……………………… $(rax_2 == rax_0 + rbx_1 ==>$

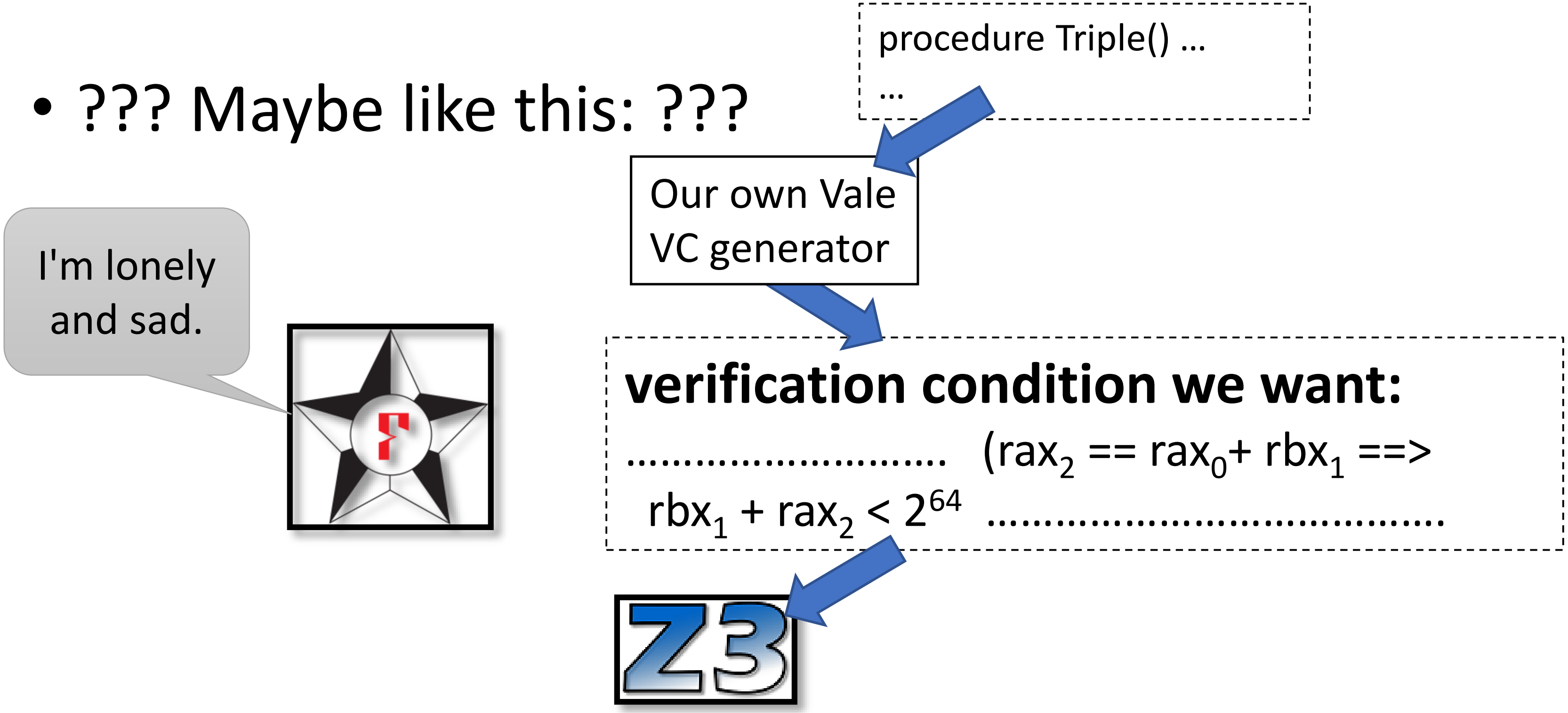
$rbx_1 + rax_2 < 2^{64}$ ………………………………

verification condition we get:

```
…
(forall (ghost_result_0:(state * fuel)).
  (let (s3, fc3) = ghost_result_0 in
    eval_code (Ins (Add64 (OReg (Rax)) (OReg (Rbx)))) fc3 s2 == Some s3 /\
    eval_operand (OReg Rax) s3 == eval_operand (OReg Rax) s2 + eval_operand (OReg Rbx) s2 /\
    s3 == update_state (OReg Rax).r s3 s2) ==>
  lemma_Add s2 (OReg Rax) (OReg Rbx) == ghost_result_0 ==>
  (forall (s3:state) (fc3:fuel). lemma_Add s2 (OReg Rax) (OReg Rbx) == Mktuple2 s3 fc3 ==>
    Cons? codes_Triple.tl /\
    (forall (any_result0:list code). codes_Triple.tl == any_result0 ==>
      (forall (any_result1:list code). codes_Triple.tl.tl == any_result1 ==>
        OReg? (OReg Rbx) /\ eval_operand (OReg Rbx) s3 + eval_operand (OReg Rax) s3 < 2^64
…
```

Let's write our own VC generator!

- ??? Maybe like this: ???



procedure Triple() …
…

Our own Vale VC generator

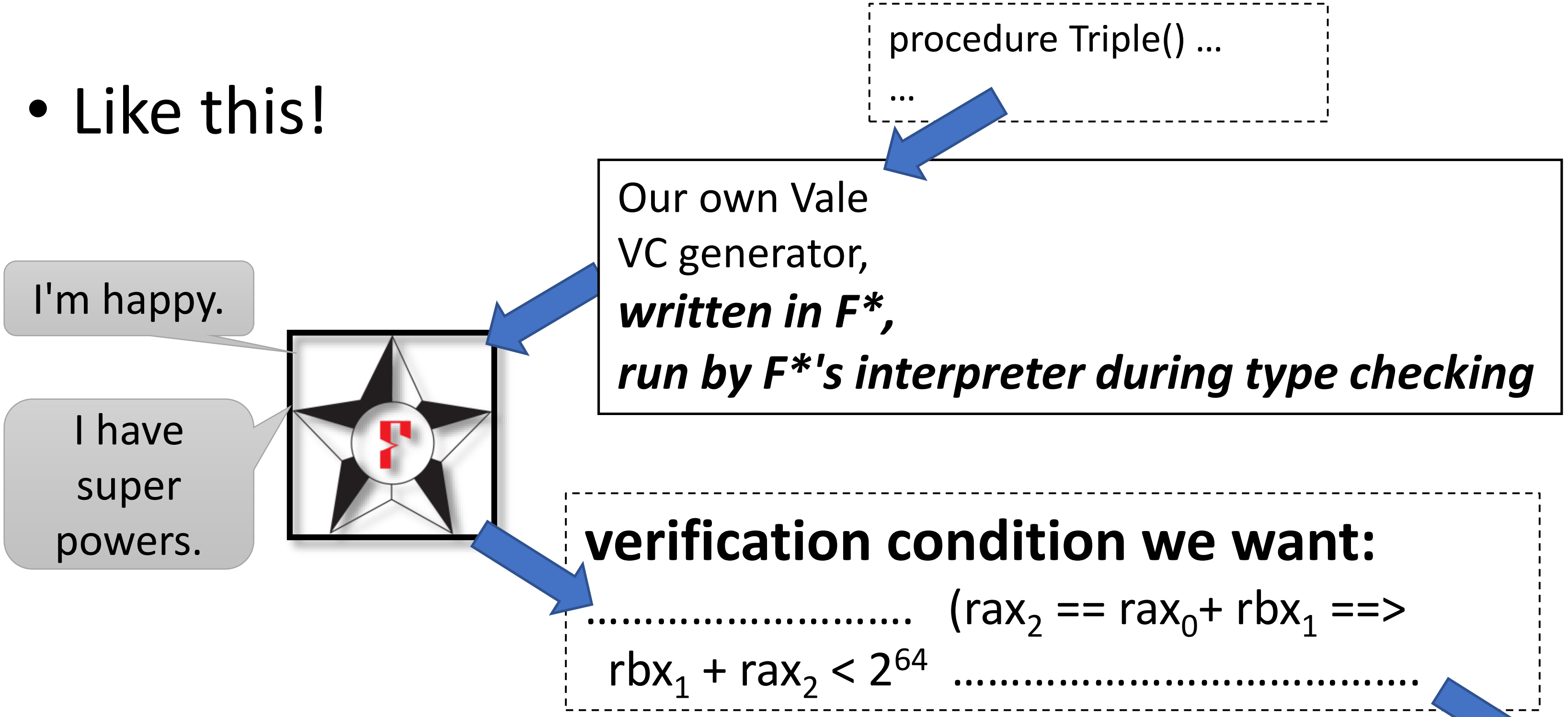
verification condition we want:

.... (rax$_2$ == rax$_0$+ rbx$_1$ ==> rbx$_1$ + rax$_2$ < 2^{64} ....

- But won't it be part of TCB?
- And how do we interact with F*?
- Can we reuse F* features and libraries?

Let's write our own VC generator!

- Like this!



procedure Triple() …
…

Our own Vale
VC generator,
written in F*,
run by F*'s interpreter during type checking

I'm happy.

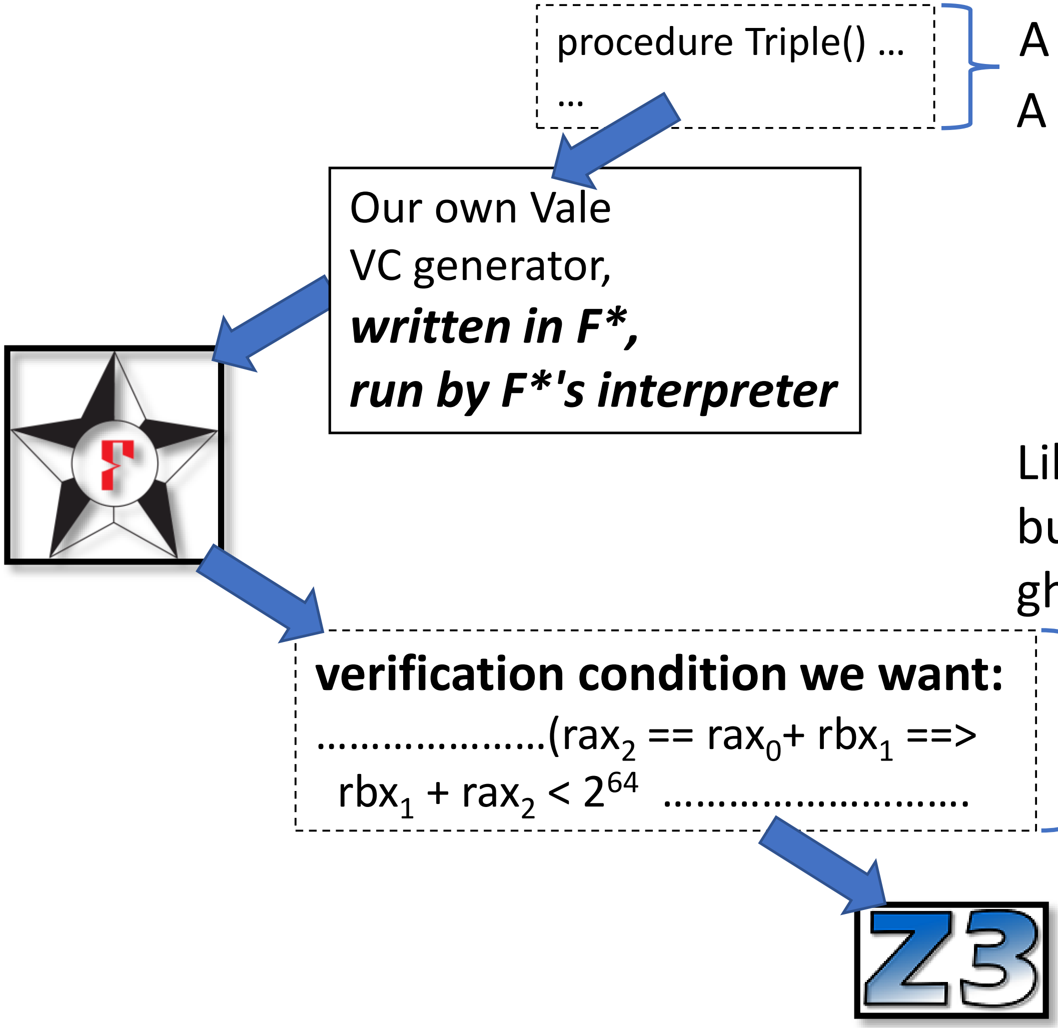
I have
super
powers.

verification condition we want:

………………………… $(rax_2 == rax_0 + rbx_1 ==>$
$rbx_1 + rax_2 < 2^{64}$ ……………………………………

- Part of TCB? ***No -- we verify its soundness in F****
- Interact with F*? ***Yes***
- Reuse F* features and libraries? ***Yes***

Let's write our own VC generator!



procedure Triple() ...
...

Our own Vale
VC generator,
written in F*,
run by F*'s interpreter

verification condition we want:

…………………(rax$_2$ == rax$_0$+ rbx$_1$ ==>
rbx$_1$ + rax$_2$ < 2^{64} ………………………..

~~A b[illegible]ing?~~

A datatype:

```
type quickCode = ...
type quickCodes =
| QEmpty
| QSeq of quickCode * quickCodes ...
| QLemma of ... (Lemma pre post) * ...
```

Like our earlier code AST,
but with assertions, lemma calls,
ghost variables, etc.

~~A b[illegible]ing?~~

~~A d[illegible]ype?~~

An F* term:

(forall rbx$_1$. rbx$_1$ == rax$_0$ ==>
rax$_0$ + rbx$_1$ < 2^{64} /\
(forall rax$_2$. rax$_2$ == rax$_0$+ rbx$_1$ ==>
rbx$_1$ + rax$_2$ < 2^{64} /\ ...

VC generator definition (in F*)

```
let rec vc_gen (cs:list code) (qcs:quickCodes cs) (k:state -> Type) : state -> Type =
  fun (s0:state) ->
    match qcs with
    | QEmpty -> k s0
    | QSeq qc qcs' -> qc.wp (vc_gen cs.tl qcs' k) s0
    | QLemma pre post lem qcs' -> pre /\ (post ==> vc_gen cs qcs' k s0)
```

```
procedure Triple() …{
  mov(rbx, rax);
  lemma_two_plus_two_is_four();
  add(rax, rbx);
  add(rbx, rax);
}
```

```
(QSeq (qc_mov Rbx Rax)
  (QLemma True (2+2==4) lemma_two_plus_to
    (QSeq (qc_add Rax Rbx)
      (QSeq (qc_add Rbx Rax)
        (QEmpty))))
```

VC generator soundness (in F*)

```
let rec vc_gen (cs:list code) (qcs:quickCodes cs) (k:state -> Type) : state -> Type =
  fun (s0:state) ->
    match qcs with
    | QEmpty -> k s0
    | QSeq qc qcs' -> qc.wp (vc_gen cs.tl qcs' k) s0
    | QLemma pre post lem qcs' -> pre /\ (post ==> vc_gen cs qcs' k s0)

val vc_sound (cs:list code) (qcs:quickCodes cs) (k:state -> Type) (s0:state) : Lemma
  (requires                (vc_gen cs qcs k s0))
  (ensures (let sN = eval_code cs s0 in k sN))
```

verification condition we want:

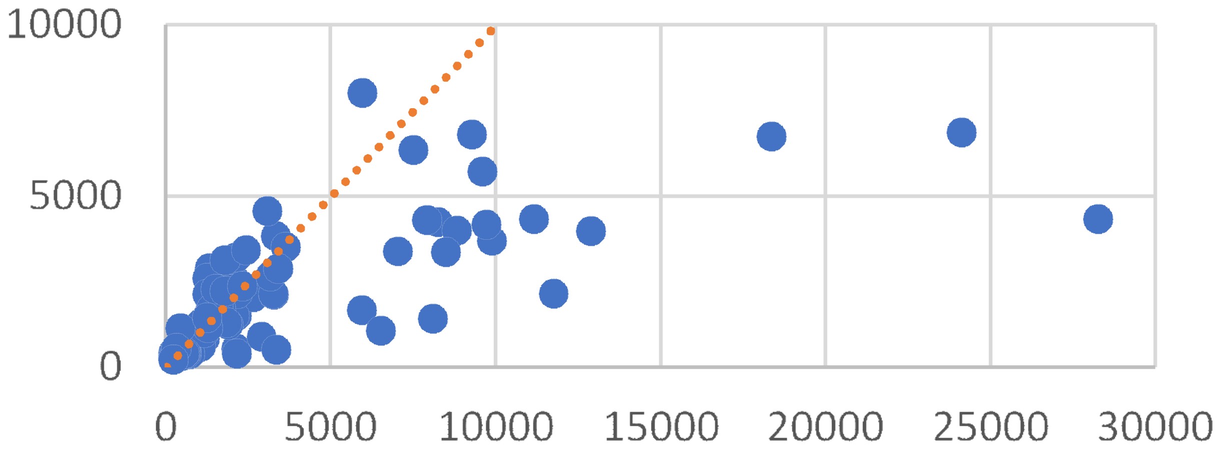
…………………(rax_2 == rax_0+ rbx_1 ==>
rbx_1 + rax_2 < 2^{64} ……………………….

```
… vc_sound […] (QSeq (qc_mov Rbx Rax) (QLemma True …))) k s0 …
```

Verification performance

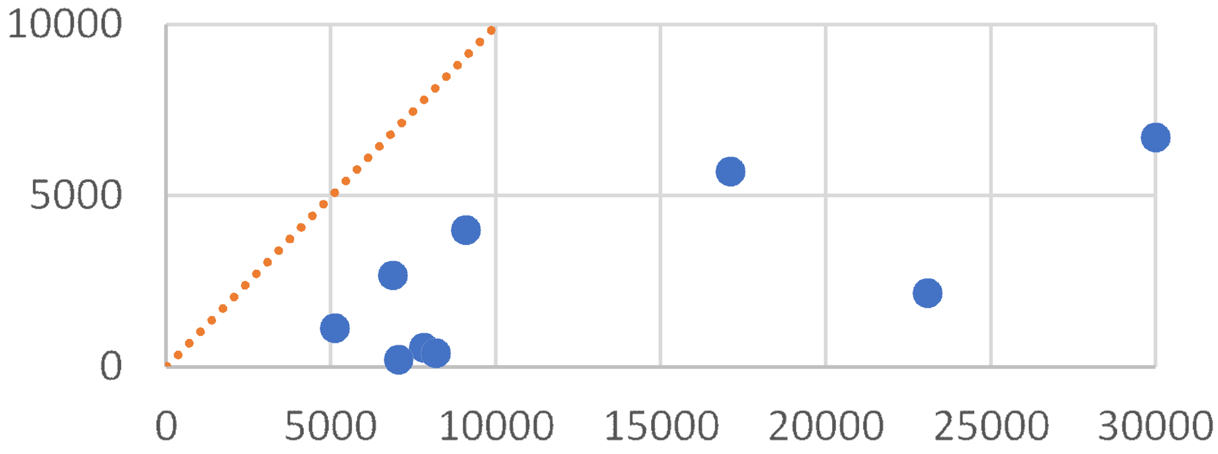
Response time to verify each Poly1305 and AES-GCM Vale procedure

x-axis: **Vale/F*$_{\text{naive}}$** (ms) y-axis**: Vale/F*** (ms)



Response time to verify each Poly1305 Vale procedure

x-axis: **Vale/Dafny** (ms) y-axis: **Vale/F*** (ms)



Using F* to verify systems software

(an F user's perspective)*

- Introductory examples
  - Bytes and words via normalization + SMT
  - Parsers/printers via tactics + SMT
    - EverParse library (USENIX Security 2019)
- Everest project and EverCrypt
  - Example cryptography: SHA, Poly1305
  - Poly1305 math via tactics + SMT
- Assembly language in Vale/F*
  - Efficient verification conditions via normalization + SMT

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