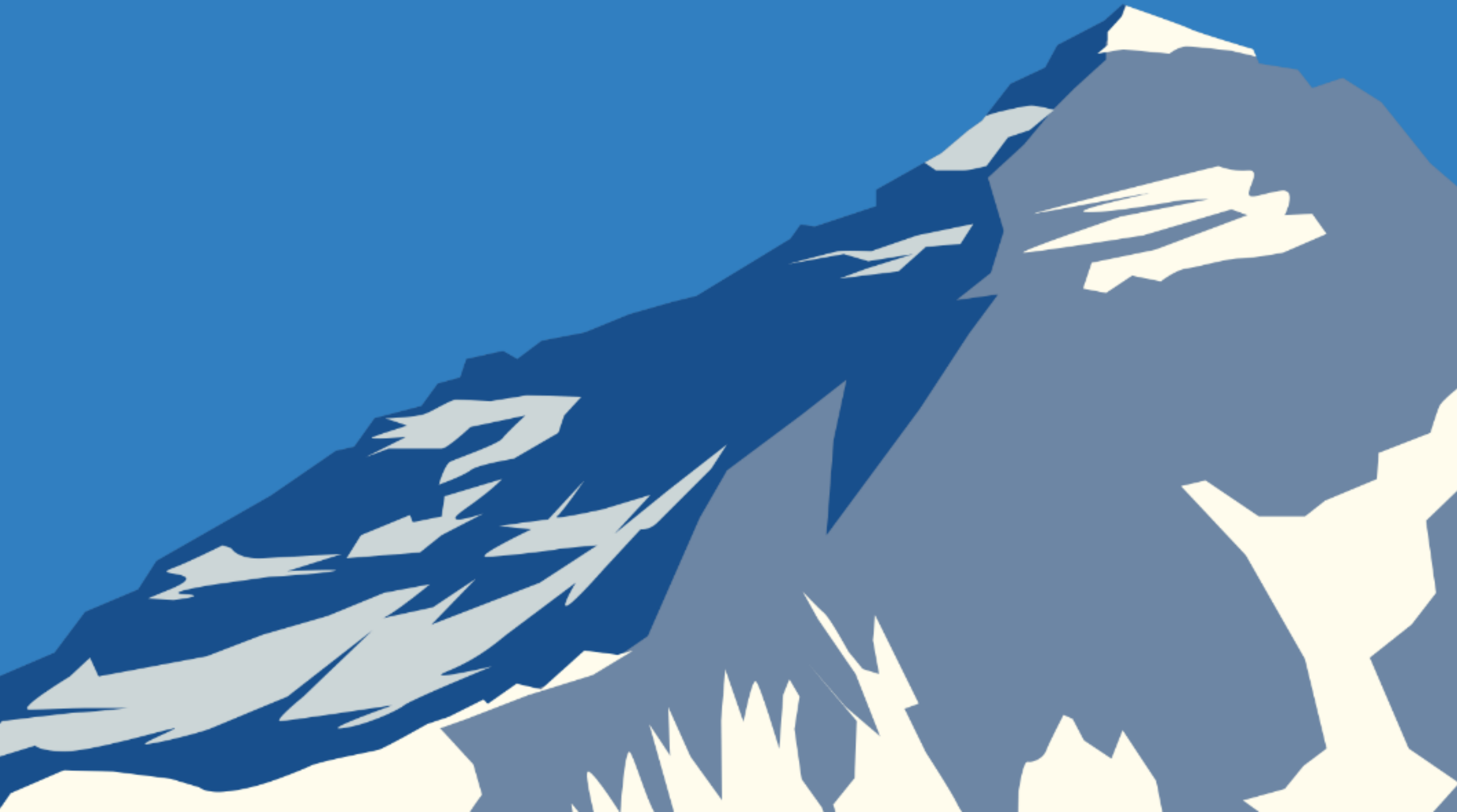
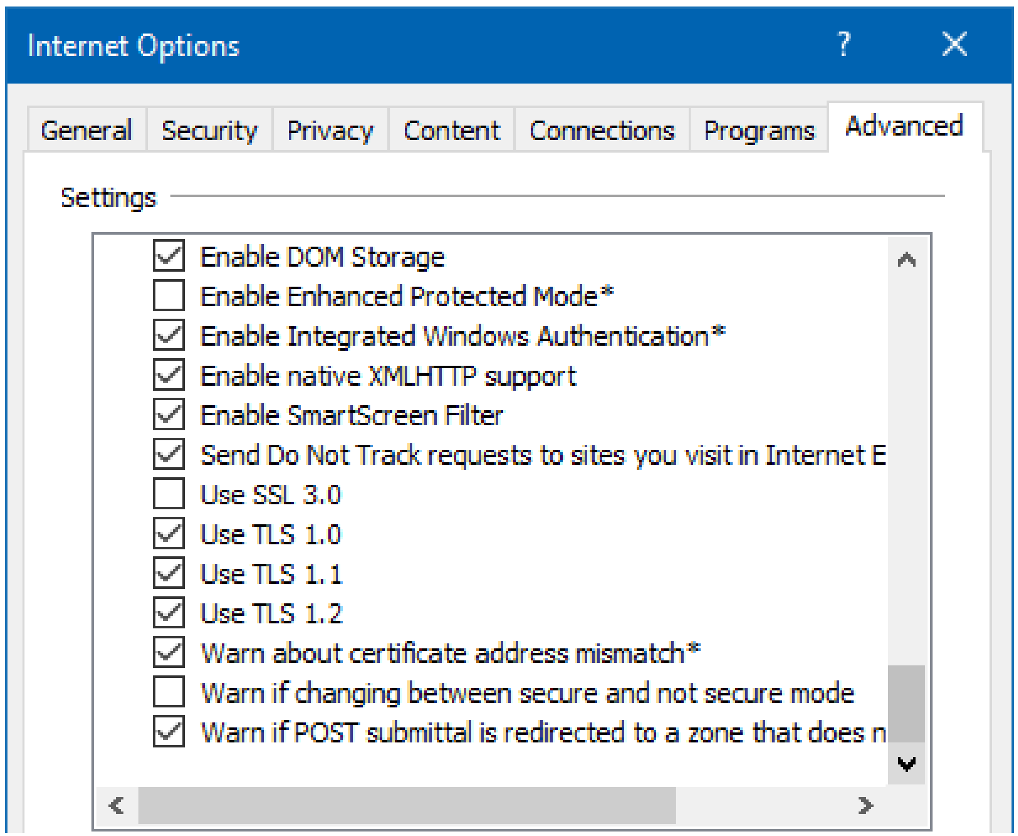
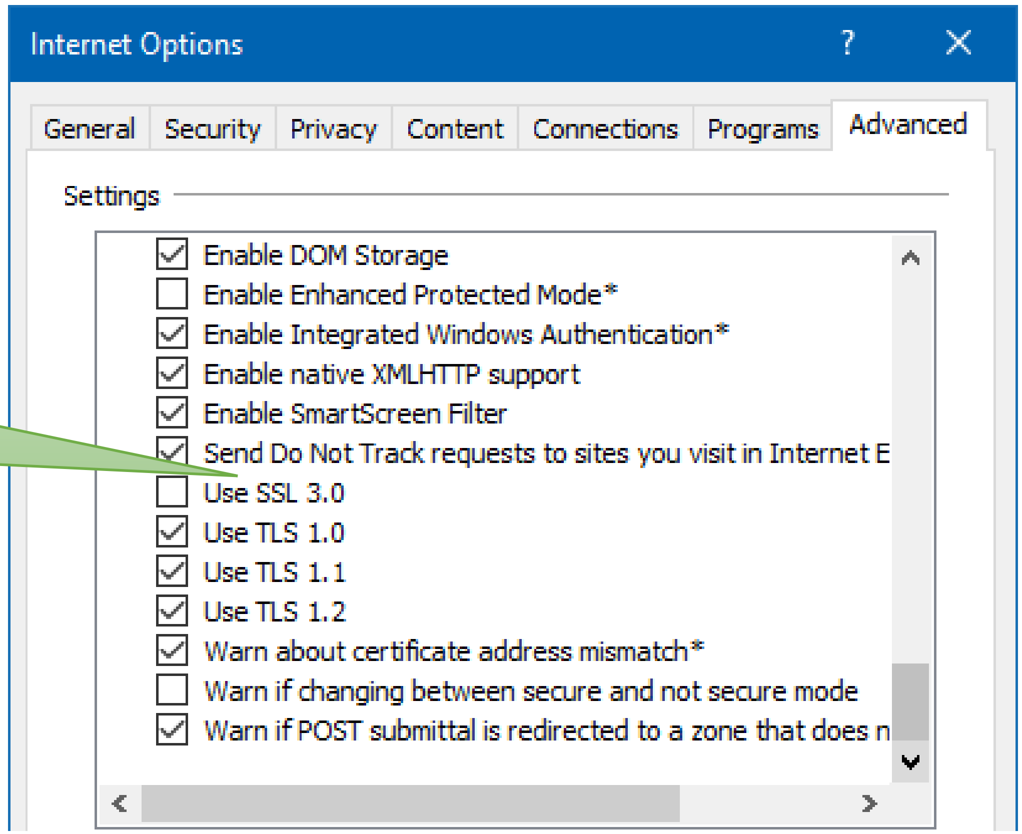
Verifying Assembly Language with Vale



Secure communication
confidentiality, integrity, authentication



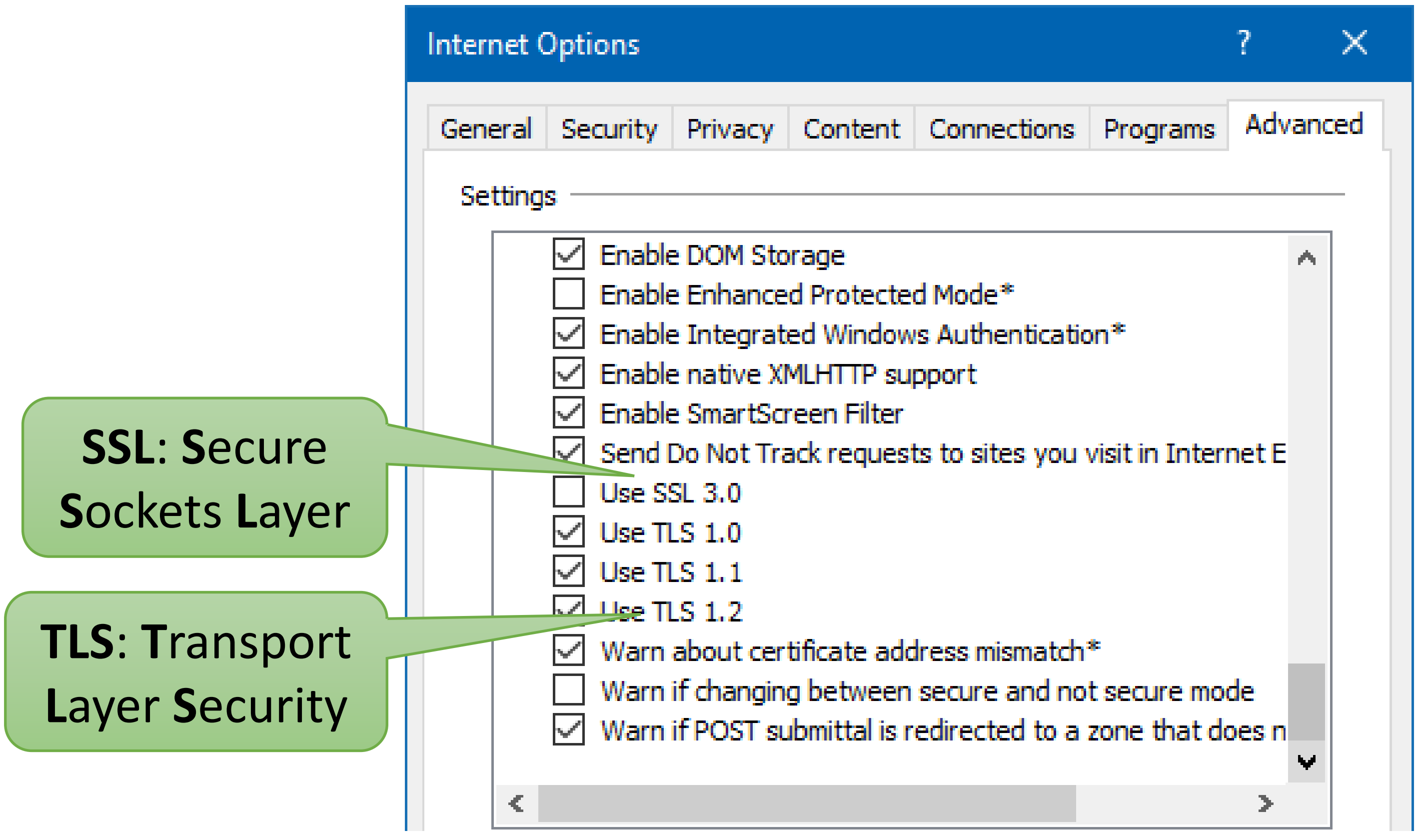
Secure communication
confidentiality, integrity, authentication



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

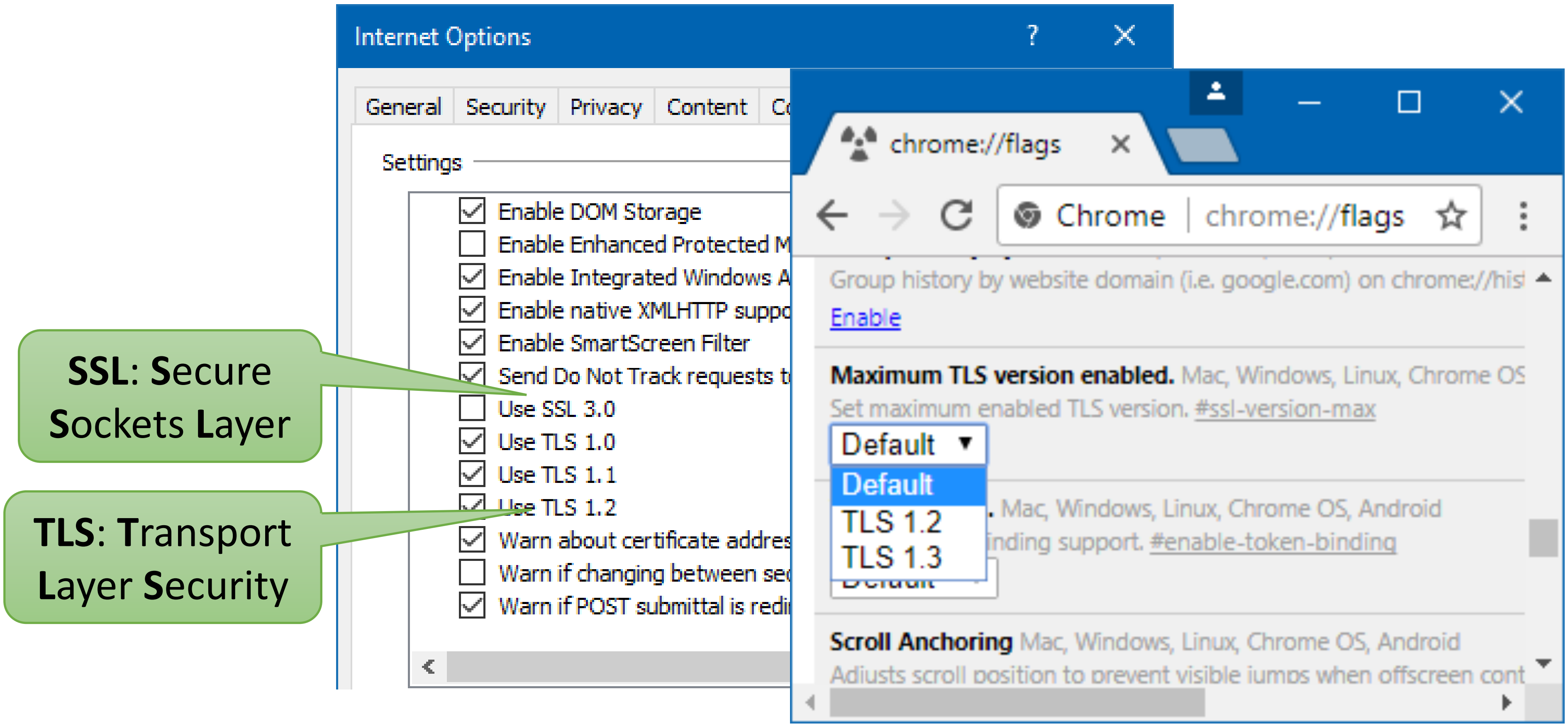
Secure communication

confidentiality, integrity, authentication

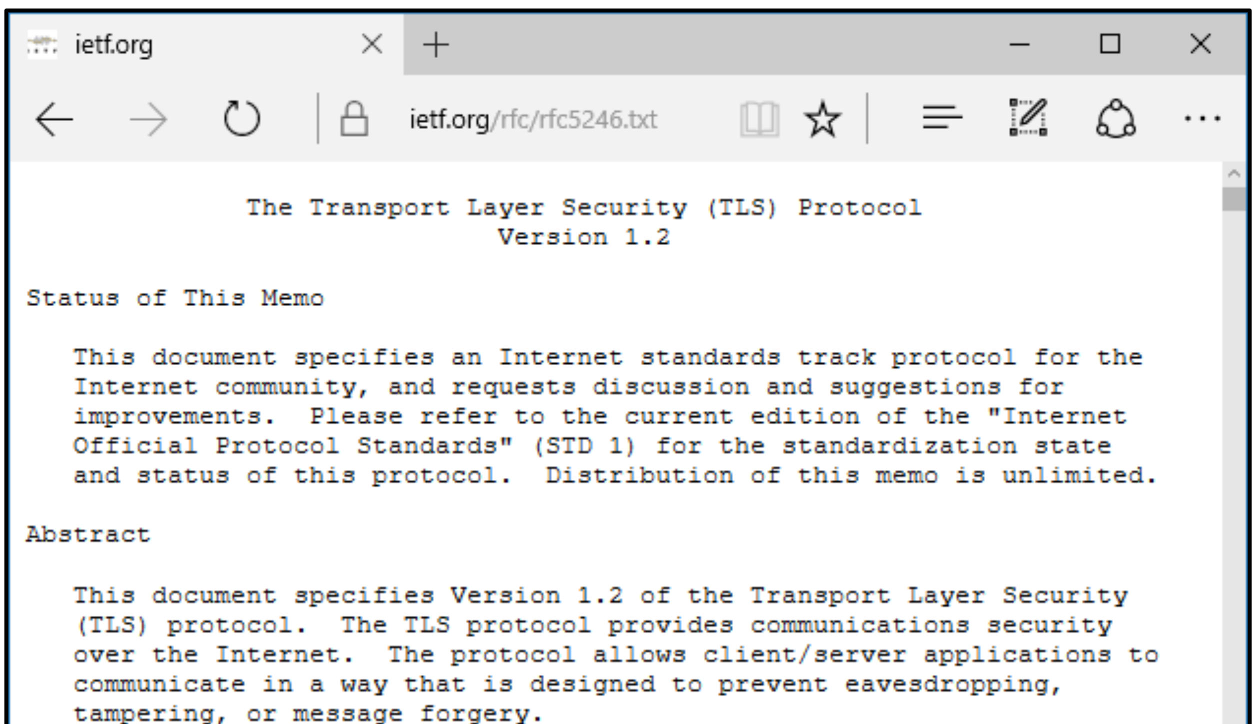


Secure communication

confidentiality, integrity, authentication



TLS standards, some implementations



The Transport Layer Security (TLS) Protocol
Version 1.2

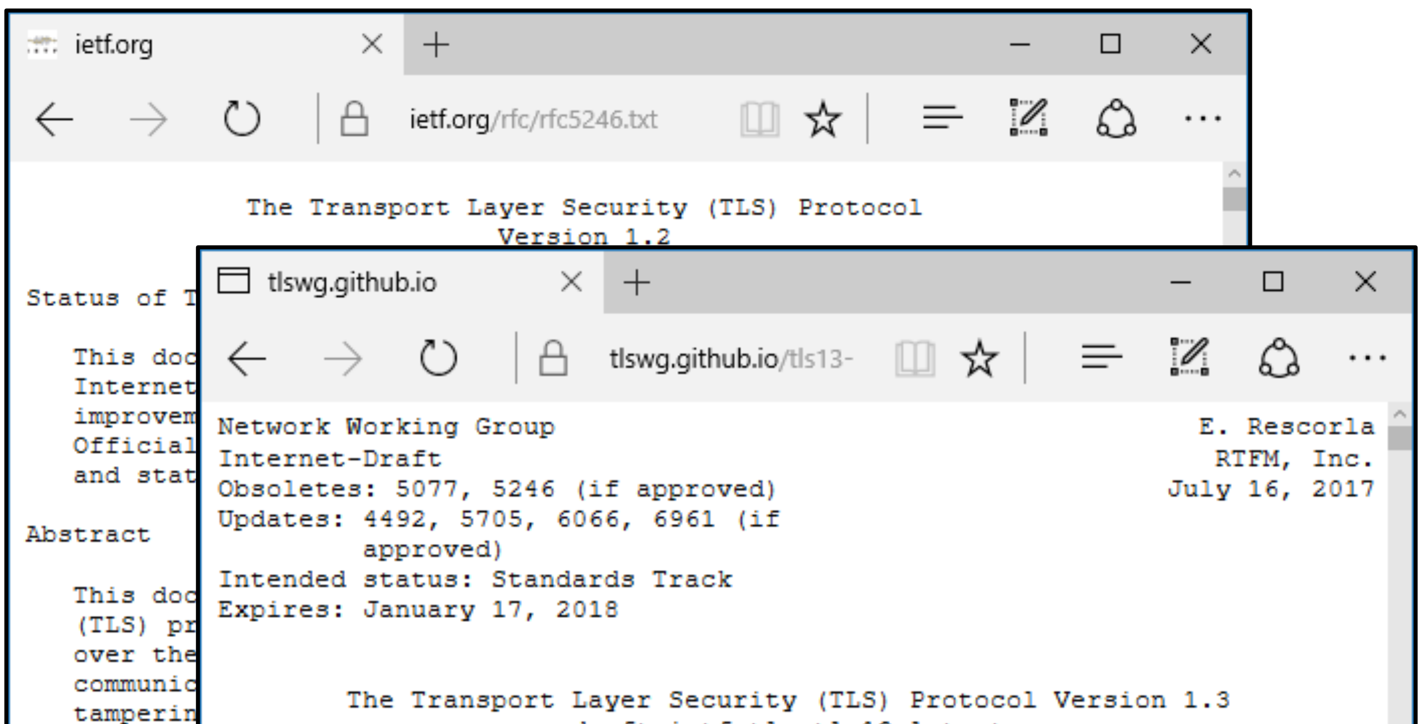
Status of This Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

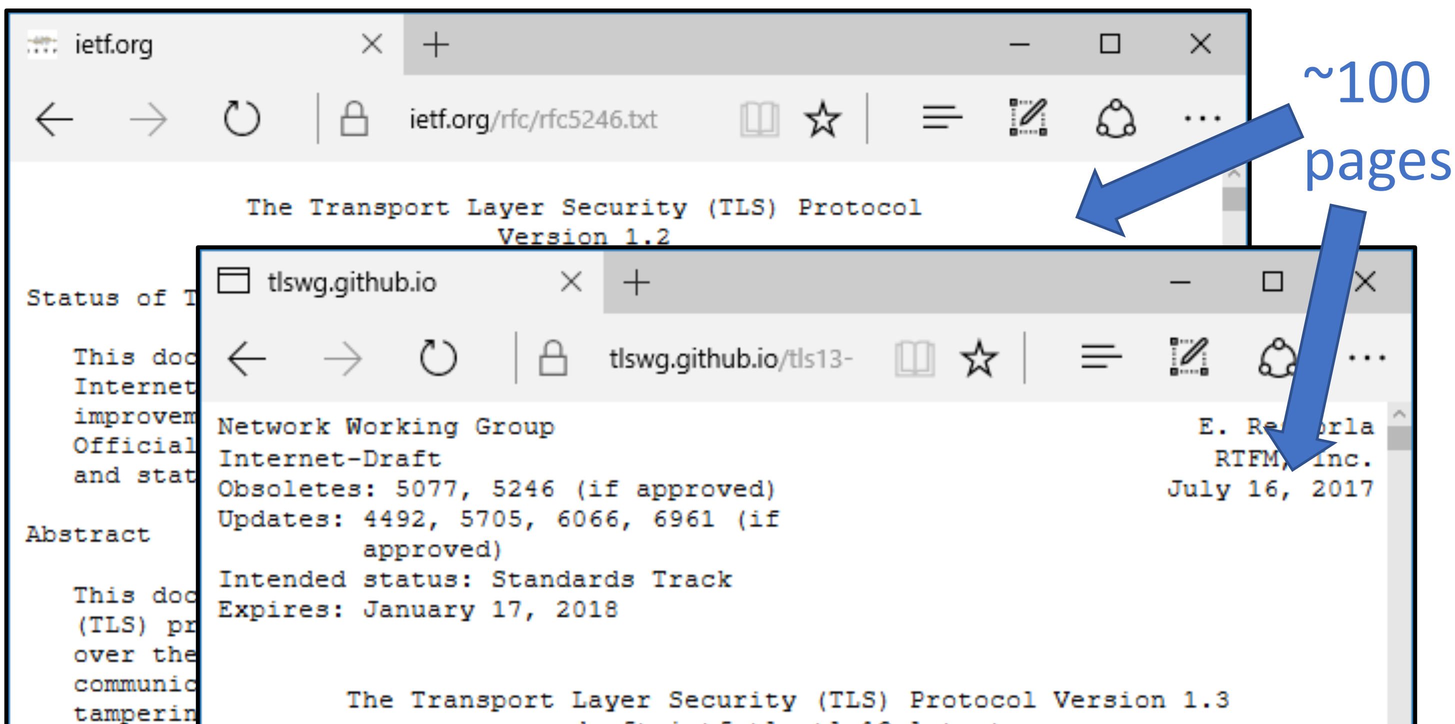
Abstract

This document specifies Version 1.2 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.

TLS standards, some implementations

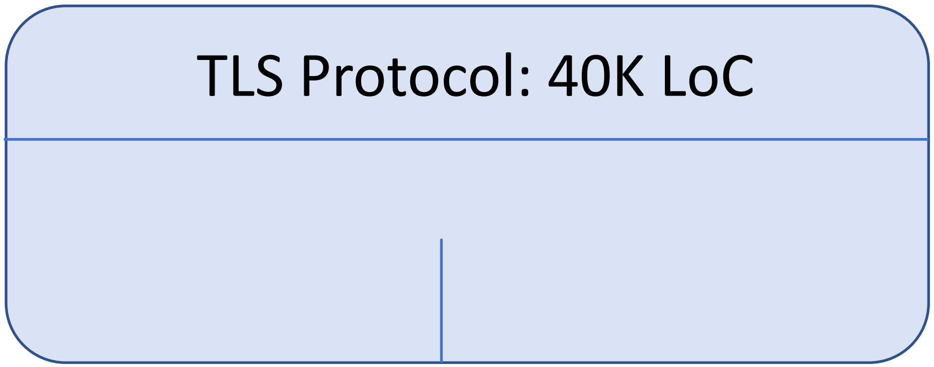


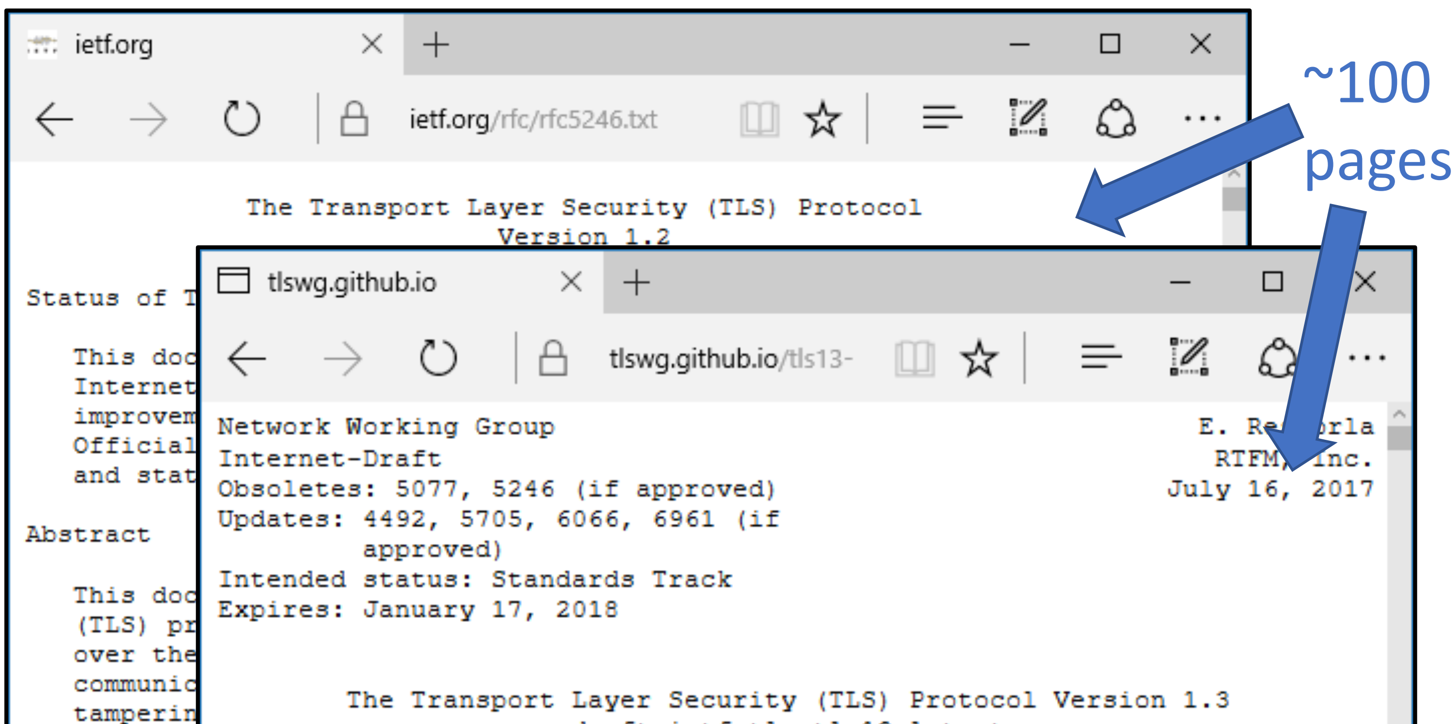
TLS standards, some implementations



TLS standards, some implementations

OpenSSL





Lines-of-Code measured with SLOCCount

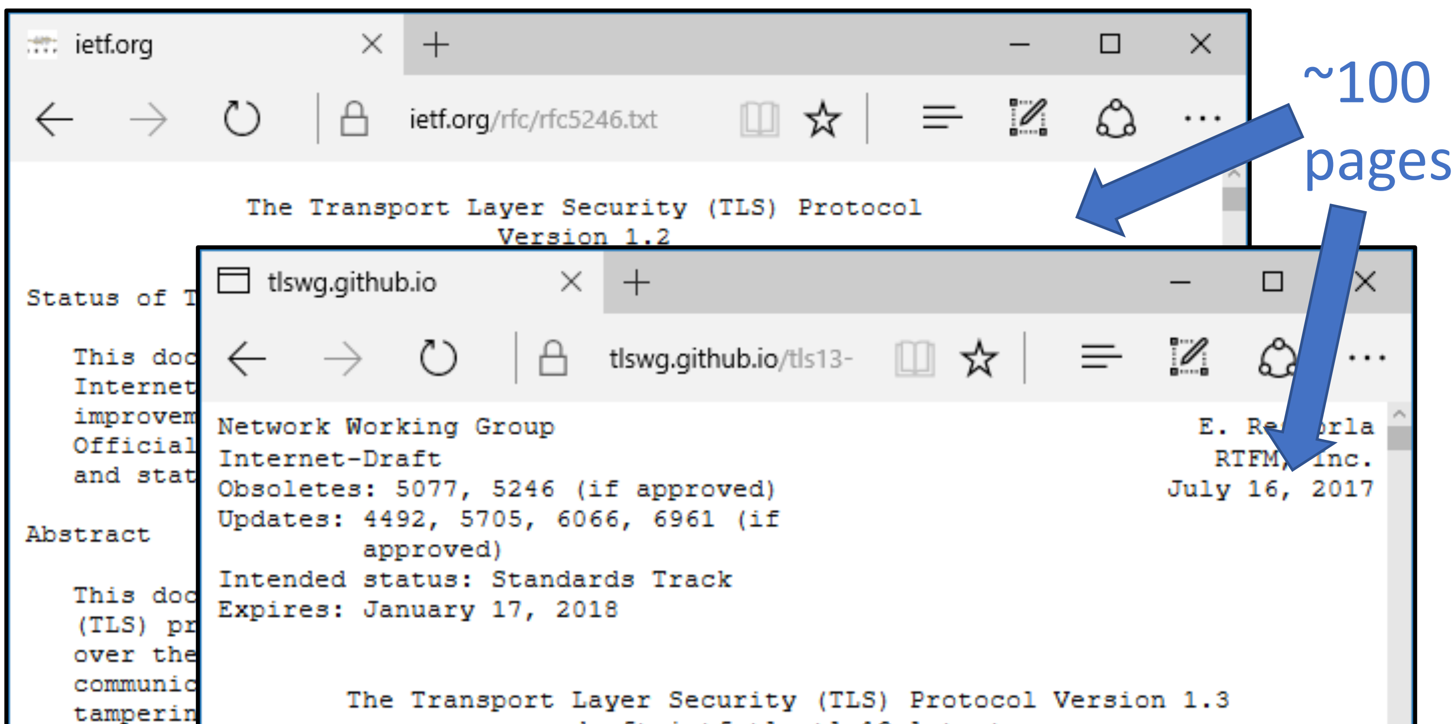
TLS standards, some implementations

OpenSSL

TLS Protocol: 40K LoC

Crypto

C: 160K LoC



Lines-of-Code measured with SLOCCount

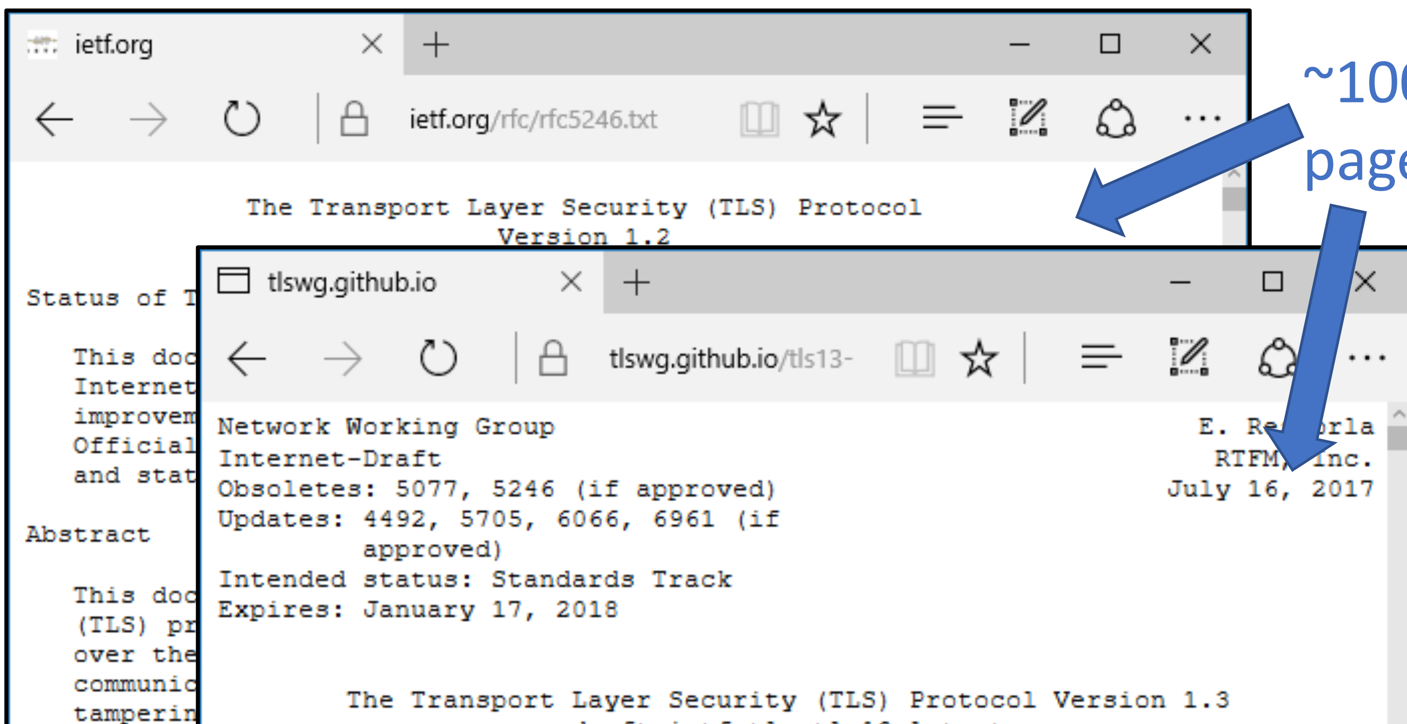
TLS standards, some implementations

OpenSSL

TLS Protocol: 40K LoC

Crypto

C: 160K LoC | Asm: 150K LoC

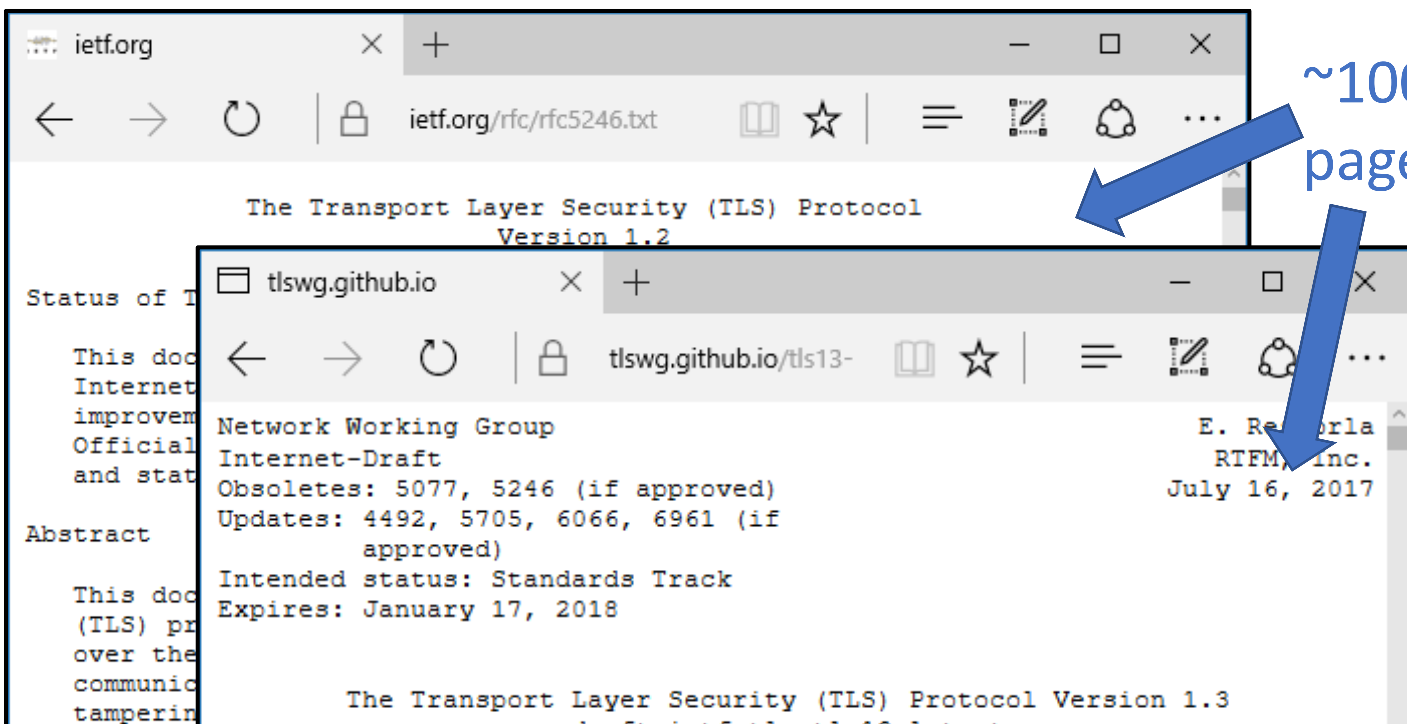


~100 pages

Lines-of-Code measured with SLOCCount

TLS standards, some implementations

OpenSSL		BoringSSL	
TLS Protocol: 40K LoC		TLS Protocol: 30K LoC	
Crypto		Crypto	
C: 160K LoC	Asm: 150K LoC	C: 100K LoC	Asm: 60K LoC



~100 pages

Lines-of-Code measured with SLOCCount

Crypto implementation bugs

Crypto implementation bugs

```
OpenSSL Security Advisory [10 Nov 2016]
========================================

ChaCha20/Poly1305 heap-buffer-overflow (CVE-2016-7054)
======================================================

Severity: High

TLS connections using *-CHACHA20-POLY1305 ciphersuites are susceptible to a DoS
attack by corrupting larger payloads. This can result in an OpenSSL crash. This
issue is not considered to be exploitable beyond a DoS.
```

Crypto implementation bugs

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

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

Severity: High

TLS connections using *-C                                   to a DoS
attack by corrupting larg                                   ash. This
issue is not considered t
```

[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
Poly1305 test failed.
got:      2637408fe03086ea73f971e3425e2820
expected: 2637408fe13086ea73f971e3425e2820

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

This was found because a run of our SSL tests happened to find a
problematic input. I've trimmed it down to the first block where they
disagree.

I'm probably going to write something to generate random inputs and stress
all your other poly1305 codepaths against a reference implementation. I
recommend doing the same in your own test harness, to make sure there
aren't others of these bugs lurking around.
```

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.
```

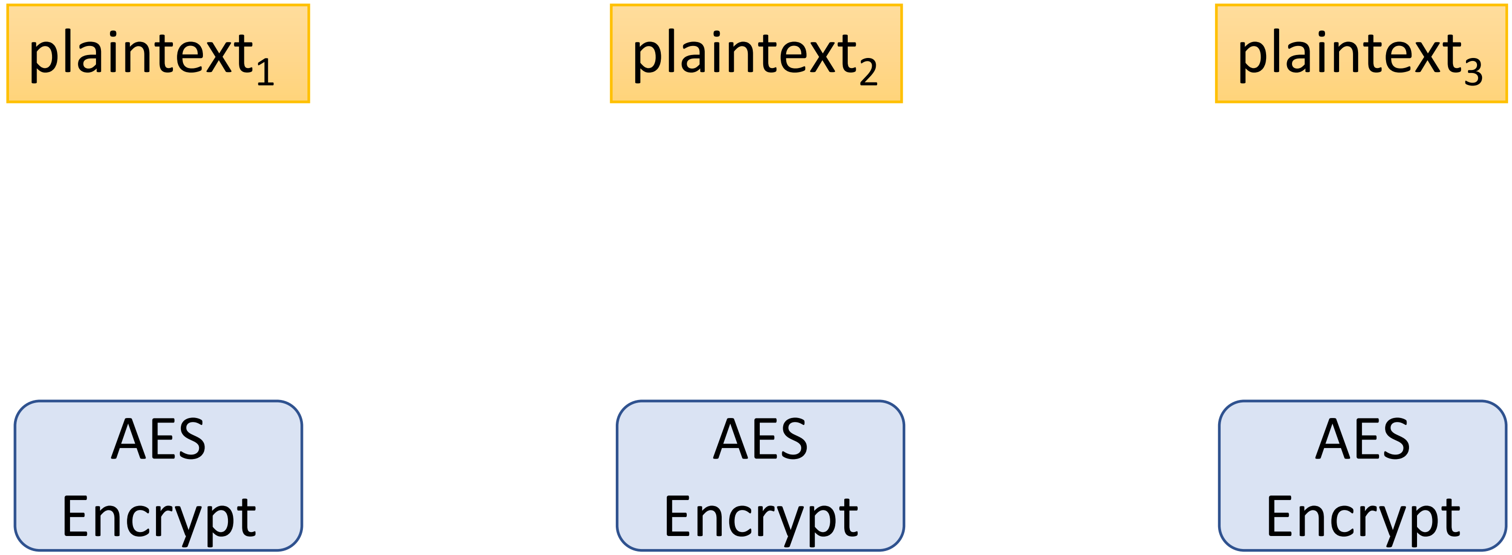
Verifying cryptography

- 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

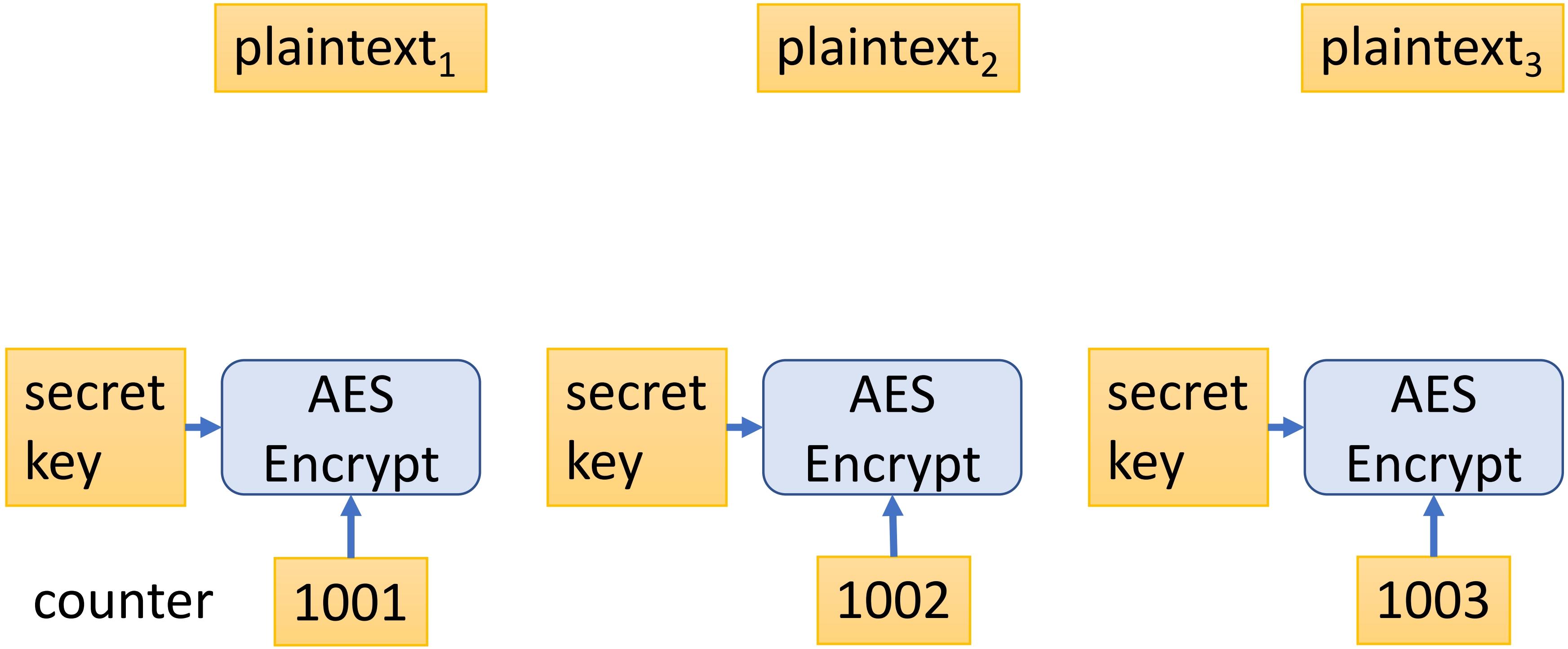
Example algorithm: AES-GCM

plaintext$_1$ plaintext$_2$ plaintext$_3$

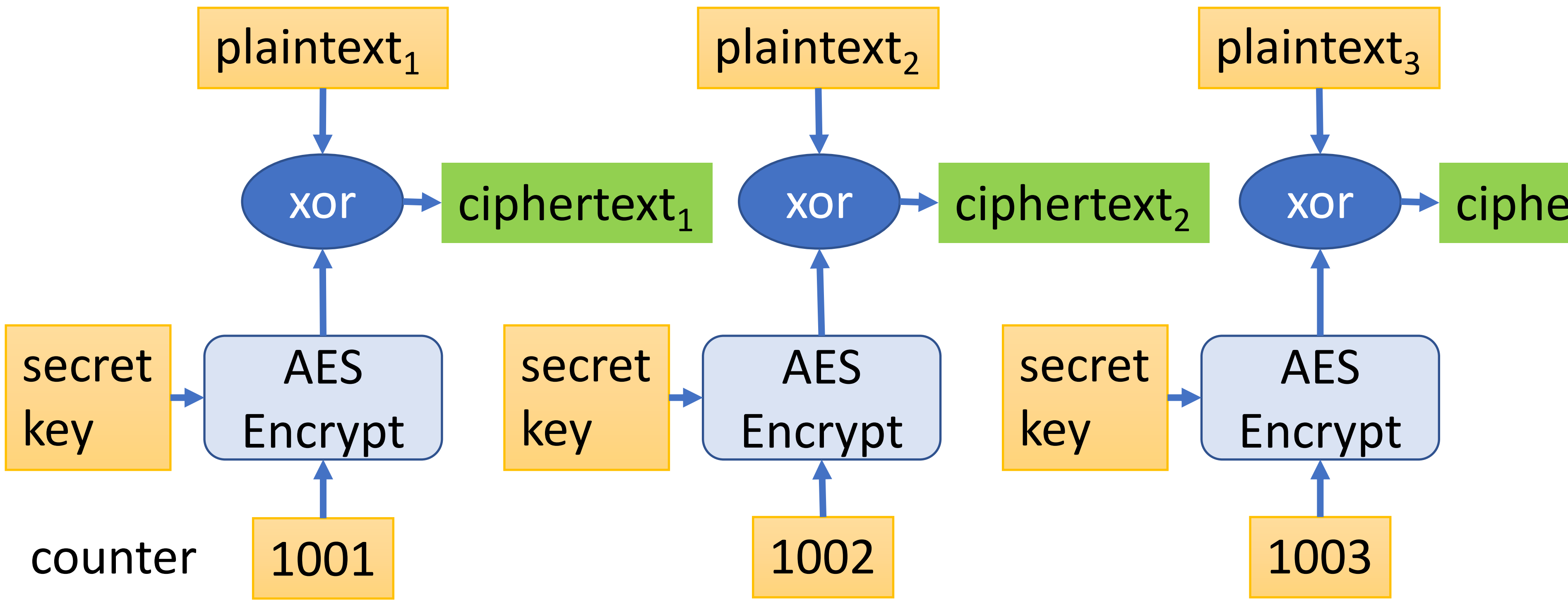
Example algorithm: AES-GCM



Example algorithm: AES-GCM



Example algorithm: AES-GCM



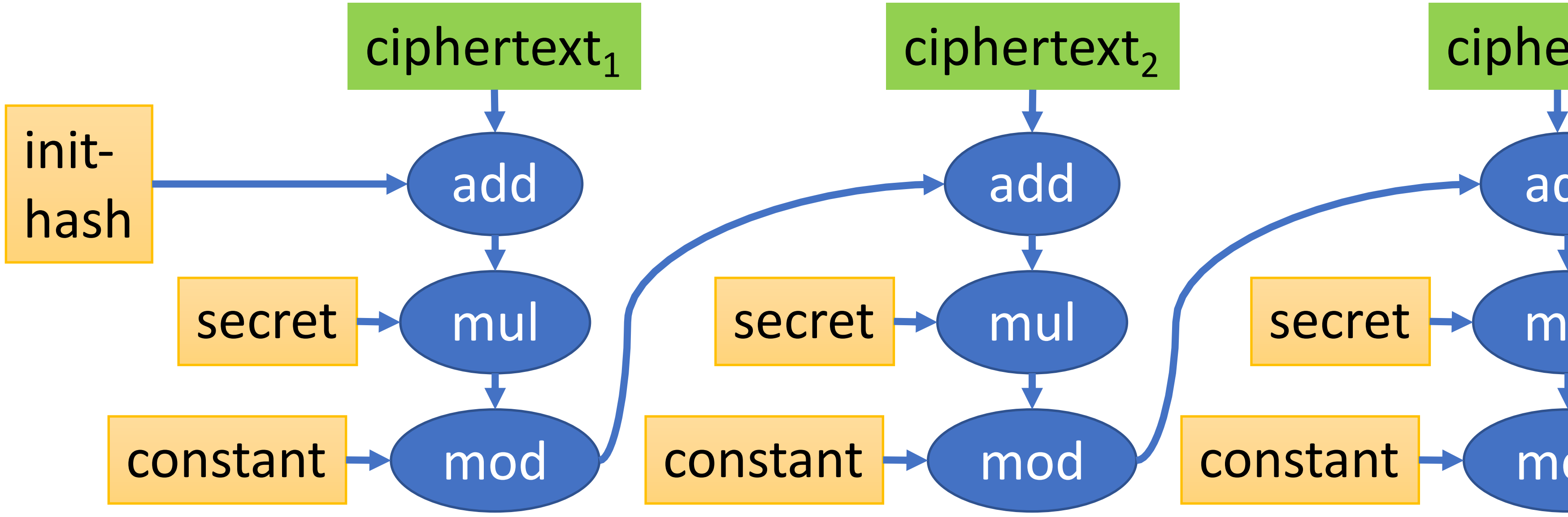
Example algorithm: AES-GCM

ciphertext$_1$

ciphertext$_2$

ciphe

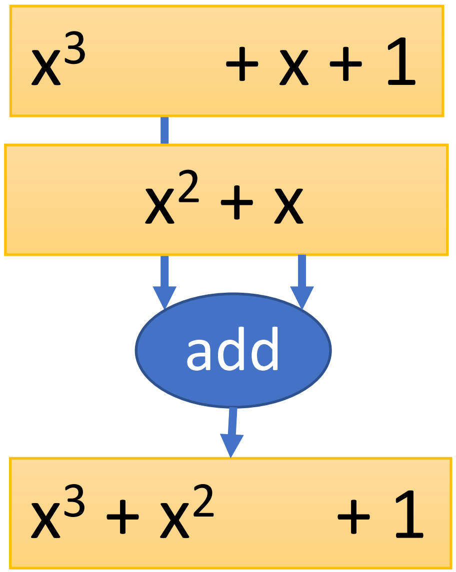
Example algorithm: AES-GCM



AES-GCM add, mul, mod

Operations on polynomials with base-2 coefficients

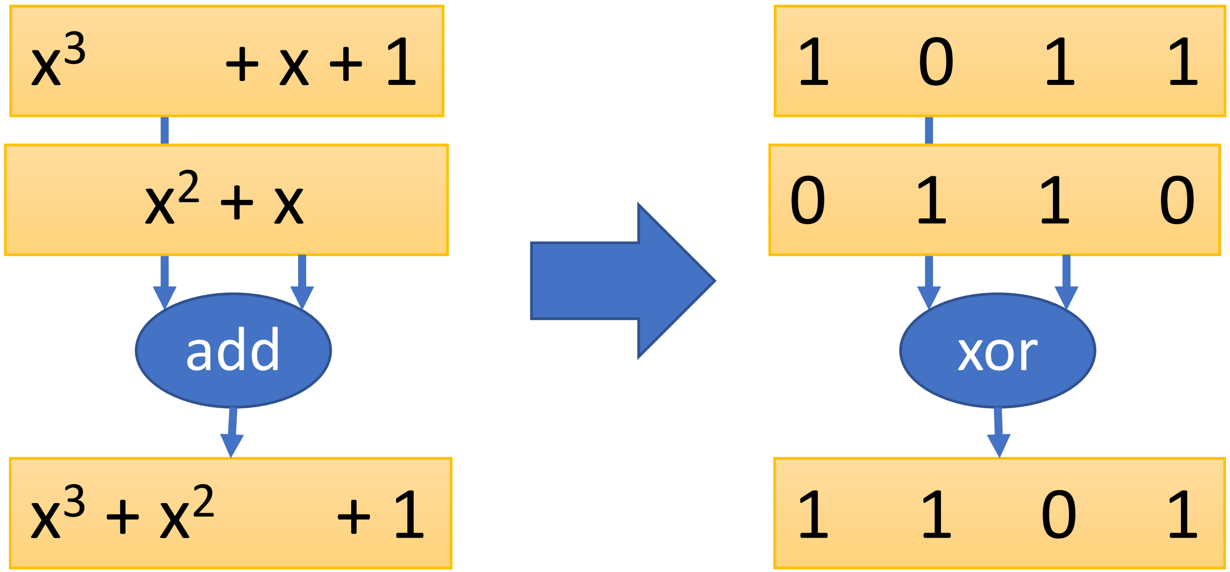
(we care only about the coefficients, not about x)



AES-GCM add, mul, mod

Operations on polynomials with base-2 coefficients

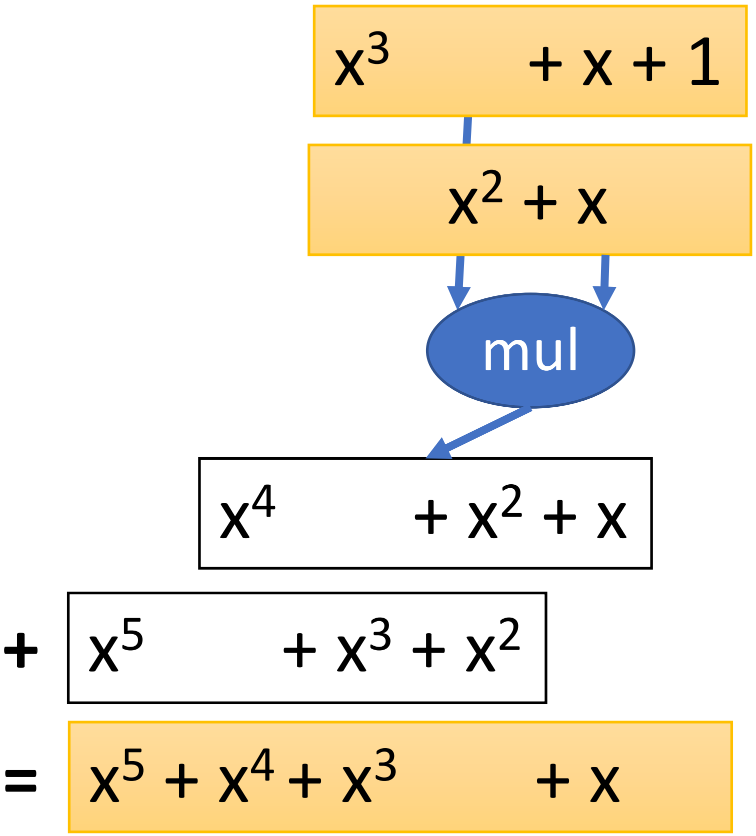
(we care only about the coefficients, not about x)



AES-GCM add, mul, mod

Operations on polynomials with base-2 coefficients

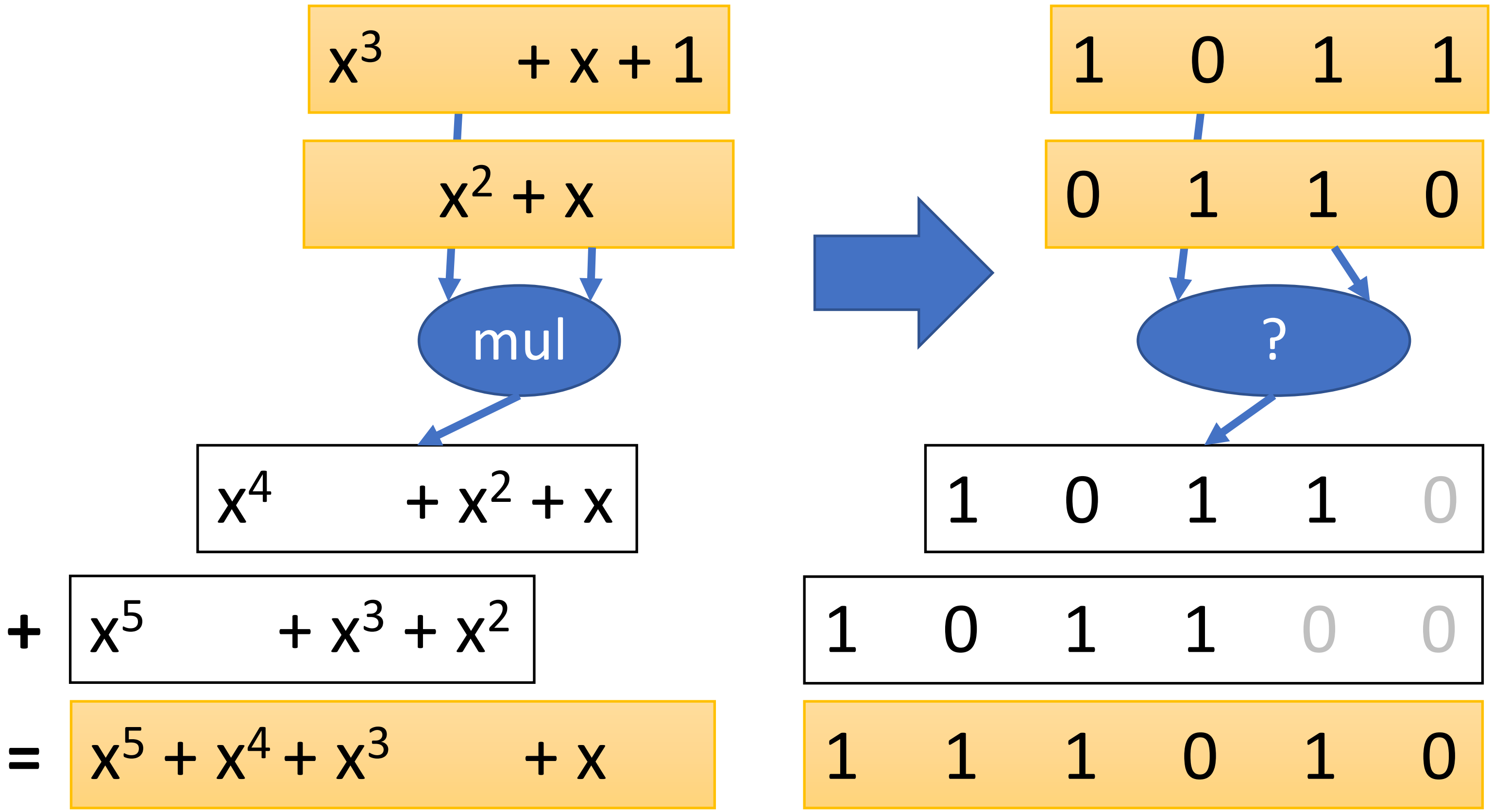
(we care only about the coefficients, not about x)



AES-GCM add, mul, mod

Operations on polynomials with base-2 coefficients

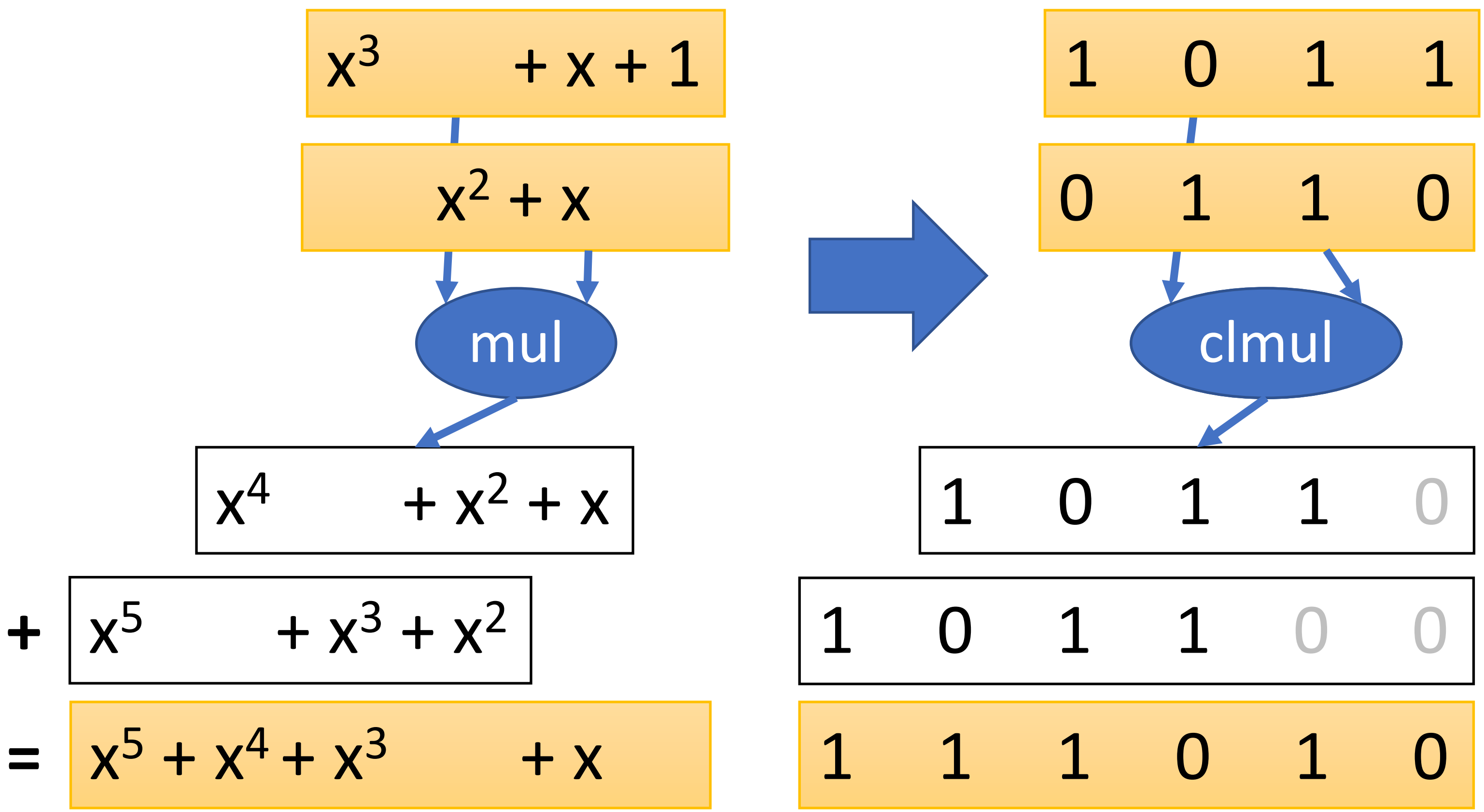
(we care only about the coefficients, not about x)



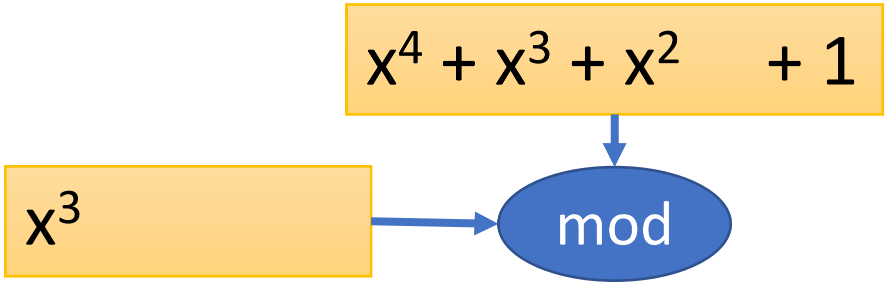
AES-GCM add, mul, mod

Operations on polynomials with base-2 coefficients

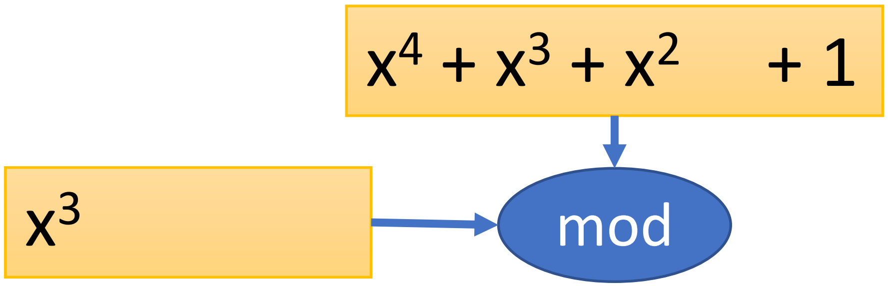
(we care only about the coefficients, not about x)



AES-GCM add, mul, mod



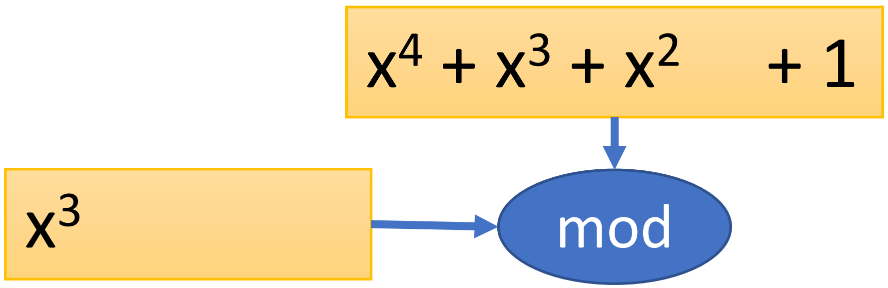
AES-GCM add, mul, mod



2437 mod 100

= 37

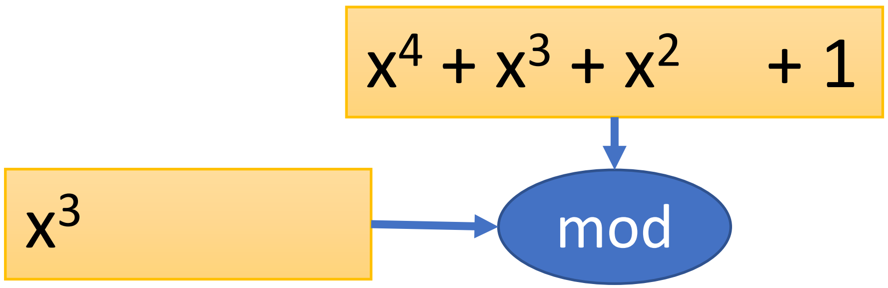
AES-GCM add, mul, mod



2437 mod 100
= 37

2437 =
100 * 24 + 37

AES-GCM add, mul, mod



2437 mod 100

= 37

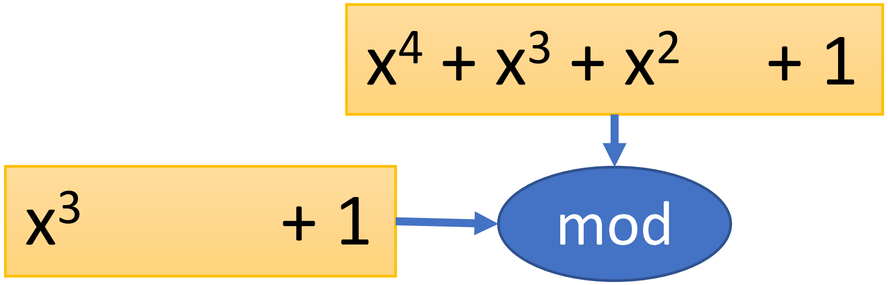
2437 =

100 * 24 + 37

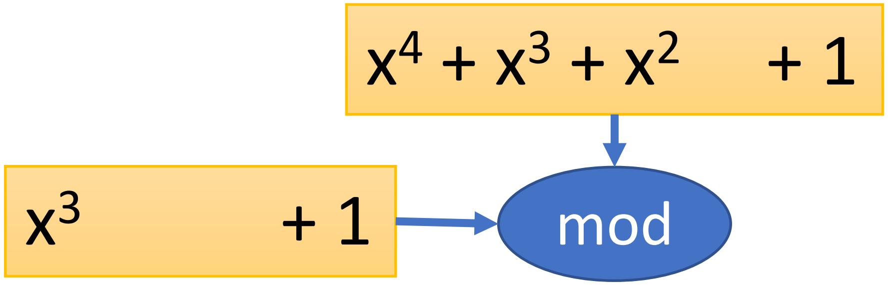
$(x^4 + x^3 + x^2 \quad + 1) \bmod (x^3)$

$= (x^2 \quad + 1)$

AES-GCM add, mul, mod



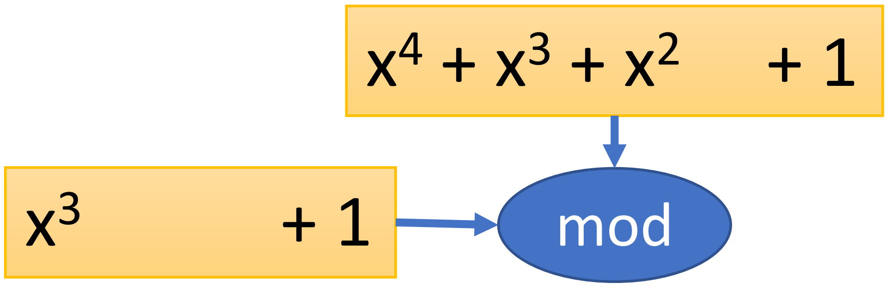
AES-GCM add, mul, mod



2437 mod 99

= 24 + 37 = 61

AES-GCM add, mul, mod



2437 mod 99

= 24 + 37 = 61

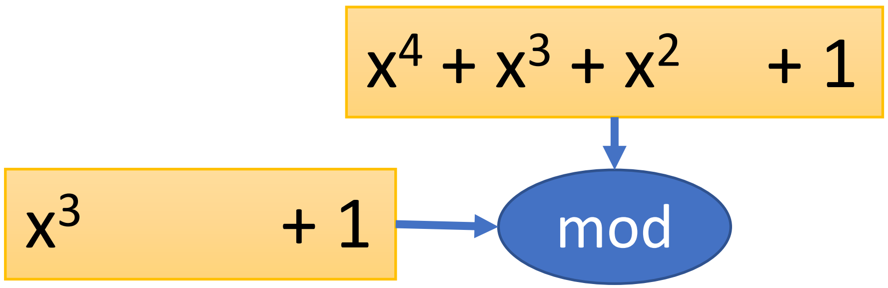
2437 =

100 * 24 + 37

(100 - 1) * 24 + 24 + 37

99 * 24 + 61

AES-GCM add, mul, mod



2437 mod 99

= 24 + 37 = 61

2437 =

100 * 24 + 37

(100 - 1) * 24 + 24 + 37

99 * 24 + 61

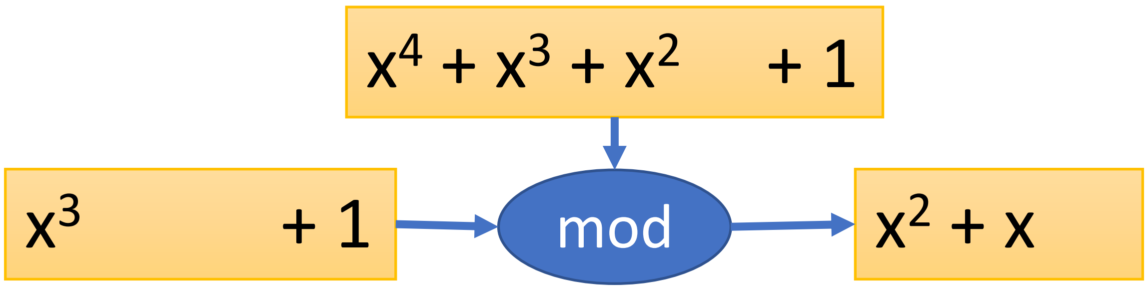
$(x^4 + x^3 + x^2 \quad + 1) =$

$(x^3) \quad * (x + 1) \qquad + (x^2 \quad + 1) =$

$(x^3 - 1) * (x + 1) + (x + 1) + (x^2 \quad + 1) =$

$(x^3 + 1) * (x + 1) + (x^2 + x \qquad)$

AES-GCM add, mul, mod



2437 mod 99

= 24 + 37 = 61

2437 =

$100 \quad * 24 \quad + 37$

$(100 - 1) * 24 + 24 + 37$

$99 \quad * 24 \quad + 61$

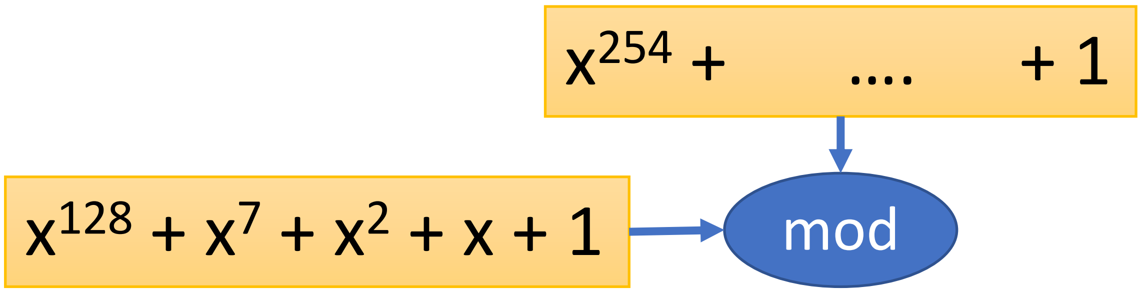
$(x^4 + x^3 + x^2 \quad + 1) =$

$(x^3) \quad * (x + 1) \quad + (x^2 \quad + 1) =$

$(x^3 - 1) * (x + 1) + (x + 1) + (x^2 \quad + 1) =$

$(x^3 + 1) * (x + 1) + (x^2 + x \quad)$

AES-GCM add, mul, mod



(demo: F* and Vale operations on polynomials)

Vale: extensible, automated assembly language verification

Vale: extensible, automated assembly language verification

machine model (Dafny/F*/Lean)

instructions

```
type reg = Rax | Rbx| ...
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: extensible, automated assembly language verification

machine model (Dafny/F*/Lean)

instructions

```
type reg = Rax | Rbx| …
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

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 = Rax | Rbx| …
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

Vale code

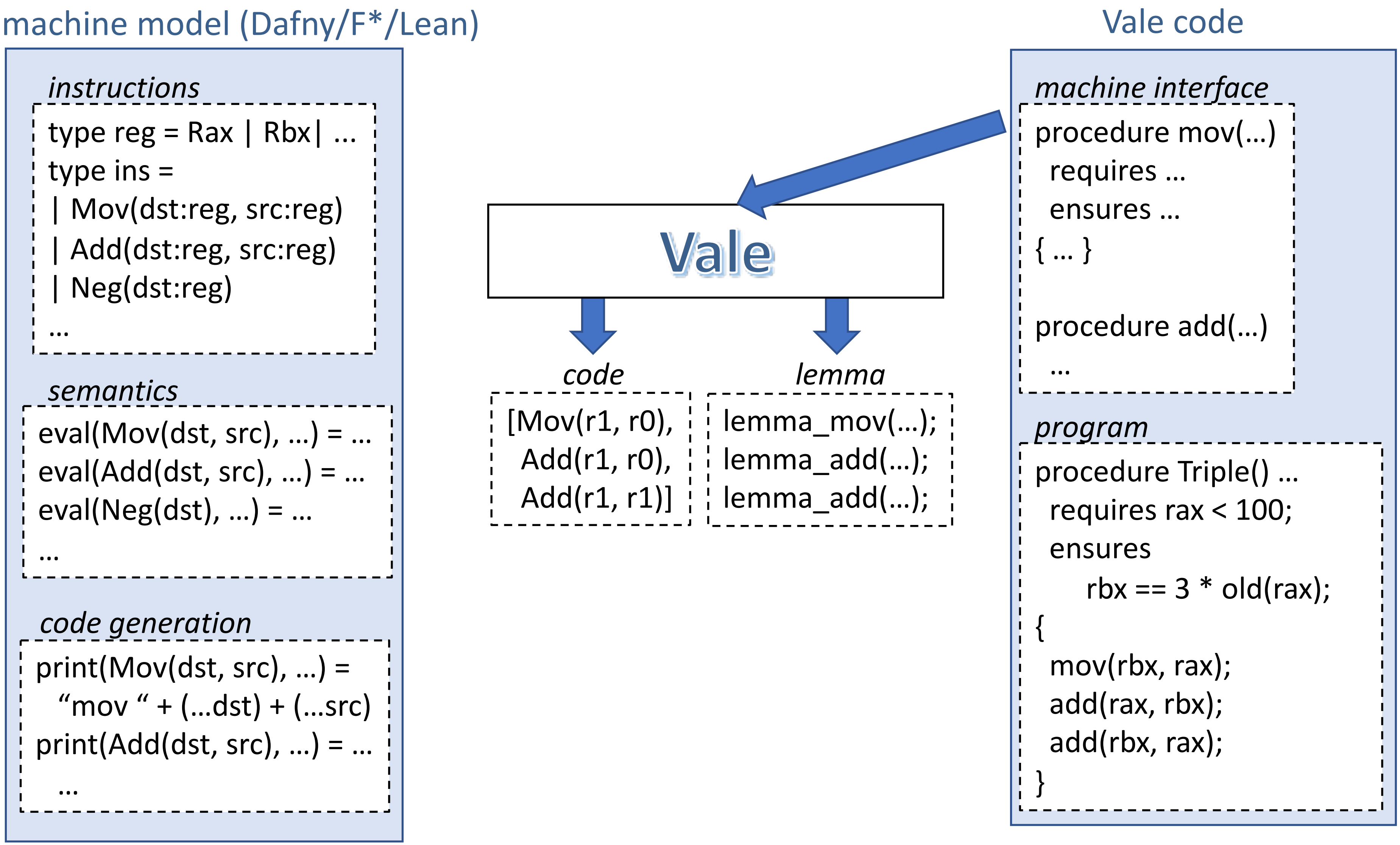
machine interface

```
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



Vale: extensible, automated assembly language verification

machine model (Dafny/F*/Lean)

instructions

```
type reg = Rax | Rbx| …
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(…);
```

Dafny

BOOGIE

Z3

THEOREM PROVER

Vale code

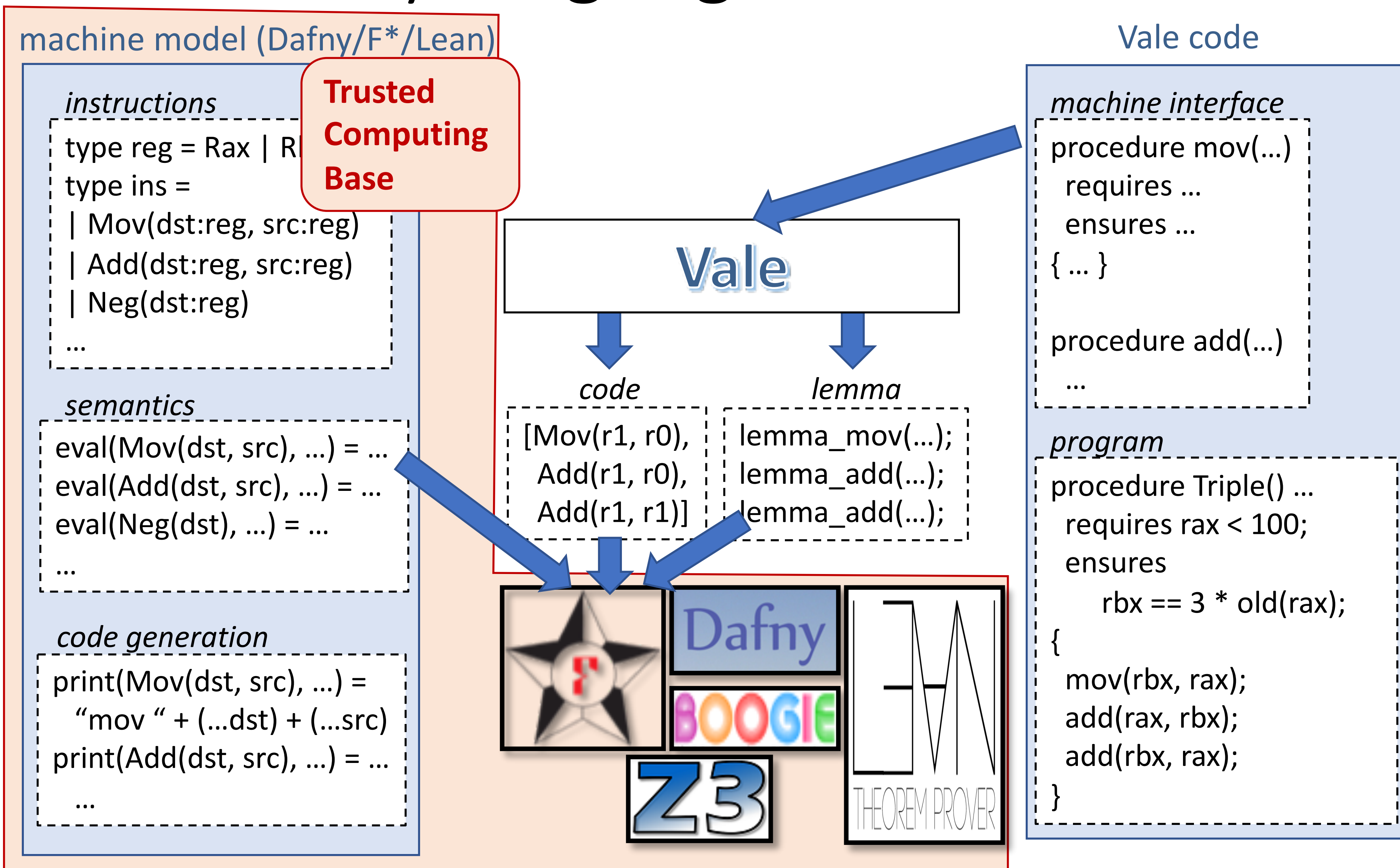
machine interface

```
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



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

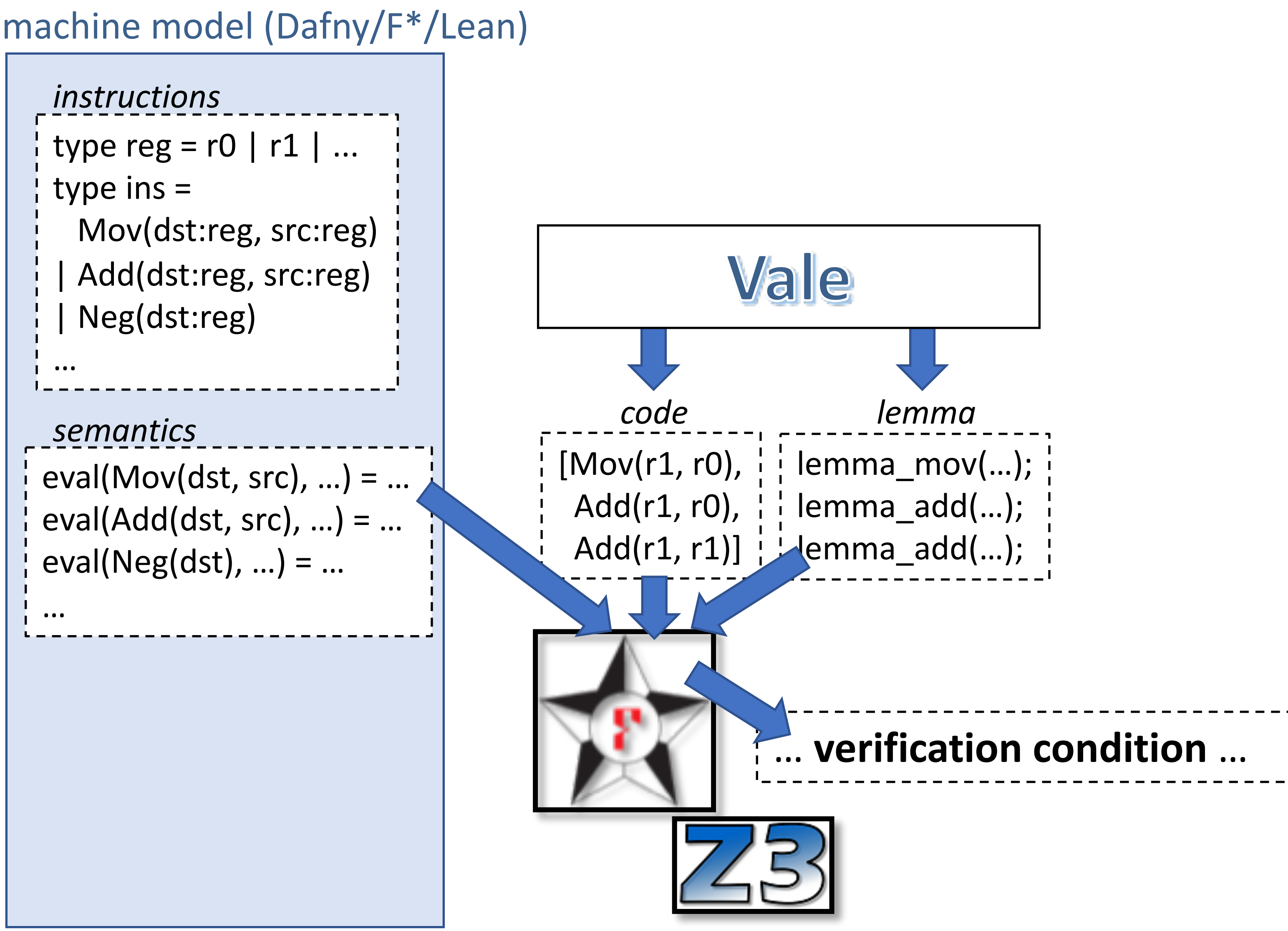
code

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

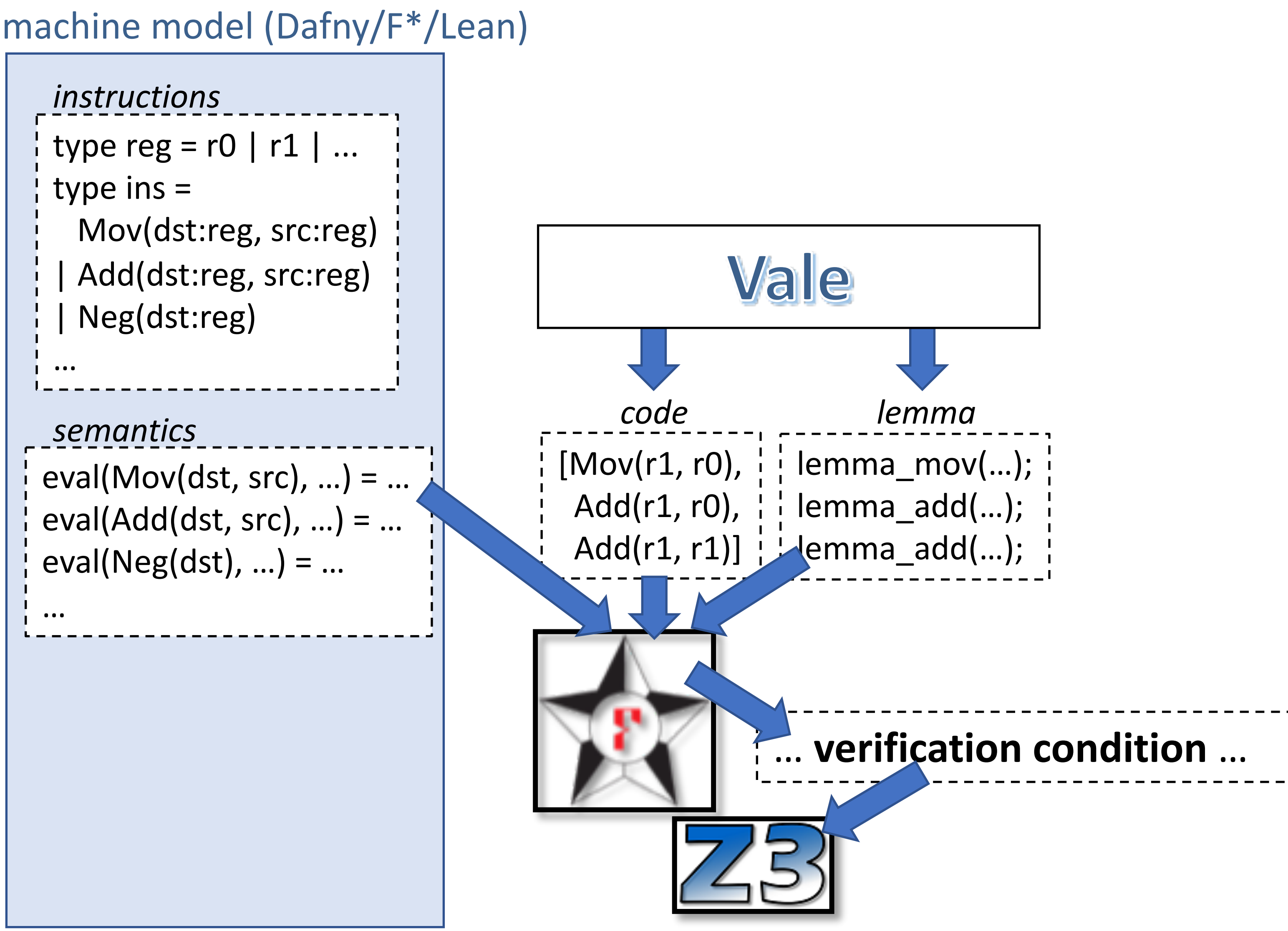
lemma

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

Vale: extensible, automated assembly language verification



Vale: extensible, automated assembly language verification



Verification condition

```
procedure Triple()
   requires rax < 100;
   ensures
      rbx == 3 * rax;
{
   Move(rbx, rax); // --> rbx1
   Add(rax, rbx);    // --> rax2
   Add(rbx, rax);    // --> rbx3
}
```

verification condition

$rax_0 < 100$

$\vert$-

$(rbx_1 == rax_0 ==>$

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

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

$rbx_3 == 3 * rax_0)))$

Verification condition

```
procedure Triple()
   requires rax < 100;
   ensures
      rbx == 3 * rax;
{
   Move(rbx, rax); // --> rbx1
   Add(rax, rbx);    // --> rax2
   Add(rbx, rax);    // --> rbx3
}
```

verification condition

$rax_0 < 100$

$|-$

$(rbx_1 == rax_0 ==>$

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

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

$rbx_3 == 3 * rax_0)))$

Verification condition

```
procedure Triple()
   requires rax < 100;
   ensures
      rbx == 3 * rax;
{
   Move(rbx, rax); // --> rbx1
   Add(rax, rbx);    // --> rax2
   Add(rbx, rax);    // --> rbx3
}
```

verification condition

$rax_0 < 100$

$|-$

$(rbx_1 == rax_0 ==>$

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

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

$rbx_3 == 3 * rax_0)))$

Verification condition

```
procedure Triple()
   requires rax < 100;
   ensures
      rbx == 3 * rax;
{
   Move(rbx, rax); // --> rbx1
   Add(rax, rbx);    // --> rax2
   Add(rbx, rax);    // --> rbx3
}
```

verification condition

$rax_0 < 100$

$|-$

$(rbx_1 == rax_0 ==>$

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

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

$rbx_3 == 3 * rax_0)))$

Verification condition

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

verification condition

$rax_0 < 100$

$|-$

$(rbx_1 == rax_0 ==>$

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

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

$rbx_3 == 3 * rax_0)))$

Verification condition

```
procedure Triple()
   requires rax < 100;
   ensures
      rbx == 3 * rax;
{
   Move(rbx, rax); // --> rbx1
   Add(rax, rbx);    // --> rax2
   Add(rbx, rax);    // --> rbx3
}
```

verification condition

$rax_0 < 100$

$\vert\text{-}$

$(rbx_1 == rax_0 ==>$

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

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

$rbx_3 == 3 * rax_0)))$

States, lemmas

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

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

States, lemmas

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

```
lemma_ ...);
lemma_add(...);
lemma_add(...);
```

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)

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!

Let's write our own VC generator!

- ??? Maybe like this: ???

procedure Triple() …
…

Our own Vale VC generator

I'm lonely and sad.

verification condition we want:

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

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

Z3

Let's write our own VC generator!

- ??? Maybe like this: ???

procedure Triple() …
…

Our own Vale VC generator

verification condition we want:

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

I'm lonely and sad.

- 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!

- ??? Maybe like this: ???

procedure Triple() …
…

Our own Vale VC generator

I'm lonely and sad.

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!

- ??? Maybe like this: ???

procedure Triple() ...
...

Our own Vale
VC generator

I'm lonely
and sad.

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?

Automated vs. expressive

"Automated reasoning systems typically fall in one of **two classes**: those that provide powerful **automation** for an impoverished logic, and others that feature **expressive logics** but only limited automation. PVS attempts to tread the middle ground between these two classes..."

- *PVS: A Prototype Verification System* (Shankar *et al*, 1992)

Automated vs. expressive

"Automated reasoning systems typically fall in one of **two classes**: those that provide powerful **automation** for an impoverished logic, and others that feature **expressive logics** but only limited automation. PVS attempts to tread the middle ground between these two classes..."

- *PVS: A Prototype Verification System* (Shankar *et al*, 1992)

Ironclad *(OSDI 2014)*

IronFleet *(SOSP 2015)*

Yggdrasil *(OSDI 2016)*

Komodo *(SOSP 2017)*

Hyperkernel *(SOSP 2017)*

Nickel *(OSDI 2018)*

seL4 *(SOSP 2009)*

Jitk *(OSDI 2014)*

FSCQ *(SOSP 2015)*

CertiKOS *(OSDI 2016)*

VDFS *(SOSP 2017)*

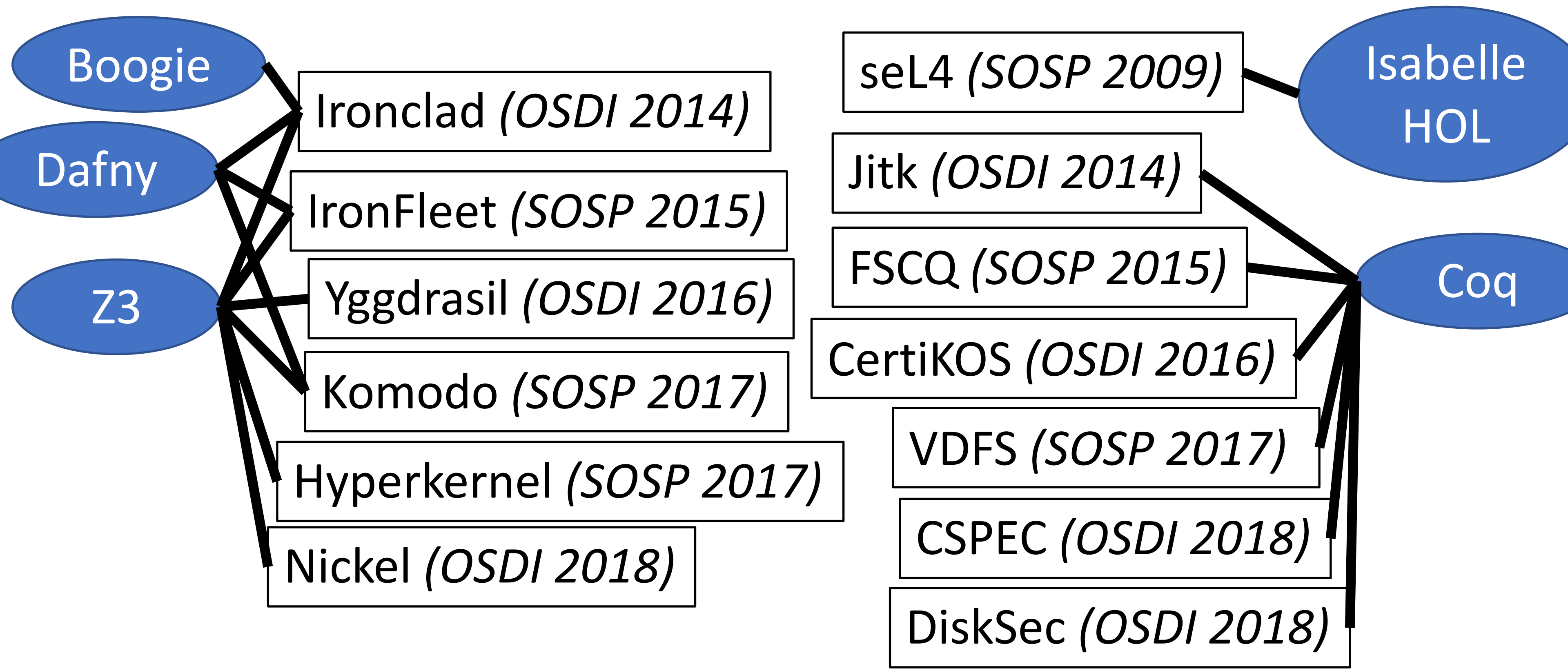
CSPEC *(OSDI 2018)*

DiskSec *(OSDI 2018)*

Automated vs. expressive

"Automated reasoning systems typically fall in one of **two classes**: those that provide powerful **automation** for an impoverished logic, and others that feature **expressive logics** but only limited automation. PVS attempts to tread the middle ground between these two classes..."

- *PVS: A Prototype Verification System* (Shankar *et al*, 1992)



Automated vs. expressive

Automated vs. expressive

```
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
```

Automated vs. expressive

```
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)

Automated vs. expressive

```
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
```

Automated vs. expressive

```
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:

$(0 <= x \wedge x < 10) ==>$
$(0 <= x \wedge x < 20)$

- 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) $\longrightarrow$ int

Let's write our own VC generator!

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:

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

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!

- 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!

- 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} ....

Let's write our own VC generator!

procedure Triple() ...
...

A big string?

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

verification condition we want:

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

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

Let's write our own VC generator!

procedure Triple() …
…

A b~~[illegible]~~ing?

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

verification condition we want:

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

Let's write our own VC generator!

procedure Triple() ...
...

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

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.

verification condition we want:

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

Let's write our own VC generator!

procedure Triple() ...
...

A ~~big string~~?

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.

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 big string?

Z3

Let's write our own VC generator!

```
procedure Triple() ...
...
```

~~A b 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.

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 ing?~~

Let's write our own VC generator!

procedure Triple() ...
...

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

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.

verification condition we want:

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

A b~~[illegible]~~ing?
A datatype?

Let's write our own VC generator!

procedure Triple() ...
...

A ~~b...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.

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...ing~~?
A ~~da...ype~~?

Let's write our own VC generator!

```
procedure Triple() ...
...
```

A ~~b ing~~?

A datatype:

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

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

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

verification condition we want:

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

A ~~b ing~~?

A ~~d 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*)

VC generator soundness (in F*)

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 normalize (vc_gen cs qcs k s0))
  (ensures (let sN = eval_code cs s0 in k sN))

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

VC generator soundness (in F*)

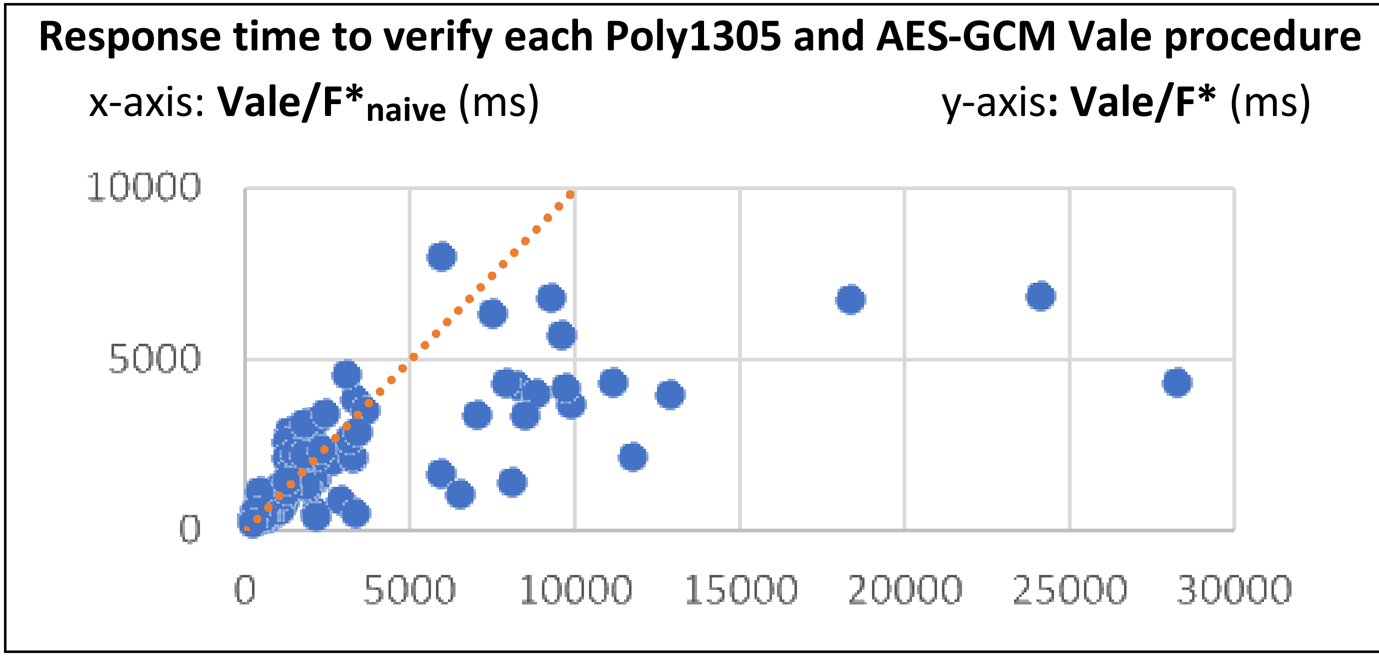
```
let rec vc_gen (cs:list code) (qcs:quickCodes cs) (k:state -> Type) : state -> Type =
  fun (s0:state) ->
    match qcs with
    | QEmpty -> k s0
    | QSeq qc qcs' -> qc.wp (vc_gen cs.tl qcs' k) s0
    | QLemma pre post lem qcs' -> pre /\ (post ==> vc_gen cs qcs' k s0)

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

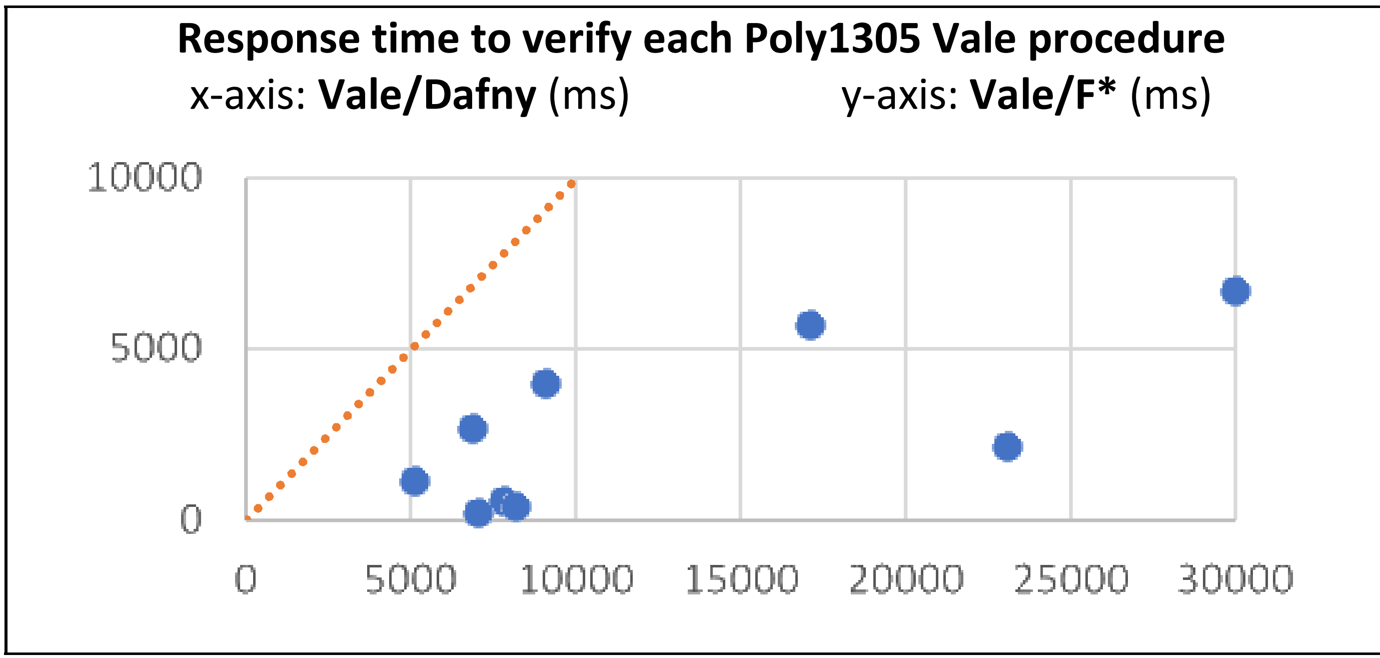
verification condition we want:

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

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

Verification performance





Conclusions

- We've verified fast assembly language crypto implementations:
  - SHA
  - Poly1305
  - AES-GCM
- F* is extensible via normalization, dependent types, …
  - We wrote our own domain-specific VC generator
  - We proved it sound
  - We run it from with F*'s type checker, and verification is fast

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