Verifying Assembly Language with Vale
Secure communication
*confidentiality, integrity, authentication*
Secure communication
confidentiality, integrity, authentication

SSL: Secure Sockets Layer
Secure communication
confidentiality, integrity, authentication

SSL: Secure Sockets Layer
TLS: Transport Layer Security
Secure communication

confidentiality, integrity, authentication

SSL: Secure Sockets Layer

TLS: Transport Layer Security
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

~100 pages
TLS standards, some implementations

**OpenSSL**

TLS Protocol: 40K LoC

Lines-of-Code measured with SLOCCount
# TLS standards, some implementations

**OpenSSL**

<table>
<thead>
<tr>
<th>TLS Protocol: 40K LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crypto</td>
</tr>
<tr>
<td>C: 160K LoC</td>
</tr>
</tbody>
</table>

~100 pages measured with SLOCCount
# TLS standards, some implementations

**OpenSSL**

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines of Code (LoC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS Protocol</td>
<td>40K LoC</td>
</tr>
<tr>
<td>Crypto C</td>
<td>160K LoC</td>
</tr>
<tr>
<td>Crypto Asm</td>
<td>150K LoC</td>
</tr>
</tbody>
</table>

~100 pages measured with SLOCCount

---

[ietf.org](https://ietf.org/rfc/rfc5246.txt)

The Transport Layer Security (TLS) Protocol

Version 1.2

Network Working Group

Internet-Draft

Obsoletes: 5077, 5246 (if approved)

Updates: 4492, 5705, 6066, 6961 (if approved)

Intended status: Standards Track

Expires: January 17, 2018

E. Rescorla

RTFM, Inc.

July 16, 2017

[ietf.org](https://ietf.org/rfc/rfc5246.txt)
## TLS standards, some implementations

<table>
<thead>
<tr>
<th>OpenSSL</th>
<th>BoringSSL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TLS Protocol</strong></td>
<td>40K LoC</td>
</tr>
<tr>
<td><strong>Crypto</strong></td>
<td></td>
</tr>
<tr>
<td>C: 160K LoC</td>
<td></td>
</tr>
<tr>
<td>Asm: 150K LoC</td>
<td></td>
</tr>
</tbody>
</table>

~100 pages

Lines-of-Code measured with SLOCCount

---

The Transport Layer Security (TLS) Protocol
Version 1.2

Network Working Group
Internet-Draft
Obsoletes: 5077, 5246 (if approved)
Updates: 4492, 5705, 6066, 6961 (if approved)
Intended status: Standards Track
Expires: January 17, 2018

E. Rescorla
RTFM, Inc.
July 16, 2017
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-bug

--------------------------------

Severity: High

TLS connections using *-cipher attack by corrupting large issue is not considered to a DoS

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

David Benjamin via RT rt.openssl.org

- 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: 2637408fe08086aad2f971e8425e2820
expected: 2637408fe08086ead2f971e8425e2820

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

David Benjamin via RT rt.openssl.org

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.openssl.org
Fri Mar 25 12:10:32 UTC 2016

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

plaintext_1

AES Encrypt

plaintext_2

AES Encrypt

plaintext_3

AES Encrypt
Example algorithm: AES-GCM

plaintext\textsubscript{1}    plaintext\textsubscript{2}    plaintext\textsubscript{3}

secret key    AES Encrypt    secret key    AES Encrypt    secret key    AES Encrypt

counter    1001    1002    1003
Example algorithm: AES-GCM

plaintext₁ → xor → ciphertext₁
plaintext₂ → xor → ciphertext₂
plaintext₃ → xor → ciphertext₃

secret key

AES Encrypt

counter

1001

1002

1003
Example algorithm: AES-GCM

ciphertext_1  ciphertext_2  ciphertext_3
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)*

\[ x^3 + x + 1 \]

\[ x^2 + x \]

add

\[ x^3 + x^2 + 1 \]
AES-GCM add, mul, mod

Operations on polynomials with base-2 coefficients
(*we care only about the coefficients, not about x*)

\[
\begin{align*}
x^3 &+ x + 1 \\
x^2 + x &
\end{align*}
\]

\[
\begin{align*}
x^3 + x^2 &+ 1 \\
1 &0 1 1 1 \\
0 &1 1 1 0 \\
1 &1 0 1 1
\end{align*}
\]
AES-GCM add, mul, mod

Operations on polynomials with base-2 coefficients
(we care only about the coefficients, not about x)

\[ x^3 + x + 1 \]

\[ x^2 + x \]

\[ x^4 + x^2 + x \]

\[ x^5 + x^3 + x^2 \]

\[ x^5 + x^4 + x^3 + x \]
AES-GCM add, mul, mod

Operations on polynomials with base-2 coefficients
*(we care only about the coefficients, not about x)*

\[
\begin{align*}
x^5 + x^3 + x^2 &+ x^3 + x^2 = x^5 + x^4 + x^3 + x \\
x^3 + x + 1 &+ x^2 + x = x^4 + x^2 + x \\
x^5 + x^4 + x^3 &+ x = 1 0 1 1 0 0 \\
\end{align*}
\]
AES-GCM add, mul, mod

Operations on polynomials with base-2 coefficients
(we care only about the coefficients, not about \(x\))

\[
\begin{align*}
x^3 &+ x + 1 \\
x^2 &+ x \\
x^4 &+ x^2 + x \\
x^5 &+ x^3 + x^2 \\
\end{align*}
\]

\[
\begin{align*}
1 &0 &1 &1 \\
0 &1 &1 &0 \\
1 &0 &1 &1 &1 &0 \\
1 &0 &1 &1 &0 &0 \\
1 &1 &1 &0 &1 &0 \\
\end{align*}
\]
AES-GCM add, mul, mod

\[ x^4 + x^3 + x^2 + 1 \]
AES-GCM add, mul, mod

\[ x^4 + x^3 + x^2 + 1 \]

\[ 2437 \mod 100 = 37 \]
AES-GCM add, mul, mod

\[ x^4 + x^3 + x^2 + 1 \]

\[ x^3 \quad \text{mod} \]

2437 mod 100 = 37

2437 = 100 * 24 + 37
AES-GCM add, mul, mod

\[ x^4 + x^3 + x^2 + 1 \]

\[ x^3 \mod \text{mod} \]

\[
2437 \mod 100 \\
= 37
\]

\[
2437 = 100 \times 24 + 37
\]

\[
(x^4 + x^3 + x^2 + 1) \mod (x^3) \\
= (x^2 + 1)
\]
AES-GCM add, mul, mod

\[ x^4 + x^3 + x^2 + 1 \]

\[ x^3 + 1 \]
AES-GCM add, mul, mod

$x^4 + x^3 + x^2 + 1$

$x^3 + 1 \mod\text{mod}$

$2437 \mod 99$

$= 24 + 37 = 61$
AES-GCM add, mul, mod

\[ x^4 + x^3 + x^2 + 1 \]

2437 mod 99
= 24 + 37 = 61

\[ 2437 = (100 - 1) \times 24 + 24 + 37 \]
\[ = 99 \times 24 + 61 \]
AES-GCM add, mul, mod

\[ x^4 + x^3 + x^2 + 1 \]

\[ x^3 + 1 \mod \]

2437 mod 99
= 24 + 37 = 61

\[ 2437 = 100 \cdot 24 + 37 \]
\[ = (100 - 1) \cdot 24 + 24 + 37 \]
\[ = 99 \cdot 24 + 61 \]

\[ (x^4 + x^3 + x^2 + 1) = \]
\[ (x^3) \cdot (x + 1) + (x^2 + 1) = \]
\[ (x^3 - 1) \cdot (x + 1) + (x + 1) + (x^2 + 1) = \]
\[ (x^3 + 1) \cdot (x + 1) + (x^2 + x ) \]
AES-GCM add, mul, mod

\[
x^4 + x^3 + x^2 + 1
\]

\[
2437 \mod 99 = 24 + 37 = 61
\]

\[
2437 = 100 \times 24 + 37
\]

\[
(100 - 1) \times 24 + 24 + 37
\]

\[
99 \times 24 + 61
\]

\[
(x^4 + x^3 + x^2 + 1) =
\]

\[
(x^3) \times (x + 1) + (x^2 + 1) =
\]

\[
(x^3 - 1) \times (x + 1) + (x + 1) + (x^2 + 1) =
\]

\[
(x^3 + 1) \times (x + 1) + (x^2 + x)
\]
AES-GCM add, mul, mod

\[ x^{254} + \ldots + 1 \]

\[ x^{128} + x^7 + x^2 + x + 1 \] 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

\[
\text{type reg} = \text{Rax} \mid \text{Rbx} \mid \ldots \\
\text{type ins} = \\ |
\quad \text{Mov}(\text{dst:reg}, \text{src:reg}) \\
\quad \text{Add}(\text{dst:reg}, \text{src:reg}) \\
\quad \text{Neg}(\text{dst:reg}) \\
\ldots
\]

semantics

\[
\text{eval(Mov(dst, src), ...)} = \ldots \\
\text{eval(Add(dst, src), ...)} = \ldots \\
\text{eval(Neg(dst), ...)} = \ldots \\
\ldots
\]

code generation

\[
\text{print(Mov(dst, src), ...)} = \\
\quad \text{“mov “ + (...dst) + (...src)} \\
\text{print(Add(dst, src), ...)} = \ldots \\
\ldots
\]

Vale code

machine interface

\[
\text{procedure mov(...)} \\
\quad \text{requires ...} \\
\quad \text{ensures ...} \\
\quad \{ ... \}
\]

\[
\text{procedure add(...)} \\
\quad \ldots
\]

program

\[
\text{procedure Triple()} ... \\
\quad \text{requires rax < 100;} \\
\quad \text{ensures} \\
\quad \quad \text{rbx == 3 * old(rax);} \\
\quad \quad \{ \\
\quad \quad \quad \text{mov(rbx, rax);} \\
\quad \quad \quad \text{add(rax, rbx);} \\
\quad \quad \quad \text{add(rbx, rax);} \\
\quad \quad \}
\]
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 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);
  }

**lemma**

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

**code**

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

- **procedure mov(...)**
  - requires ...
  - ensures ...
  - { ... }

- **procedure add(...)**
  - ...

---

**machine interface**

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

---

**Trusted Computing Base**

- Dafny
- BOOGIE
- Z3
- THEOREM PROVER

---

**program**

- lemma_mov(...);
- lemma_add(...);
- lemma_add(...);
Vale: extensible, automated assembly language verification

machine model (Dafny/F*/Lean)

instructions

\[
\text{type reg} = r0 \mid r1 \mid \ldots \\
\text{type ins} = \\
\quad \text{Mov}(\text{dst}:\text{reg}, \text{src}:\text{reg}) \\
\quad \text{Add}(\text{dst}:\text{reg}, \text{src}:\text{reg}) \\
\quad \text{Neg}(\text{dst}:\text{reg}) \\
\ldots
\]

semantics

\[
\text{eval}(\text{Mov}(\text{dst}, \text{src}), \ldots) = \ldots \\
\text{eval}(\text{Add}(\text{dst}, \text{src}), \ldots) = \ldots \\
\text{eval}(\text{Neg}(\text{dst}), \ldots) = \ldots \\
\ldots
\]
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 …
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 ...

Z3
procedure Triple()
    requires rax < 100;
    ensures
        rbx == 3 * rax;
{
    Move(rbx, rax);  // --&gt; rbx_1
    Add(rax, rbx);   // --&gt; rax_2
    Add(rbx, rax);   // --&gt; rbx_3
}
procedure Triple()
    requires rax < 100;
    ensures
        rbx == 3 * rax;
{
    Move(rbx, rax); // --> rbx1
1    Add(rax, rbx); // --> rax2
2    Add(rbx, rax); // --> rbx3
3
}
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} \land (rax_2 == rax_0 + rbx_1 ==> 
  rbx_1 + rax_2 < 2^{64} \land (rbx_3 == rbx_1 + rax_2 ==> 
  rbx_3 == 3 \cdot rax_0))))
procedure Triple()
    requires rax < 100;
    ensures rbx == 3 * rax;
{
    Move(rbx, rax); // --> rbx1
    Add(rax, rbx);  // --> rax2
    Add(rbx, rax); // --> rbx3
}
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} \land (rax_2 == rax_0 + rbx_1 ==> rbx_1 + rax_2 < 2^{64} \land (rbx_3 == rbx_1 + rax_2 ==> rbx_3 == 3 * rax_0))))
procedure Triple()
    requires rax < 100;
    ensures rbx == 3 * rax;
{
    Move(rbx, rax); // --> rbx_1
    Add(rax, rbx);  // --> rax_2
    Add(rbx, rax);  // --> rbx_3
}
States, lemmas

\[\text{Mov}(r1, r0), \text{Add}(r1, r0), \text{Add}(r1, r1)\]

lemma_mov(...);
lemma_add(...);
States, lemmas

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

\begin{align*}
\text{mov}(r1, r0), \\
\text{add}(r1, r0), \\
\text{add}(r1, r1)
\end{align*}
States, lemmas

\[ (\text{Mov}(r1, r0), \text{Add}(r1, r0), \text{Add}(r1, r1)) \]

**lemma_mov** (...);
**lemma_add** (...);
**lemma_add** (...);

**requires** ...
\[ \begin{align*}
  & s1.ok /\ \\ \\
  & \text{valid_operand} \ s1 \ \text{dst} /\ \\ \\
  & \text{valid_operand} \ s1 \ \text{src} /\ \\ \\
  & ( \text{eval_operand} \ s1 \ \text{dst} \\
  & + \text{eval_operand} \ s1 \ \text{src} ) < 2^{64}
\end{align*} \]

**ensures** ...
\[ \begin{align*}
  & s2.ok /\ \ \\
  & s2 == (...)\text{framing...} \ s1 /\ \\
  & \text{eval_operand} \ s2 \ \text{dst} == \\
  & ( \text{eval_operand} \ s1 \ \text{dst}, \\
  & + \text{eval_operand} \ s1 \ \text{src})
\end{align*} \]

**type** state = {
  ok:bool;
  regs:regs;
  flags:nat64;
  mem:mem;
}

s1 : state
s2 : state

**s1** : state
**s2** : state

\}
Ugh! Default SMT query looks awful!

**verification condition we want:**

\[
\begin{align*}
\text{verification condition we get:} & \\
& \text{(forall (ghost_result_0:(state * fuel))).} \\
& \text{(let (s3, fc3) = ghost_result_0 in} \\
& \quad \text{eval_code (Ins (Add64 (OReg (Rax)) (OReg (Rbx)))) fc3 s2 == Some s3} \\
& \quad \text{eval_operand (OReg Rax) s3 == eval_operand (OReg Rax) s2} \\
& \quad \text{lemma_Add s2 (OReg Rax) (OReg Rbx) == ghost_result_0} \\
& \quad \text{(forall (s3:state) (fc3:fuel). lemma_Add s2 (OReg Rax) (OReg Rbx) == Mktuple2 s3 fc3} \\
& \quad \text{Cons? codes_Triple.tl} \\
& \quad \text{(forall (any_result0:list code). codes_Triple.tl == any_result0} \\
& \quad \text{(forall (any_result1:list code). codes_Triple.tl.tl == any_result1} \\
& \quad \text{OReg? (OReg Rbx)} \\
& \quad \text{eval_operand (OReg Rbx) s3} \\
& \text{< 2^{64}}
\end{align*}
\]
Let's write our own VC generator!
Let's write our own VC generator!

• ??? Maybe like this: ???

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.

procedure Triple() …

procedure Triple() …
Let's write our own VC generator!

• ??? Maybe like this: ???

procedure Triple() ...
...

Our own Vale
VC generator

verification condition we want:
........................................ (rax₂ == rax₀ + rbx₁ =>
rbx₁ + rax₂ < 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: ???

  Our own Vale VC generator

  verification condition we want:
  
  \[
  \begin{align*}
  & (rax_2 == rax_0 + rbx_1) \\
  & rbx_1 + rax_2 < 2^{64}
  \end{align*}
  \]

  procedure Triple() ...

  ...

  \[\text{I'm lonely and sad.}\]

- 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: ???

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

**verification condition we want:**
```
............... (rax_2 == rax_0 + rbx_1 ==> rbx_1 + rax_2 < 2^{64} )
```

Our own Vale VC generator

procedure Triple() ...
...

I'm lonely and sad.
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)
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 \land x < 10\}
type nat20 = x:int\{0 <= x \land 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 \land x < 10\}
type nat20 = x:int\{0 <= x \land x < 20\}

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

• 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

ask Z3 to prove:
(0 <= x \land x < 10) \implies (0 <= x \land x < 20)
Automated vs. expressive

- 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) \rightarrow int
Let's write our own VC generator!
Let's write our own VC generator!

• Like this!

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

procedure Triple() ...
...

verification condition we want:

\[ (rax_2 = rax_0 + rbx_1 \Rightarrow rbx_1 + rax_2 < 2^{64}) \]

I'm happy.
I have super powers.
Let's write our own VC generator!

• Like this!

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

verification condition we want:

\[
\begin{align*}
&\text{("\(\text{rax}_2 = \text{rax}_0 + \text{rbx}_1\) \implies \text{rbx}_1 + \text{rax}_2 < 2^{64}\)}\end{align*}
\]

• 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!

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

verification condition we want:

\[ (rax_2 == rax_0 + rbx_1 \Rightarrow rbx_1 + rax_2 < 2^{64}) \]

- 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!

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

procedure Triple() …

• 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:
...........................(rax₂ == rax₀ + rbx₁ ==> 
rbx₁ + rax₂ < 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:

\[ (rax_2 == rax_0 + rbx_1) \Rightarrow rbx_1 + rax_2 < 2^{64} \]
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:

.........................(rax₂ == rax₀ + rbx₁ =>
rbx₁ + rax₂ < 2^{64} .......................
Let's write our own VC generator!

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 datatype:

\[
\text{type quickCode} = \ldots \\
\text{type quickCodes} = \\
| QEmpty \\
| QSeq of quickCode * quickCodes \ldots \\
| QLemma of ... (Lemma pre post) \ldots
\]

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

Z3
Let's write our own VC generator!

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

procedure Triple() ... ...

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 big string?

verification condition we want:

...............(rax₂ == rax₀ + rbx₁ ==> rbx₁ + rax₂ < 2^{64} ......................

Z3
Let's write our own VC generator!

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

**verification condition we want:**

\[(rax_2 = rax_0 + rbx_1 \Rightarrow rbx_1 + rax_2 < 2^{64})\]

A big string?

**A datatype:**

```plaintext
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 big string?

**Z3**
Let's write our own VC generator!

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

verification condition we want:
........................................
...............(rax₂ == rax₀ + rbx₁ ==> rbx₁ + rax₂ < 2⁶⁴ ......................

procedure Triple() ...
...

A blocking?
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 blocking?
A datatype?

Z3
Let's write our own VC generator!

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

 verification condition we want:  
  \[ (\text{rax}_2 = \text{rax}_0 + \text{rbx}_1 \Rightarrow \text{rbx}_1 + \text{rax}_2 < 2^{64} ) \]

A boolean?  
A datatype:  
\[
\text{type quickCode} = \ldots  
\text{type quickCodes} =  
| \text{QEmpty}  
| \text{QSeq of quickCode * quickCodes} \ldots  
| \text{QLemma of ... (Lemma pre post) * ...}
\]

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

A boolean?  
A datatype?
Let's write our own VC generator!

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

verification condition we want:

...............(rax₂ == rax₀ + rbx₁ ==> rbx₁ + rax₂ < 2^{64} .................

A blocking? 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.

An F* term:

(forall rbx₁. rbx₁ == rax₀ ==> rax₀ + rbx₁ < 2^{64} /
(forall rax₂. rax₂ == rax₀+ rbx₁ ==> rbx₁ + rax₂ < 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);
}
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 ...
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

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

- x-axis: `Vale/F*_{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)`
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/