Cryptographic Primitives Implemented Efficiently, Correctly and Securely

Nik Swamy, OPLSS 2021

Thanks to Jonathan Protzenko and Chris Hawblitzel for these slides. Errors are mine
Crypto code is hard to get right

But is critical for system security
Many bugs in Curve25519 implementations (C and assembly)

Correct bounds in 32-bit code.
The 32-bit code was illustrative of the tricks used in the original curve25519 paper rather than rigorous. However, it has proven quite popular.

This change fixes an issue that Robert Ransom found where outputs between $2^{255}-19$ and $2^{255}-1$ weren't correctly reduced in fcontract. This appears to leak a small fraction of a bit of security of private keys.

Additionally, the code has been cleaned up to reflect the real-world needs. The ref10 code also exists for 32-bit, generic C but is somewhat slower and objections around the lack of qasm availability have been raised.

This bug is triggered when the last limb $n[15]$ of the input argument $n$ of this function is greater or equal than $0xffff$. In these cases the result of the scalar multiplication is not reduced as expected resulting in a wrong packed value. This code can be fixed simply by replacing $m[15]=0xffff$ by $m[15]=0xffff$.

While visiting 30c3, I attended the You-broke-the-internet workshop on NaCl. One thing mentioned in the talk was that auditing crypto code is a lot of work, and that this is one of the reasons why Ed25519 isn't included in NaCl yet (they promised a version including it for 2014). The speakers mentioned a bug in the amd64 assembly implementation of Ed25519 as an example of a bug that can only be found by auditing, not by randomized tests. This bug is caused by a carry being added in the wrong place, but since that carry is usually zero, the bug is hard to find (occurs with probability $2^{-60}$ or so).

The TweetNaCl paper briefly mentions this bug as well:

Partial audits have revealed a bug in this software ($r_1 \leftarrow 0 \land carry \text{ should be } r_2 \leftarrow 0 \land carry$ in amd64-64-24k) that would not be caught by random tests; this illustrates the importance of audits.

Searching for this string in the SUPERCOP source code turns up four matches:

This bug is triggered when the last limb $n[15]$ of the input argument $n$ of this function is greater or equal than $0xffff$. In these cases the result of the scalar multiplication is not reduced as expected resulting in a wrong packed value. This code can be fixed simply by replacing $m[15]=0xffff$ by $m[15]=0xffff$.
3 Bugs in OpenSSL implementation of Poly1305

OpenSSL Security Advisory [10 Nov 2016]

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

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

I'm probably going to write something to generate random inputs and stress all your other poly1305 code paths against a reference implementation.
Implementation bug in AES-GCM

The fragility of AES-GCM authentication algorithm

Shay Gueron\textsuperscript{1,2}, Vlad Krasnov\textsuperscript{2}

\textsuperscript{1} Department of Mathematics, University of Haifa, Israel
\textsuperscript{2} Intel Corporation, Israel Development Center, Haifa, Israel

March 15, 2013

Abstract. A new implementation of the GHASH function has been recently committed to a Git version of OpenSSL, to speed up AES-GCM. We identified a bug in that implementation, and made sure it was quickly fixed before trickling into an official OpenSSL trunk. Here, we use this (already fixed) bug
AES-GCM

Evaluate polynomials in this field to get an authentication code! (see also: Poly1305)

GHASH (AES-GCM):
- \( p = 2^{128} (q = 2, n = 128) \)
- \( P = x^{128} + x^7 + x^2 + x + 1 \)

“the math”
Distilling the math for implementors

“the algorithm”
Writing the actual code

A long way from the math

“the reality”
\[ GF(2^{128}) = GF(2)[X]/(x^{128} + x^7 + x^2 + x + 1) \]
Specification ("the mathematical truth")

Pseudo-code ("implementation blueprint")

Vale/F* ("assembly-like")

via Vale printer

Assembly (.asm)

proof

Vale

via KreMLin compiler

proof

Low* ("C-like")

proof

C code (.c, .h)

proof
What do we verify?

Safety
Memory- and type-safety. Mitigates buffer overruns, dangling pointers, code injections. No undefined behavior.

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.

Our specifications and implementations are written together, in one language (F*)
Drift between spec and implementation cannot happen.

Each application can do custom proofs beyond functional correctness and safety:
- non malleability (parsers)
- crypto games (TLS)
- security reduction (Merkle Trees)
- etc. etc.
Verified Assembly Language in Vale / F*
We have a **fast** verified AES-GCM

Performance of various verified symmetric crypto / hash implementations

- **Ironclad Apps SHA256**
- **Andrew Appel SHA256**
- **Vale AES-CBC+Poly1305**
- **HACL* ChaChaPoly**
- **Jasmin ChaCha20 + Poly1305**
- **Vale AES-GCM-128**

Fastest OpenSSL assembly code
Important optimizations:
- delay mod operations
- parallelize add/mul operations
- math+bitwise tricks for mod
- careful instruction scheduling

\[((\text{init} + c_1) \times s \mod P + c_2) \times s \mod P + c_3) \times s \mod P\]

\rightarrow ((\text{init} + c_1) \times s + c_2) \times s + c_3) \times s \mod P\]

\rightarrow ((\text{init} + c_1) \times s^3 + c_2 \times s^2 + c_3 \times s^1) \mod P\]

\rightarrow ((\text{init} + c_1) \times (s^3 \mod P) + c_2 \times (s^2 \mod P) + c_3 \times s^1) \mod P\]
Vale: extensible, automated assembly language verification

machine model (F*)

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), ...) = ...
- ...

Trusted Computing Base

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);
  - }

Z3
Vale: extensible, automated assembly language verification

machine model (F*)

instructions

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

semantics

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

Vale

code

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

lemma

\[
\text{lemma_mov(...);} \\
\text{lemma_add(...);} \\
\text{lemma_add(...);} \\
\text{... verification condition ...}
\]

Z3
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} \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))))
Demo

• Verification condition generation for Vale
Ugh! Default SMT query looks awful!

**verification condition we want:**

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

**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: ???

Our own Vale VC generator

verification condition we want:

procedure Triple() …

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!

• Like this!

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

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

• 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:

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

A big string?
A datatype:

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

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

A big string?
A datatype?
An F* term:

\[
(\forall \text{rbx}_1. \text{rbx}_1 = \text{rax}_0 \Rightarrow \text{rax}_0 + \text{rbx}_1 < 2^{64} \land \\
(\forall \text{rax}_2. \text{rax}_2 = \text{rax}_0 + \text{rbx}_1 \Rightarrow \text{rbx}_1 + \text{rax}_2 < 2^{64} \land \ldots)
\]

Z3