HACL*
High-Assurance Cryptographic Library

Marina Polubelova

PROSECCO, INRIA Paris

VeriCrypt @ IndoCrypt
December 11, 2020

HACL*: a verified C crypto library

- A growing library of verified crypto algorithms
  - AEAD: Chacha20-Poly1305, AES-GCM
  - Hashes: SHA2, SHA3, Blake2
  - HMAC and HKDF
  - ECC: Curve25519, Ed25519, P256
  - High-level APIs: Box and HPKE

    coming soon: generic bignum library, RSAPSS, FFDHE
- Developed as a collaboration between the **Prosecco** team at **INRIA Paris**, **Microsoft Research**, and **Carnegie Mellon University**

HACL*: a verified C crypto library

- Implemented and verified in F* and compiled to C
  - **Memory safety** proved in the C memory model
  - **Secret independence** ("constant-time") enforced by typing
  - **Functional correctness** against a mathematical spec written in F*
- Generates readable, portable, standalone C code
  - Performance comparable to hand-written C crypto libraries
  - Used in Mozilla Firefox, WireGuard VPN, Tezos Blockchain, etc

Writing Verified Crypto Code

CRYPTO STANDARD
(IETF/NIST)

Obsoleted by: 8439 INFORMATIONAL
Errata Exist

Internet Research Task Force (IRTF)	Y. Nir
Request for Comments: 7539	Check Point
Category: Informational	A. Langley
ISSN: 2070-1721	Google, Inc.
	May 2015

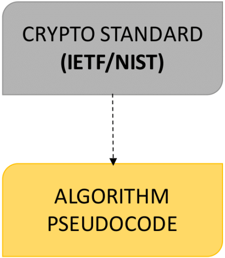
ChaCha20 and Poly1305 for IETF Protocols

Abstract

This document defines the ChaCha20 stream cipher as well as the use of the Poly1305 authenticator, both as stand-alone algorithms and as a "combined mode", or Authenticated Encryption with Associated Data (AEAD) algorithm.

This document does not introduce any new crypto, but is meant to serve as a stable reference and an implementation guide. It is a product of the Crypto Forum Research Group (CFRG).

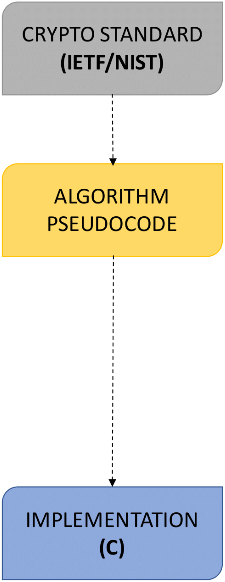
Writing Verified Crypto Code



2.5.1. The Poly1305 Algorithms in Pseudocode

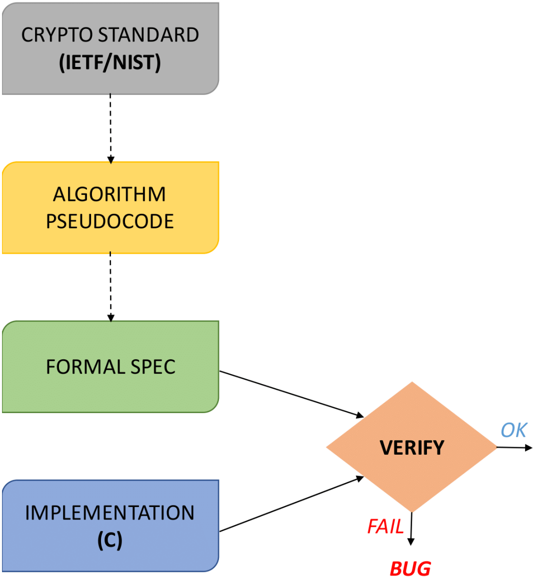
```
clamp(r): r &= 0x0ffffffc0ffffffc0ffffffc0fffffff
poly1305_mac(msg, key):
   r = (le_bytes_to_num(key[0..15])
   clamp(r)
   s = le_num(key[16..31])
   accumulator = 0
   p = (1<<130)-5
   for i=1 upto ceil(msg length in bytes / 16)
      n = le_bytes_to_num(msg[((i-1)*16)..(i*16)] | [0x01])
      a += n
      a = (r * a) % p
      end
   a += s
   return num_to_16_le_bytes(a)
   end
```

Writing Verified Crypto Code




```
void Poly1305_Update(POLY1305 *ctx, const unsigned char *inp, size_t len)
{
#ifdef POLY1305_ASM
    /*
     * As documented, poly1305_blocks is never called with input
     * longer than single block and padbit argument set to 0. This
     * property is fluently used in assembly modules to optimize
     * padbit handling on loop boundary.
     */
    poly1305_blocks_f poly1305_blocks_p = ctx->func.blocks;
#endif
    size_t rem, num;

    if ((num = ctx->num)) {
        rem = POLY1305_BLOCK_SIZE - num;
        if (len >= rem) {
            memcpy(ctx->data + num, inp, rem);
            poly1305_blocks(ctx->opaque, ctx->data, POLY1305_BLOCK_SIZE, 1);
            inp += rem;
            len -= rem;
        } else {
            /* Still not enough data to process a block. */
            memcpy(ctx->data + num, inp, len);
            ctx->num = num + len;
            return;
        }
    }

    rem = len % POLY1305_BLOCK_SIZE;
    len -= rem;

    if (len >= POLY1305_BLOCK_SIZE) {
        poly1305_blocks(ctx->opaque, inp, len, 1);
        inp += len;
    }

    if (rem)
        memcpy(ctx->data, inp, rem);

    ctx->num = rem;
}
```

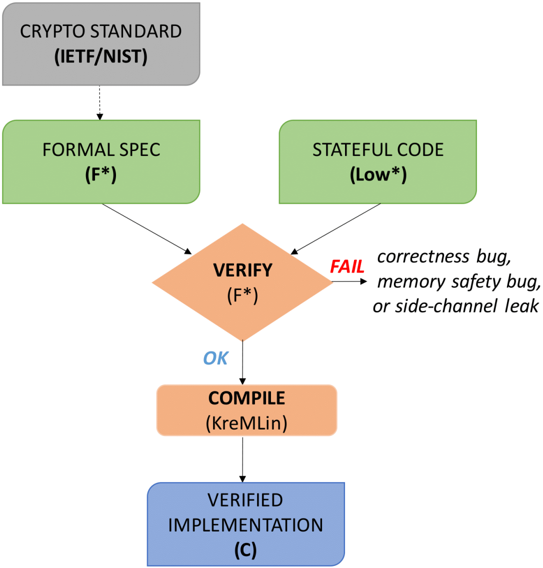
Writing Verified Crypto Code



Verification Guaranties:

1. *functional correctness*
2. *memory safety*
3. *secret independence*

HACL* programming and verification workflow



Agenda

How to write a formally verified implementation of cryptographic algorithms in F⋆?

- Example: Poly1305
- Field operations
  - Unsaturated and saturated bignum representations
  - Modulo-specific optimizations
- Polynomial evaluation
- Demo
  - F⋆ specification
  - Low⋆ implementation
- Do you want to give it a try? ☺

Poly1305

- a one-time MAC[1] function
- takes a 32-byte *key* and a *message* of arbitrary length and produces a 16-byte *tag*
- standardized as IETF RFC 7539 "ChaCha20 and Poly1305 for IETF Protocols" in 2015
- designed by Bernstein in 2005

[1] Message Authentication Code (MAC)

Poly1305

How to compute a 16-byte tag?

- split a 32-byte *key* into two 128-bit integers r and s, where r should be clamped
- split an input *message* into 16-byte blocks encoded to the field elements $m_1, \ldots, m_n$
- evaluate the following polynomial over $\mathbb{F}_p$, where $p = 2^{130} - 5$

  $acc = m_1 \times r^n + m_2 \times r^{n-1} + \ldots + m_n \times r \bmod p$

  in practice, Horner's method is used:

  $acc = (\ldots((0 + m_1) \times r + m_2) \times r + \ldots + m_n) \times r \bmod p$
- finally, compute

  $tag = acc + s \bmod 2^{128}$

Poly1305

How to compute a 16-byte tag?

- split an input *message* into 16-byte blocks encoded to the field elements $m_1, \ldots, m_n$
- evaluate the following polynomial over $\mathbb{F}_p$, where $p = 2^{130} - 5$

$$acc = (\ldots((0 + m_1) \times r + m_2) \times r + \ldots + m_n) \times r \bmod p$$

Bignum operations over $\mathbb{F}_{2^{130}-5}$

```
let prime = pow2 130 − 5
let felem = x:nat{x < prime}
let zero : felem = 0
let one : felem = 1

let fadd (x:felem) (y:felem) : felem = (x + y) % prime
let fmul (x:felem) (y:felem) : felem = (x ∗ y) % prime

let blocksize = 16
let block = lbytes blocksize
let block_last (len:nat{len < blocksize}) = lbytes len

val encode_block: b:block → felem
val encode_last: len:nat{len < blocksize} → b:block_last len → felem
```

Polynomial evaluation

$$acc = (\ldots((acc0 + m_1) \times r + m_2) \times r + \ldots + m_n) \times r \bmod p,$$
where $+$ is fadd and $\times$ is fmul

```
let poly_update1 (r:felem) (b:block) (acc:felem) : felem =
  (acc `fadd` encode_block b) `fmul` r

let poly_update_last (r:felem) len (b:block_last len) (acc:felem) : felem =
  if len = 0 then acc else (acc `fadd` encode_last len b) `fmul` r

let poly_update (msg:bytes) (acc0:felem) (r:felem) : felem =
  repeat_blocks #uint8 #felem #felem blocksize msg
    (poly_update1 r)
    (poly_update_last r)
  acc0
```

F⋆ specification

- F⋆ specification:
  `https://github.com/aseemr/Indocrypt-VerifiedCrypto-Tutorials/blob/main/FStar/exercises/lowstar/example-poly/Spec.Poly.fst`
- For simplicity, we ignore the last block if it is partial

F⋆ specification

- F⋆ specification:

  `https://github.com/aseemr/Indocrypt-VerifiedCrypto-Tutorials/blob/main/FStar/exercises/lowstar/example-poly/Spec.Poly.fst`
- For simplicity, we ignore the last block if it is partial
- Are we ready for the Low⋆ implementation?
- Not yet, since we are dealing with 130-bit integers!

Representing large integers

- Any number can be represented as $a = (a_n\ a_{n-1}\ \ldots\ a_0)_r$, where r is called **radix** or **base**
- evaluation function as_nat $a = \sum_{i=0}^{n} a_i \cdot r^i$

r	numeral systems
2	binary
8	octal
10	decimal
16	hexadecimal
...	

r	unsigned machine integers
2^8	uint8
2^{16}	uint16
2^{32}	uint32
2^{64}	uint64
...	

- if $0 \leq a_i < r$
  - \+ such a representation is unique
  - − some arithmetic operations may require to handle carries
- if a_i might be $\geq r$
  - \+ we can postpone carry propagation!
  - a is in a *reduced* form if $0 \leq a_i < r$

Bignum operations over $\mathbb{F}_{2^{130}-5}$

- Let's stick with radix-2^{26} representation of a field element
- as_nat $a = \sum_{i=0}^{4} a_i \cdot 2^{26 \cdot i}$
- bignum addition

  as_nat a + as_nat b ==
  $\sum_{i=0}^{4} a_i \cdot r^i + \sum_{i=0}^{4} b_i \cdot r^i ==$
  $\sum_{i=0}^{4} (a_i + b_i) \cdot r^i ==$
  as_nat (a ‘fadd‘ b)
- multiplication by a scalar

  as_nat $a \cdot b_i$ ==
  $\left(\sum_{i=0}^{4} a_i \cdot r^i\right) \cdot b_i ==$
  $\sum_{i=0}^{4} (a_i \cdot b_i) \cdot r^i ==$
  as_nat (a ‘smul‘ b_i)

Bignum operations over $\mathbb{F}_{2^{130}-5}$

- multiplication by a scalar and then bignum addition

  $$\text{as\_nat } a \cdot b_i + \text{as\_nat } c ==$$
  $$\left(\sum_{i=0}^{4} a_i \cdot r^i\right) \cdot b_i + \sum_{i=0}^{4} c_i \cdot r^i ==$$
  $$\sum_{i=0}^{4}(a_i \cdot b_i + c_i) \cdot r^i ==$$
  $$\text{as\_nat } ((a \text{ `sfmul` } b_i) \text{ `fadd` } c_i)$$

- bignum multiplcation

  $$\text{as\_nat } a \cdot \text{as\_nat } b ==$$
  $$\text{as\_nat } a \cdot \left(\sum_{i=0}^{4} b_i \cdot r^i\right) ==$$
  $$\text{as\_nat } a \cdot b_0 + \text{as\_nat } a \cdot (b_1 \cdot r) + \text{as\_nat } a \cdot (b_2 \cdot r^2) +$$
  $$\text{as\_nat } a \cdot (b_3 \cdot r^3) + \text{as\_nat } a \cdot (b_4 \cdot r^4) ==$$

  $\ldots$

Bignum operations over $\mathbb{F}_{2^{130}-5}$

- we have such a nice property for modular reduction:
  $2^{130} \bmod p = 5$ or $r^5 \bmod p = 5$
- $\text{as\_nat}\ a \cdot r \bmod p ==$
  $\text{as\_nat}\ (a_4\ a_3\ a_2\ a_1\ a_0)_r \cdot r \bmod p ==$
  $(a_0 \cdot p + a_1 \cdot p^2 + a_2 \cdot p^3 + a_3 \cdot p^4 + a_4 \cdot r^5) \bmod p ==$
  $(a_0 \cdot p + a_1 \cdot p^2 + a_2 \cdot p^3 + a_3 \cdot p^4 + a_4 \cdot 5) \bmod p ==$
  $\text{as\_nat}\ (a_3\ a_2\ a_1\ a_0\ (5 \cdot a_4))_r \bmod p$
- modular bignum multiplication
  $(\text{as\_nat}\ a \cdot \text{as\_nat}\ b) \bmod p ==$
  $(\text{as\_nat}\ a \cdot b_0 + \text{as\_nat}\ a \cdot (b_1 \cdot r) + \cdots) \bmod p$
  $(\text{as\_nat}\ a \cdot b_0 + \text{as\_nat}\ (a_3\ a_2\ a_1\ a_0\ (5 \cdot a_4))_r \cdot b_1 + \cdots) \bmod p$

Bignum operations over $\mathbb{F}_{2^{130}-5}$

So far so good? In the real world, our coefficients are bounded with $2^{machine\ word\ size}$. So we need to make sure that all our computations won't "overflow"

- bignum addition

  $$\text{as\_nat}\ a + \text{as\_nat}\ b ==$$
  $$\sum_{i=0}^{4} a_i \cdot r^i + \sum_{i=0}^{4} b_i \cdot r^i ==$$
  $$\sum_{i=0}^{4} (a_i + b_i) \cdot r^i == \{a_i + b_i < 2^{bits\ t}\}$$
  $$\text{as\_nat}\ (a \text{ `fadd` } b)$$

- multiplication by a scalar

  $$\text{as\_nat}\ a \cdot b_i ==$$
  $$\left(\sum_{i=0}^{4} a_i \cdot r^i\right) \cdot b_i ==$$
  $$\sum_{i=0}^{4} (a_i \cdot b_i) \cdot r^i == \{a_i \cdot b_i < 2^{bits\ t}\}$$
  $$\text{as\_nat}\ (a \text{ `smul` } b_i)$$

Bignum operations over $\mathbb{F}_{2^{130}-5}$

So far so good? In the real world, our coefficients are bounded with $2^{machine\ word\ size}$. So we need to make sure that all our computations won't "overflow"

- multiplication by a scalar and then bignum addition

  $\text{as\_nat } a \cdot b_i + \text{as\_nat } c ==$
  $\left(\sum_{i=0}^{4} a_i \cdot r^i\right) \cdot b_i + \sum_{i=0}^{4} c_i \cdot r^i ==$
  $\sum_{i=0}^{4}(a_i \cdot b_i + c_i) \cdot r^i == \{a_i \cdot b_i + c_i < 2^{bits\ t}\}$
  $\text{as\_nat } ((a \text{ `sfmul` } b_i) \text{ `fadd` } c_i)$

- bignum multiplcation

  $\text{as\_nat } a \cdot \text{as\_nat } b ==$
  $\text{as\_nat } a \cdot \left(\sum_{i=0}^{4} b_i \cdot r^i\right) ==$
  $\text{as\_nat } a \cdot b_0 + \text{as\_nat } a \cdot (b_1 \cdot r) + \text{as\_nat } a \cdot (b_2 \cdot r^2) +$
  $\text{as\_nat } a \cdot (b_3 \cdot r^3) + \text{as\_nat } a \cdot (b_4 \cdot r^4) ==$
  ... really?

Definition of machine integer base types in HACL*

```
type inttype =
  | U1 | U8 | U16 | U32 | U64 | U128 | S8 | S16 | S32 | S64 | S128
type secrecy_level =
  | SEC | PUB
val sec_int_t: inttype → Type0 (* secret machine integers *)
let pub_int_t (t:inttype) = (* public machine integers *)
  match t with
  | U1 → n:UInt8.t{UInt8.v n < 2}
  | U8 → UInt8.t
  | U16 → UInt16.t
  | U32 → UInt32.t
  | ...
let int_t (t:inttype) (l:secrecy_level) =
  match l with
  | PUB → pub_int_t t
  | SEC → sec_int_t t

val add_mod: #t:inttype{unsigned t} → #l:secrecy_level
  → int_t t l → int_t t l → int_t t l
```

Representation an element of $\mathbb{F}_{2^{130}-5}$ in F*

- radix-2^{26} representation

```
let felem5 = (uint64 & uint64 & uint64 & uint64 & uint64)
let as_nat5 (f:felem5) : GTot nat =
  let (s0, s1, s2, s3, s4) = f in
  v s0 + v s1 * pow26 + v s2 * pow52 + v s3 * pow78 + v s4 * pow104
```

Representation an element of $\mathbb{F}_{2^{130}-5}$ in F*

- radix-2^{26} representation

```
let felem5 = (uint64 & uint64 & uint64 & uint64 & uint64)
let as_nat5 (f:felem5) : GTot nat =
  let (s0, s1, s2, s3, s4) = f in
  v s0 + v s1 * pow26 + v s2 * pow52 + v s3 * pow78 + v s4 * pow104
```

- f is in a *reduced* form if felem_fits5 f (1, 1, 1, 1, 1) holds

```
let scale32 = s:nat{s ≤ 64}
let nat5 = (nat & nat & nat & nat & nat)
let scale32_5 = x:nat5{let (x1,x2,x3,x4,x5) = x in
                        x1 ≤ 64 ∧ x2 ≤ 64 ∧ x3 ≤ 64 ∧ x4 ≤ 64 ∧ x5 ≤ 64}
let felem_fits1 (x:uint64) (m:scale32) =
  uint_v x ≤ m * max26
let felem_fits5 (f:felem5) (m:scale32_5) =
  let (x1,x2,x3,x4,x5) = f in
  let (m1,m2,m3,m4,m5) = m in
  felem_fits1 x1 m1 ∧ felem_fits1 x2 m2 ∧
  felem_fits1 x3 m3 ∧ felem_fits1 x4 m4 ∧
  felem_fits1 x5 m5
```

Bignum addition over $\mathbb{F}_{2^{130}-5}$

```
val fadd5: f:felem5 → g:felem5 → Pure felem5
  (requires
    felem_fits5 f (2, 2, 2, 2, 2) ∧
    felem_fits5 g (1, 1, 1, 1, 1))
  (ensures λ out →
    felem_fits5 out (3, 3, 3, 3, 3) ∧
    feval5 out == fadd (feval5 f) (feval5 g))
let fadd5 (f0, f1, f2, f3, f4) (g0, g1, g2, g3, g4) =
  let o0 = f0 +! g0 in
  let o1 = f1 +! g1 in
  let o2 = f2 +! g2 in
  let o3 = f3 +! g3 in
  let o4 = f4 +! g4 in
  (o0, o1, o2, o3, o4)
```

- No need to compute modular reduction immediately

Bignum multiplication over $\mathbb{F}_{2^{130}-5}$

```
val mul5: f:felem5 → r:felem5 → r5:felem5 → Pure felem_wide5
  (requires
    felem_fits5 f (3, 3, 3, 3, 3) ∧ felem_fits5 r (1, 1, 1, 1, 1) ∧
    felem_fits5 r5 (5, 5, 5, 5, 5) ∧ r5 == precomp_r5 r)
  (ensures λ out →
    felem_wide_fits5 out (63, 51, 39, 27, 15) ∧
    feval_wide out == fmul (feval5 f) (feval5 r))
let mul5 (f0, f1, f2, f3, f4) (r0, r1, r2, r3, r4) (r50, r51, r52, r53, r54) =
  let out = smul5 f0 (r0, r1, r2, r3, r4) in
  let out = smul_add5 f1 (r54, r0, r1, r2, r3) out in
  let out = smul_add5 f2 (r53, r54, r0, r1, r2) out in
  let out = smul_add5 f3 (r52, r53, r54, r0, r1) out in
  let out = smul_add5 f4 (r51, r52, r53, r54, r0) out in
  out
```

Demo

- Low-level specification written in F⋆:

  example-poly/Hacl.Spec.Poly.fst

  example-poly/Hacl.Spec.Poly.Lemmas.fst

  example-poly/Hacl.Spec.Poly.Lemmas0.fst

- Low⋆ implementation:

  example-poly/Hacl.Impl.Poly.Field.fst

  example-poly/Hacl.Impl.Poly.fst

Exercise

Write a verified implementation of Gimli[2] in F*!

What should you first look at?

- lowstar/gimli/Spec.Gimli.fst
- lowstar/gimli/Hacl.Impl.Gimli.fst

	F*	Low*
	Lib.IntTypes.fsti	
	Lib.RawIntTypes.fsti (BREAKS secret independence)	
lib/	Lib.Sequence.fsti	Lib.Buffer.fsti
	Lib.ByteSequence.fsti	Lib.ByteBuffer.fsti
	Lib.LoopCombinators.fsti	Lib.Loops.fsti
	specs/	*code/*
chacha20	Spec.Chacha20.fst	Hacl.Impl.Chacha20.Core32.fst
		Hacl.Impl.Chacha20.fst
SHA3	Spec.SHA3.fst	Hacl.Impl.SHA3.fst

[2] Gimli: a cross-platform permutation `https://gimli.cr.yp.to/spec.html`

Conclusion

Questions?

- HACL*: `https://github.com/project-everest/hacl-star`
- F*: `https://www.fstar-lang.org`

- INRIA PROSECCO: `http://prosecco.inria.fr`
- Microsoft Project Everest: `https://project-everest.github.io`