Formal Functional Correctness of libsecp256k1's Modular Inverse

Dr. Russell O'Connor

Ordo Ordinalium Computationis Formae Fidelis Concentus Zenla

Correctness

Software is written by humans and therefore has bugs.
—John Jacobs

The job of putting in bugs is called programming.
—Ellen Ullman

Correctness

Is it even possible to write correct software?

Correctness

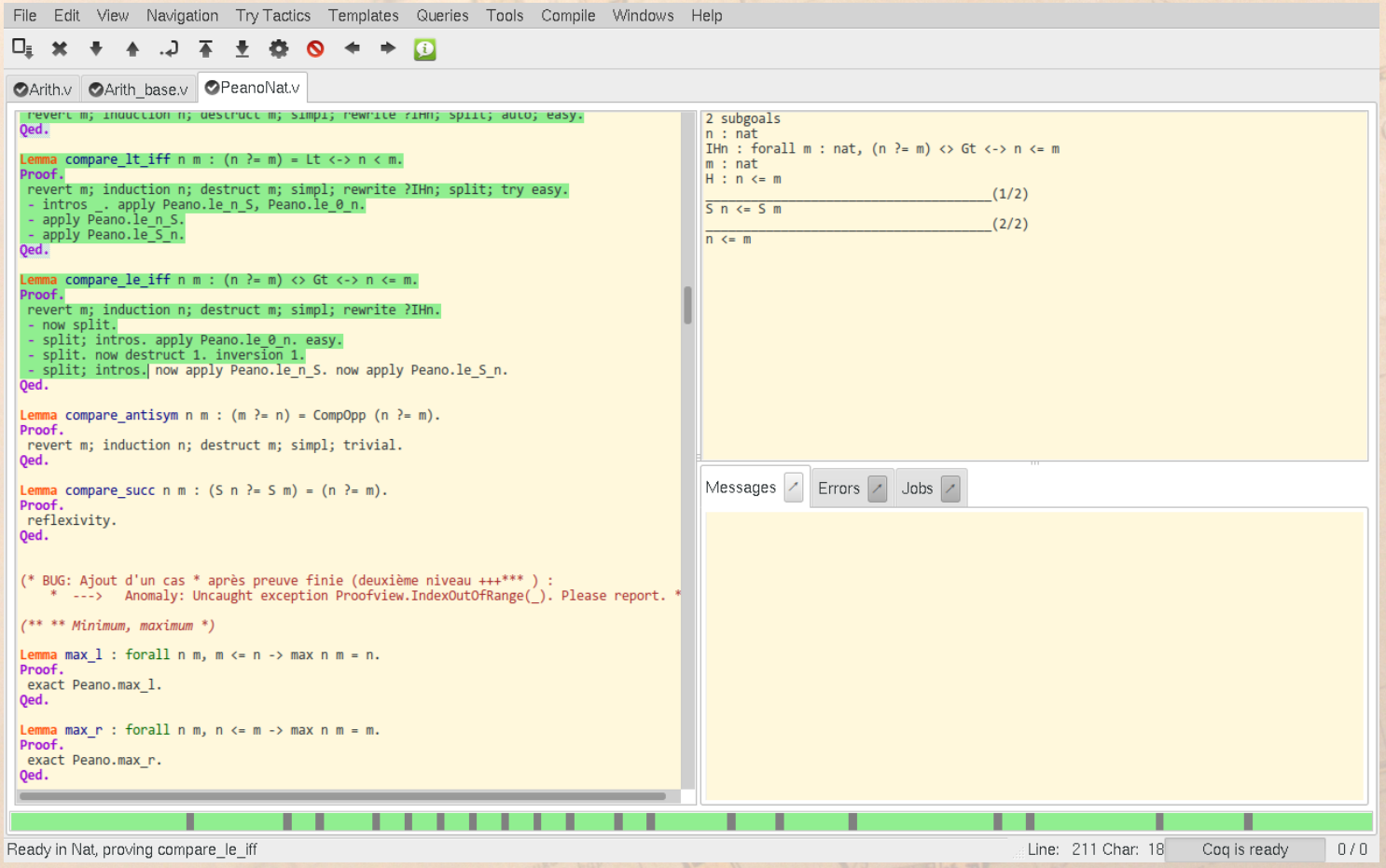
Is it possible to write correct ~~software~~ mathematics?

Formal Logic

There is a methodology for what constitutes a correct mathematical proof!

$$\frac{p \qquad p \rightarrow q}{\therefore q}$$

The Rocq Prover



History of SafeGCD

- Modular inverse is a key cryptographic subroutine.

$$x \cdot y \equiv 1 \pmod{p}$$

- Modular inverses can be computed with the Euclidean Algorithm. (300 B.C.)

$$x \cdot y + k \cdot p = 1$$

- A variant called SafeGCD by Bernstein and Yang (2019 A.D.)
- Merged into libsecp256k1 by Peter Dettman (2021 A.D.)

SafeGCD Correctness

How do we know the new SafeGCD algorithm is correct?

SafeGCD Correctness

- The SafeGCD algorithm has an invariant so that *when* f is 1 then d is the modular inverse.

$$x \cdot d \equiv f \pmod{p}$$

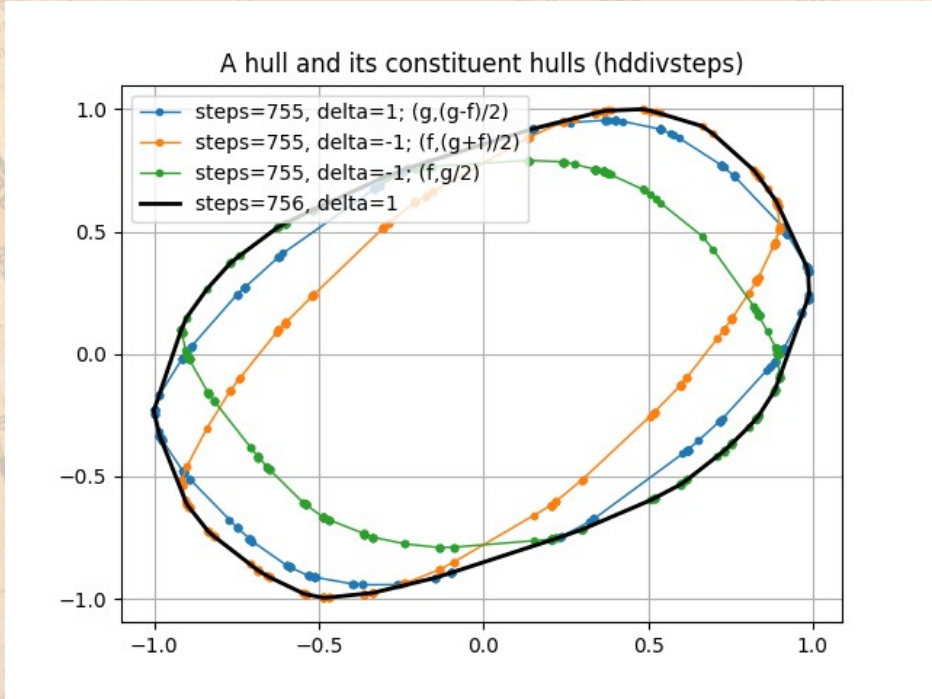
$$x \cdot e \equiv g \pmod{p}$$

$$\gcd(f,g) = \gcd(p,x)$$

- Proving that the algorithm terminates is not clear.
  - The original paper relies on the “spectral radius” of matrices.

SafeGCD Correctness

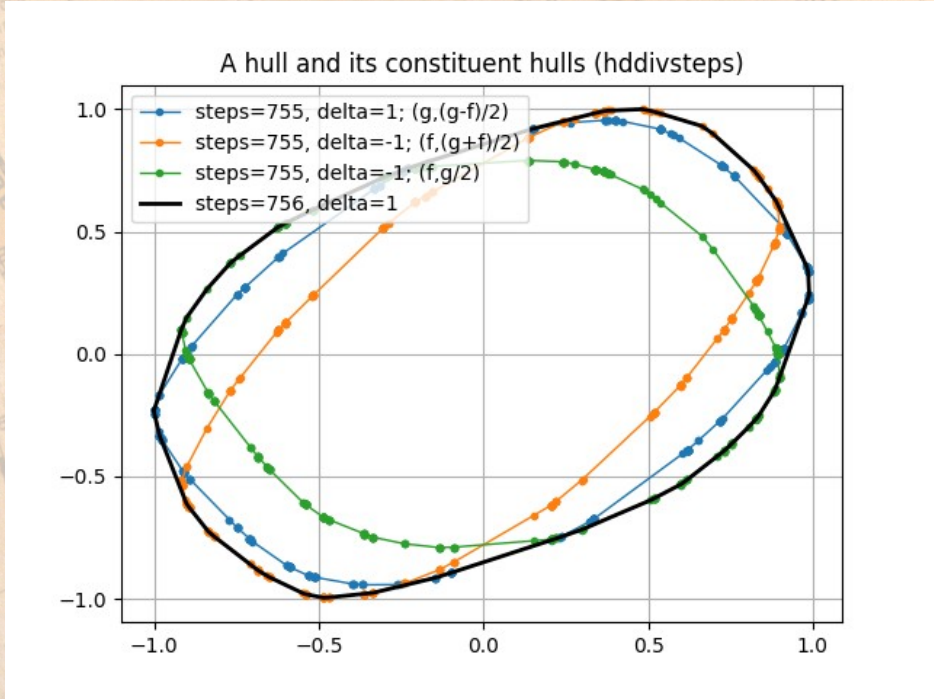
- Pieter Wuille's proof depends on an iterated convex hull argument of rational points.
- This computation was "verified" by a Python program.



safegcd-bounds

SafeGCD Correctness

- Verification program reimplemented in the Rocq prover and proved correct by O'Connor and Polestra. (2021 A.D.)



safegcd-bounds

Proof by Reflection

Proofs and programs are intertwined in dependent type theory. Proofs can reason about programs, and programs can be executed to yield proofs.

How do we know the implementation is correct?

```
static int64_t secp256k1_modinv64_divsteps_62_var(int64_t eta, uint64_t f0, uint64_t g0, secp256k1_modinv64_trans2x2
*t) {
    /* Transformation matrix; see comments in secp256k1_modinv64_divsteps_62. */
    uint64_t u = 1, v = 0, q = 0, r = 1;
    uint64_t f = f0, g = g0, m;
    uint32_t w;
    int i = 62, limit, zeros;

    for (;;) {
        /* Use a sentinel bit to count zeros only up to i. */
        zeros = secp256k1_ctz64_var(g | (UINT64_MAX << i));
        /* Perform zeros divsteps at once; they all just divide g by two. */
        g >>= zeros;
        u <<= zeros;
        v <<= zeros;
        eta -= zeros;
        i -= zeros;
        /* We're done once we've done 62 divsteps. */
        if (i == 0) break;
        VERIFY_CHECK((f & 1) == 1);
        VERIFY_CHECK((g & 1) == 1);
        VERIFY_CHECK((u * f0 + v * g0) == f << (62 - i));
        VERIFY_CHECK((q * f0 + r * g0) == g << (62 - i));
        /* Bounds on eta that follow from the bounds on iteration count (max 12*62 divsteps). */
        VERIFY_CHECK(eta >= -745 && eta <= 745);
        /* If eta is negative, negate it and replace f,g with g,-f. */
        if (eta < 0) {
```

Implementation Correctness

- Hoare Triples introduced by Floyd and Hoare. (1960 A.D.)

  {precondition} code {postcondition}

- Reynolds et. al. introduce the *separating conjunction*: P∗Q (1999 A.D.)
- Yang proves the completeness of *separation logic*. (2001 A.D.)

Separation Logic

$$\{p \mapsto i \ * \ q \mapsto j\}$$

$$\begin{aligned} *p &= *p \wedge *q \\ *q &= *p \wedge *q \\ *p &= *p \wedge *q \end{aligned}$$

$$\{p \mapsto j \ * \ q \mapsto i\}$$

Separation Logic

For the last 30 years experts have regarded pointer manipulation as an unsolved challenge for program verification and shared-memory concurrency as an even greater challenge. Now, thanks to Concurrent Separation Logic, both of these problems have been elegantly and efficiently solved; and they have the same solution.

—2016 Gödel Prize citation

Verified Software Toolchain

- VST implements Separation logic in the Rocq prover.
- C semantics provided by COMPCERT.

SafeGCD's C Implementation

```
Lemma body_secp256k1_modinv64_var: semax_body Vprog Gprog
f_secp256k1_modinv64_var secp256k1_modinv64_var_spec.
Proof.
start_function.
fastforward 5.
change (upd_Znth 4 _ _) with (map Vlong (Signed62.reprn 5 0)).
fastforward 5.
change (upd_Znth 4 _ _) with (map Vlong (Signed62.reprn 5 1)).
unfold make_modinfo.
unfold_data_at (data_at _ _ _ modinfo).
assert_PROP (field_compatible t_secp256k1_modinv64_modinfo (DOT _modulus)
modinfo) by entailer.
forward_call (m, v_f, field_address t_secp256k1_modinv64_modinfo (DOT
_modulus) modinfo, Tsh, sh_modinfo);
[rewrite field_address_offset by assumption;entailer!! [].
change (data_at sh_modinfo t_secp256k1_modinv64_signed62) with
 (data_at sh_modinfo (nested_field_type t_secp256k1_modinv64_modinfo (DOT
_modulus))).
rewrite ← field_at_data_at.
assert (Hmodinfo :
 ((field_at sh_modinfo t_secp256k1_modinv64_modinfo (DOT _modulus) (map Vlong
(Signed62.reprn 5 m)) modinfo) *
  (field_at sh_modinfo t_secp256k1_modinv64_modinfo (DOT _modulus_inv62)
(Vlong (Int64.repr (modInv m (2 ^ 62)))) modinfo) ⊢
  data_at sh_modinfo t_secp256k1_modinv64_modinfo (map Vlong (Signed62.reprn 5
m), Vlong (Int64.repr (modInv m (2 ^ 62)))) modinfo))
 by (unfold_data_at (data_at _ _ _ modinfo);entailer!!).
sep_apply Hmodinfo.
fold (make_modinfo m).
clear Hmodinfo.
forward_call.
fastforward 3.
change (Int.neg (Int.repr 1)) with (Int.repr (-1)).
rewrite ← Zodd_equiv in H.
set (init := divstep.init m x H).
assert (HfBound : forall i, -m < divstep.f (fst (divstep.stepN i init)) ⩽ m).
1:{
  clear -H0 H1; intro i.
  injection (divstep.Trans.transN_step i init).
  intros _.
  destruct (divstep.Trans.bounded_transN i init) as [[Huv Huv'] _].
  nia.
}
assert (HgBound : forall i, -m < divstep.g (fst (divstep.stepN i init)) < m).
1:{
  clear -H0 H1; intro i.
  case (divstep.fgBoundsStrict init i);[cbn; lia[].
  intros _.
  change (divstep.f init) with m.
  change (divstep.g init) with x.
  lia.
}
set (invariant := fun P f ⇒ (EX i:nat, EX len:nat, EX d:Z, EX e:Z,
 PROP ( 0 ⩽ Z.of_nat i < 12
      ; (1 ⩽ len ⩽ 5)%nat
      ; -2^(62 * Z.of_nat len + 1) ⩽ divstep.f (fst (divstep.stepN (f i)
init)) ⩽ 2^(62 * Z.of_nat len + 1) - 1
```

```
Espec : OracleKind
x, m : Z
ptrx, modinfo : val
shx, sh_modinfo, sh_debruijn, sh_SECP256K1_SIGNED62_ONE : share
gv : globals
Delta_specs := abbreviate
            : Maps.PTree.t funspec
Delta := abbreviate
      : tycontext
v_d, v_e, v_f, v_g, v_t : val
H : Z.Odd m
H0 : 0 ⩽ x < m
H1 : 1 < m < 2 ^ 256
H2 : x = 0 ∨ rel_prime x m
SH : writable_share shx
SH0 : readable_share sh_modinfo
SH1 : readable_share sh_debruijn
SH2 : readable_share sh_SECP256K1_SIGNED62_ONE
POSTCONDITION := abbreviate
              : ret_assert
MORE_COMMANDS := abbreviate
              : statement
```

```
semax Delta
  (PROP ( )
   LOCAL (lvar _t
            (Tstruct _secp256k1_modinv64_trans2×2
               noattr) v_t;
   lvar _g
     (Tstruct _secp256k1_modinv64_signed62 noattr) v_g;
   lvar _f
     (Tstruct _secp256k1_modinv64_signed62 noattr) v_f;
   lvar _e
     (Tstruct _secp256k1_modinv64_signed62 noattr) v_e;
   lvar _d
     (Tstruct _secp256k1_modinv64_signed62 noattr) v_d;
   gvars gv; temp _x ptrx; temp _modinfo modinfo)
   SEP (data_at_ Tsh
          (Tstruct _secp256k1_modinv64_trans2×2 noattr)
          v_t;
   data_at_ Tsh
     (Tstruct _secp256k1_modinv64_signed62 noattr) v_g;
   data_at_ Tsh
     (Tstruct _secp256k1_modinv64_signed62 noattr) v_f;
   data_at_ Tsh
     (Tstruct _secp256k1_modinv64_signed62 noattr) v_e;
   data_at_ Tsh
     (Tstruct _secp256k1_modinv64_signed62 noattr) v_d;
   data_at_shx_t_secp256k1_modinv64_signed62
```

raw.githack.com/BlockstreamResearch/simplicity/c1dddedd553b403da877377e658f17f0d2184cc4/alectryon/verif_modinv64_impl.v.html

Modular Inverse Correctness

The Rocq prover allows us to combine the SafeGCD proof of termination with Separation Logic of VST to prove functional correctness of the C implementation.

Caveats

- VST does not support passing, returning, and assignment of structures.
- Constant-time not covered by the proof.
- No support for reasoning about C++.

Is it even possible to write correct software?

There is an alternative to programming beyond our ability to reason about the code we write. We have tools needed for eliminating defects from our software. We simply need to slow down and use them.

Pick up the Torch

- Remix7531 has already done a Formal Verification of secp256k1 modular scalar multiplication.

Now it is your turn.