Verified Arithmetic for Cryptography

John Harrison
Amazon Web Services

Certified and Symbolic-Numeric Computation, Lyon

Tue 23rd May 2023 (11:30–12:30)

Plan for the talk

- Arithmetic, bignums, cryptography
- Security, side-channels and “constant-time” code
- Proving machine code correct

Bignums in cryptography

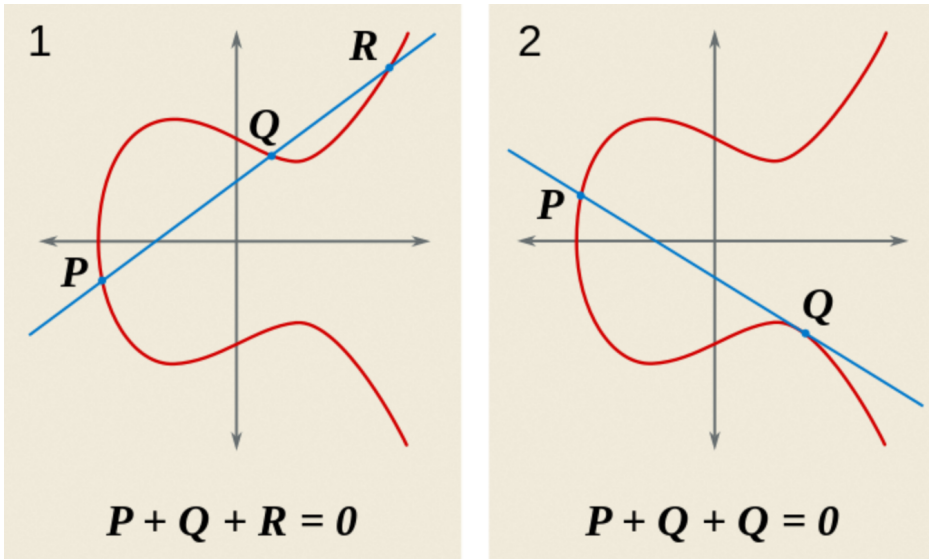
Bignums (typically 256 – 4096 bits long) are pervasive in public-key cryptography

- RSA: uses modular exponentiation $a^k \bmod n$ directly
- ECDH, ECDSA: point operations on elliptic curve over finite fields

We will usually consider nonnegative numbers (main interest is in modular operations), but use 2s complement where appropriate for negation.

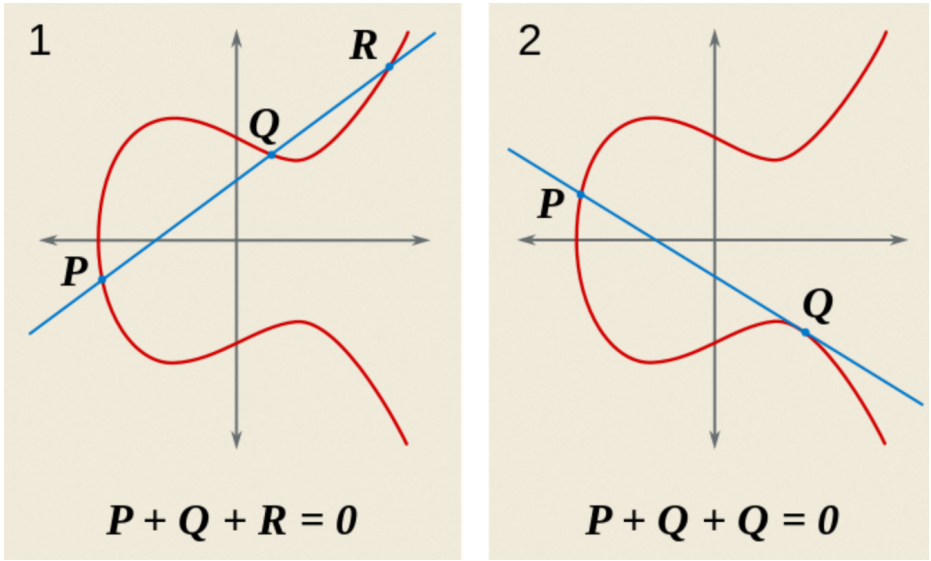
Elliptic curves

Often defined by equations of the form $y^2 = x^3 + ax + b$



Elliptic curves

Often defined by equations of the form $y^2 = x^3 + ax + b$



Coordinate operations are just modular bignum arithmetic.

A few popular prime moduli for elliptic curves

- NIST P-256: $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
- NIST P-384: $2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
- NIST P-521: $2^{521} - 1$
- Curve25519: $2^{255} - 19$
- SM2: $2^{256} - 2^{224} - 2^{96} + 2^{64} - 1$

RSA uses much bigger moduli, usually product of two primes.

Bignums in cryptography

Crypto bignums should have three properties:

- Efficient
- Correct
- Secure

Bignums in cryptography

Crypto bignums should have three properties:

- Efficient
- Correct
- Secure

Public crypto libraries aim for these goals, but sometimes don't achieve all of them.

Improving the public libraries

Two main components of libraries like `OpenSSL` and `BoringSSL`:

- `libtls`: Transport layer security
- `libcrypto`: Cryptography

Efficient, correct and secure

The goal for `s2n-bignum` is to provide a basic foundational layer of arithmetic on which `libcrypto` operations can be based, which is

Efficient, correct and secure

The goal for `s2n-bignum` is to provide a basic foundational layer of arithmetic on which `libcrypto` operations can be based, which is

- Efficient: hand-crafted machine code

Efficient, correct and secure

The goal for `s2n-bignum` is to provide a basic foundational layer of arithmetic on which `libcrypto` operations can be based, which is

- Efficient: hand-crafted machine code
- Correct: every function is formally verified mathematically

Efficient, correct and secure

The goal for `s2n-bignum` is to provide a basic foundational layer of arithmetic on which `libcrypto` operations can be based, which is

- Efficient: hand-crafted machine code
- Correct: every function is formally verified mathematically
- Secure: all code is written in "constant-time" style

Efficient, correct and secure

The goal for `s2n-bignum` is to provide a basic foundational layer of arithmetic on which `libcrypto` operations can be based, which is

- Efficient: hand-crafted machine code
- Correct: every function is formally verified mathematically
- Secure: all code is written in "constant-time" style

All code is open-source:
`https://github.com/awslabs/s2n-bignum`

Security, side-channels and "constant-time" code

Cyber-attacks

The crooked timber of humanity

Nearly two centuries ago, France was hit by the world's first cyber-attack. Tom Standage argues that it holds lessons for us today

https://www.economist.com/1843/2017/10/05/the-crooked-timber-of-humanity

Security holes in arithmetic?

THE PARIS256 ATTACK

Or, Squeezing a Key Through a Carry Bit.

Sean Devlin, Filippo Valsorda

Introduction

We present an adaptive key recovery attack exploiting a small carry propagation bug in the Go standard library implementation of the NIST P-256 elliptic curve, reported to the Go project as issue 20040.

Following our attack, the vulnerability was assigned CVE-2017-8932, and caused the release of Go 1.7.6 and 1.8.2.

`https://i.blackhat.com/us-18/Wed-August-8/us-18-Valsorda-Squeezing-A-Key-Through-A-Carry-Bit-wp.pdf`

Timing and cache attacks (1996, 2005)

Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems

Paul C. Kocher

Cryptography Research, Inc.
607 Market Street, 5th Floor, San Francisco, CA 94105, USA.
E-mail: paul@cryptography.com

Abstract. By carefully measuring the amount of time required to perform private key operations, attackers may be able to find fixed Diffie-Hellman exponents, factor RSA keys, and break other cryptosystems. Against a vulnerable system, the attack is computationally inexpensive and often requires only known ciphertext. Actual systems are potentially at risk, including cryptographic tokens, network-based cryptosystems, and other applications where attackers can make reasonably accurate timing measurements. Techniques for preventing the attack for RSA and Diffie-Hellman are presented. Some cryptosystems will need to be revised to protect against the attack, and new protocols and algorithms may need to incorporate measures to prevent timing attacks.

Keywords: timing attack, cryptanalysis, RSA, Diffie-Hellman, DSS.

CACHE MISSING FOR FUN AND PROFIT

COLIN PERCIVAL

ABSTRACT. Simultaneous multithreading — put simply, the sharing of the execution resources of a superscalar processor between multiple execution threads — has recently become widespread via its introduction (under the name "Hyper-Threading") into Intel Pentium 4 processors. In this implementation, for reasons of efficiency and economy of processor area, the sharing of processor resources between threads extends beyond the execution units; of particular concern is that the threads share access to the memory caches.

We demonstrate that this shared access to memory caches provides not only an easily used high bandwidth covert channel between threads, but also permits a malicious thread (operating, in theory, with limited privileges) to monitor the execution of another thread, allowing in many cases for theft of cryptographic keys.

Finally, we provide some suggestions to processor designers, operating system vendors, and the authors of cryptographic software, of how this attack could be mitigated or eliminated entirely.

https://paulkocher.com/doc/TimingAttacks.pdf
https://papers.freebsd.org/2005/cperciva-cache_missing.files/cperciva-cache_missing-paper.pdf

Attacking binary exponentiation

Simplified binary exponentiation by repeated squaring:

$$\begin{aligned} a^{2n} &= (a^n)^2 \\ a^{2n+1} &= a \times (a^n)^2 \end{aligned}$$

Attacking binary exponentiation

Simplified binary exponentiation by repeated squaring:

$$\begin{aligned} a^{2n} &= (a^n)^2 \\ a^{2n+1} &= a \times (a^n)^2 \end{aligned}$$

Example:

$$\begin{aligned} a^3 &= a \times a^2 \\ a^6 &= (a^3)^2 \\ a^{13} &= a \times (a^6)^2 \end{aligned}$$

Attacking binary exponentiation

Simplified binary exponentiation by repeated squaring:

$$\begin{aligned} a^{2n} &= (a^n)^2 \\ a^{2n+1} &= a \times (a^n)^2 \end{aligned}$$

Example:

$$\begin{aligned} a^3 &= a \times a^2 \\ a^6 &= (a^3)^2 \\ a^{13} &= a \times (a^6)^2 \end{aligned}$$

Each step does an extra multiplication for a 1 bit

Side-channels

Just some of many side-channels by which systems may ‘leak’ secret info (like a private key) to an observer:

- Execution time
- Memory access pattern
- Power consumption
- Electromagnetic radiation emitted
- ...
- Microarchitectural bugs

Side-channels

Just some of many side-channels by which systems may 'leak' secret info (like a private key) to an observer:

- Execution time ←
- Memory access pattern ←
- Power consumption
- Electromagnetic radiation emitted
- ...
- Microarchitectural bugs

Main worries in typical multitasking OS on shared machine

How can you avoid timing/cache side-channels?

Want execution time, if not literally constant, *uncorrelated* with (secret) data being manipulated. How?

How can you avoid timing/cache side-channels?

Want execution time, if not literally constant, *uncorrelated* with (secret) data being manipulated. How?

- Add randomization or salting to the algorithm
- Balance timing of paths
- Just make it too fast to observe

How can you avoid timing/cache side-channels?

Want execution time, if not literally constant, *uncorrelated* with (secret) data being manipulated. How?

- Add randomization or salting to the algorithm
- Balance timing of paths
- Just make it too fast to observe
- Always perform exactly the same operations regardless of (secret) data. $\longleftarrow$ Our chosen solution

How can you ‘always do the same thing’?

When there is control flow depending on secret data:

```
if (n >= p) n = n - p;
```

How can you ‘always do the same thing’?

When there is control flow depending on secret data:

```
if (n >= p) n = n - p;
```

convert it into dataflow using masking, conditional moves etc.

```
b = (n < p) - 1;
n = n - (p & b);
```

What about the compiler?

The compiler may naively turn mask creation back into a branch:

```
b = (n < p) - 1;
```

What about the compiler?

The compiler may naively turn mask creation back into a branch:

```
b = (n < p) - 1;
```

Time to break out your copy of "Hacker's Delight":

```
b = (((~n & p) | ((~n | p) & (n - p))) >> 63) - 1;
```

What about the compiler?

The compiler may naively turn mask creation back into a branch:

```
b = (n < p) - 1;
```

Time to break out your copy of "Hacker's Delight":

```
b = (((~n & p) | ((~n | p) & (n - p))) >> 63) - 1;
```

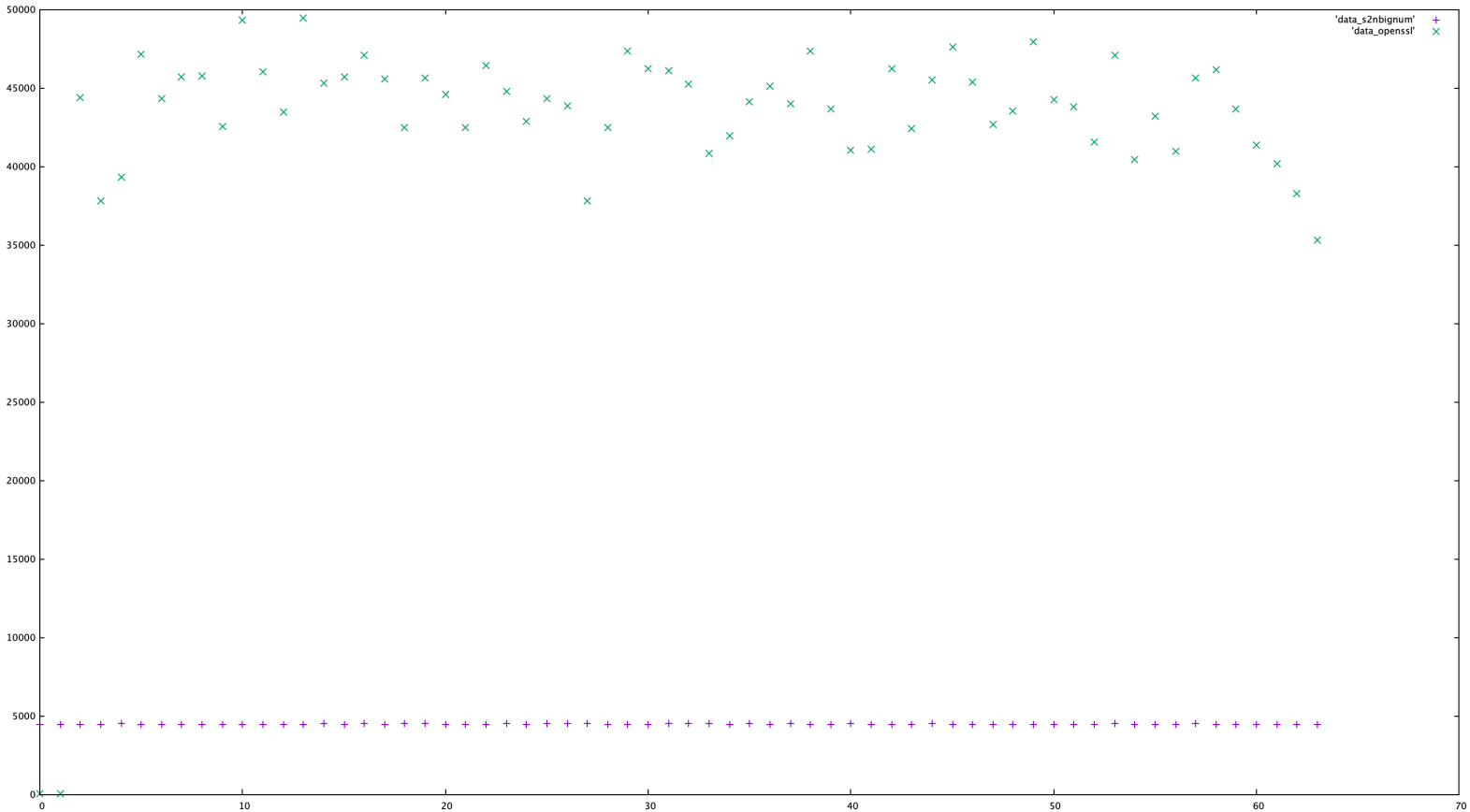
But then the compiler may even recognize this and transform it back.

Are the machine instructions constant-time?

- Some definitely not, e.g. division by zero is special
- General assumption that simple things like add, mul mostly are
- Almost never documented or guaranteed
- Interesting exception: ARM with DIT bit

Some empirical results on timing

Times for 384-bit modular inverse at bit densities 0–63, nanoseconds on Intel® Xeon® Platinum 8175M, 2.5 GHz.



With constant time flag, OpenSSL function is much slower *and* is still more variable.

Proving machine code correct

Life's better in machine code!

Working directly with machine code suits this project:

- Access to special instructions, operations on flags
- Precise scheduling to maximize efficiency
- Fine control of instruction sequence for constant-time-ness
- Simpler semantics than mainstream high-level languages

Life's better in machine code!

Working directly with machine code suits this project:

- Access to special instructions, operations on flags
- Precise scheduling to maximize efficiency
- Fine control of instruction sequence for constant-time-ness
- Simpler semantics than mainstream high-level languages

But machine code has familiar big drawbacks too . . .

Coding and verification approach

We verify pre-existing machine code, rather than autogenerate it 'correct by construction':

Coding and verification approach

We verify pre-existing machine code, rather than autogenerate it 'correct by construction':

- ☺ Independent of compiler or even macro-assembler correctness.

Coding and verification approach

We verify pre-existing machine code, rather than autogenerate it 'correct by construction':

- ☺ Independent of compiler or even macro-assembler correctness.
- ☺ Applicable to highly tuned efficient code that is hard to generate automatically as well as compiled C code etc.

Coding and verification approach

We verify pre-existing machine code, rather than autogenerate it 'correct by construction':

- ☺ Independent of compiler or even macro-assembler correctness.
- ☺ Applicable to highly tuned efficient code that is hard to generate automatically as well as compiled C code etc.
- ☺ Code is conventional human-readable and human-modifiable, usage is independent of prover infrastructure.

Coding and verification approach

We verify pre-existing machine code, rather than autogenerate it 'correct by construction':

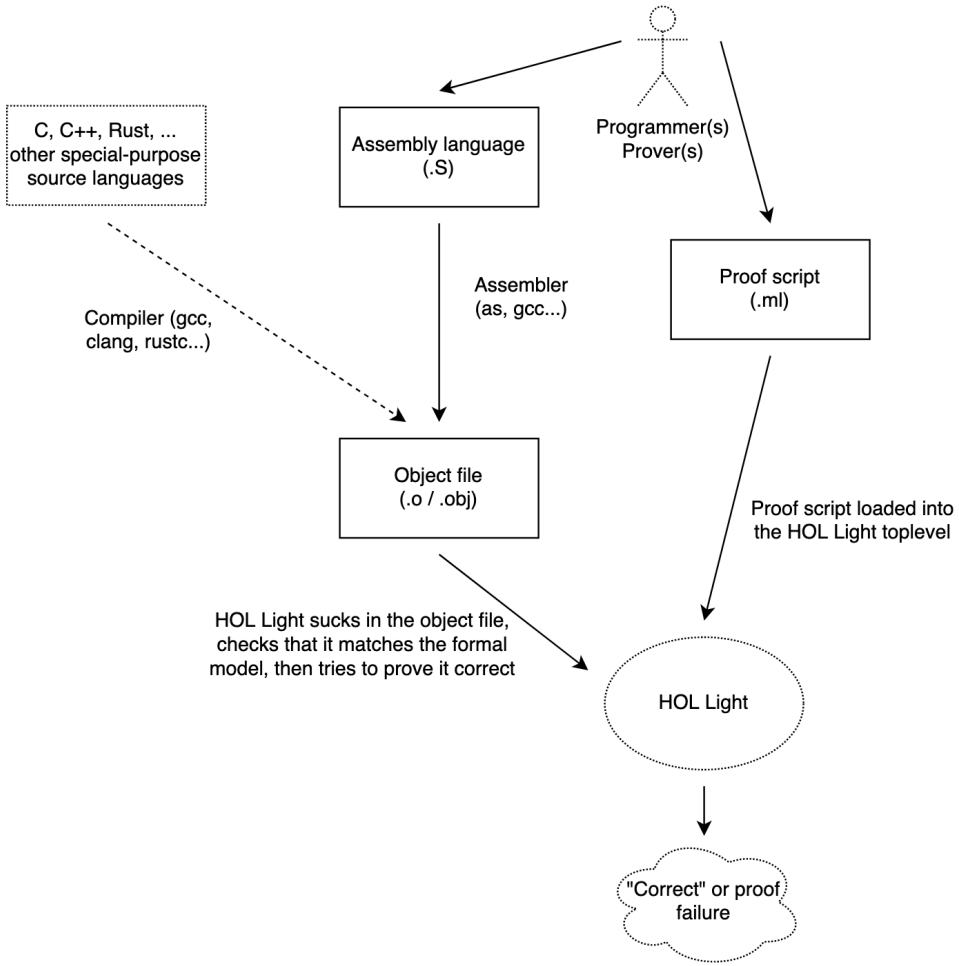
- ☺ Independent of compiler or even macro-assembler correctness.
- ☺ Applicable to highly tuned efficient code that is hard to generate automatically as well as compiled C code etc.
- ☺ Code is conventional human-readable and human-modifiable, usage is independent of prover infrastructure.
- ☹ Much more work involved writing code at this level, less structured representation.

Coding and verification approach

We verify pre-existing machine code, rather than autogenerate it 'correct by construction':

- ☺ Independent of compiler or even macro-assembler correctness.
- ☺ Applicable to highly tuned efficient code that is hard to generate automatically as well as compiled C code etc.
- ☺ Code is conventional human-readable and human-modifiable, usage is independent of prover infrastructure.
- ☹ Much more work involved writing code at this level, less structured representation.
- ☺/☹ Exposure of low-level details like exact stack and PC offsets and particular registers.

Coding and verification flow



Verifying the actual code

Formalization of code as byte sequence derived from the object file:

```
define_assert_from_elf "bignum_montmul_p256_mc"
  "arm/p256/bignum_montmul_p256.o"
```

Verifying the actual code

Formalization of code as byte sequence derived from the object file:

```
define_assert_from_elf "bignum_montmul_p256_mc"
  "arm/p256/bignum_montmul_p256.o"
```

Automatically re-check when code and/or proof changes:

```
p256/%.correct: proofs/%.ml p256/%.o ; .....
```

Modeling instruction decoding and execution

Decoding instruction byte sequences to their semantics:

```
...
| [0b11010110010111111000000:22; Rn:5; 0:5] ->
  SOME (arm_RET (XREG' Rn))
| [0b10011011110:11; Rm:5; 0b011111:6; Rn:5; Rd:5] ->
  SOME (arm_UMULH (XREG' Rd) (XREG' Rn) (XREG' Rm))
| [1:1; x; 0b1110000:7; ld; 0:1; imm9:9; 0b01:2; Rn:5; Rt:5] ->
  SOME (arm_ldst ld x Rt (XREG_SP Rn) (Postimmediate_Offset (word_sx imm9)))
...
```

Modeling instruction decoding and execution

Decoding instruction byte sequences to their semantics:

```
...
| [0b11010110010111111000000:22; Rn:5; 0:5] ->
  SOME (arm_RET (XREG' Rn))
| [0b10011011110:11; Rm:5; 0b011111:6; Rn:5; Rd:5] ->
  SOME (arm_UMULH (XREG' Rd) (XREG' Rn) (XREG' Rm))
| [1:1; x; 0b1110000:7; ld; 0:1; imm9:9; 0b01:2; Rn:5; Rt:5] ->
  SOME (arm_ldst ld x Rt (XREG_SP Rn) (Postimmediate_Offset (word_sx imm9)))
...
```

Semantics details the state changes from each instruction:

```
arm_ADDS Rd Rm Rn s =
  let m = read Rm s
  and n = read Rn s in
  let d = word_add m n in
  (Rd := d ,,
   NF := (ival d < &0) ,,
   ZF := (val d = 0) ,,
   CF := ~(val m + val n = val d) ,,
   VF := ~(ival m + ival n = ival d)) s
```

Nondeterminism

The semantics is a *relation* between initial and final states that might not be deterministic (= a function).

```
x86_IMUL3 dest (src1,src2) s =
  let x = read src1 s and y = read src2 s in
  let z = word_mul x y in
  (dest := z ,,
   CF := ~(ival x * ival y = ival z) ,,
   OF := ~(ival x * ival y = ival z) ,,
   UNDEFINED_VALUES[ZF;SF;PF;AF]) s
```

Correctness proved for *all possible* sequences of states from an initial state.

Hoare logic + Symbolic simulation

The approach to verification tries to combine the best of two previous methods:

- Machine code Hoare logic (as developed by Magnus Myreen)
- Symbolic simulation (as used in Galois's SAW tool)

Hoare logic + Symbolic simulation

The approach to verification tries to combine the best of two previous methods:

- Machine code Hoare logic (as developed by Magnus Myreen)
- Symbolic simulation (as used in Galois's SAW tool)

These are combined in two ways:

- Use Hoare logic for high-level invariants and breakpoints, symbolic simulation for routine parts.
- Symbolic simulation can simulate through subroutines atomically based on their Hoare triples.

Verification results

Correctness as elaborated Hoare triples with 'frame condition':

```
|- nonoverlapping (word pc,0x2de) (z,8 * 12) /\
   (y = z \/ nonoverlapping (y,8 * 6) (z,8 * 12)) /\
   nonoverlapping (x,8 * 6) (z,8 * 12)
   ==> ensures x86
        (\s. bytes_loaded s (word pc) bignum_mul_6_12_mc /\
             read RIP s = word(pc + 0x06) /\
             C_ARGUMENTS [z; x; y] s /\
             bignum_from_memory (x,6) s = a /\
             bignum_from_memory (y,6) s = b)
        (\s. read RIP s = word (pc + 0x2d7) /\
             bignum_from_memory (z,12) s = a * b)
        (MAYCHANGE [RIP; RAX; RBP; RBX; RCX; RDX;
                    R8; R9; R10; R11; R12; R13] ,,
         MAYCHANGE [memory :> bytes(z,8 * 12)] ,,
         MAYCHANGE SOME_FLAGS)
```

Symbolic state representation

```
ENSURES_INIT_TAC "s0" THEN
BIGNUM_LDIGITIZE_TAC "x_" `bignum_from_memory (x,6) s0` THEN
```

Sets up a machine state s0 with initial assumptions and digitized bignum.

Symbolic state representation

```
ENSURES_INIT_TAC "s0" THEN
BIGNUM_LDIGITIZE_TAC "x_" `bignum_from_memory (x,6) s0` THEN
```

Sets up a machine state s0 with initial assumptions and digitized bignum.

```
...
 3 [`bytes_loaded s0 (word pc) bignum_mul_6_12_mc`]
 4 [`read RIP s0 = word (pc + 6)`]
 5 [`read RDI s0 = z`]
 6 [`read RSI s0 = x`]
 7 [`read RDX s0 = y`]
 8 [`bignum_of_wordlist [x_0; x_1; x_2; x_3; x_4; x_5] = a`]
 9 [`bignum_from_memory (y,6) s0 = b`]
10 [`MAYCHANGE [] s0 s0`]
11 [`read (memory :> bytes64 x) s0 = x_0`]
...
16 [`read (memory :> bytes64 (word_add x (word 40))) s0 = x_5`]
```

Symbolic simulation

```
X86_ACCSTEPS_TAC BIGNUM_MUL_6_12_EXEC (1--132) (1--132) THEN
```

Effectively ‘executes’ code on the *symbolic* values

Symbolic simulation

```
  X86_ACCSTEPS_TAC BIGNUM_MUL_6_12_EXEC (1--132) (1--132) THEN
```

Effectively ‘executes’ code on the *symbolic* values

```
  3 [‘bignum_of_wordlist [x_0; x_1; x_2; x_3; x_4; x_5] = a‘]
  4 [‘bignum_of_wordlist [y_0; y_1; y_2; y_3; y_4; y_5] = b‘]
  5 [‘2 pow 64 * (val mulhi_s4) + (val mullo_s4) = (val y_0) * (val x_0)‘]
  6 [‘2 pow 64 * (val mulhi_s6) + (val mullo_s6) = (val y_0) * (val x_1)‘]
...
111 [‘2 pow 64 * (bitval carry_s126) + (val sum_s126) =
      (val sum_s125) + (bitval carry_s124)‘]
115 [‘read (memory :> bytes64 (word_add z (word 48))) s132 = sum_s112‘]
116 [‘read OF s132 <=> carry_s125‘]
117 [‘read RAX s132 = mullo_s123‘]
118 [‘read R11 s132 = sum_s121‘]
119 [‘read R9 s132 = sum_s115‘]
120 [‘read RDX s132 = y_5‘]
...
```

Conclusion by algebra . . . almost

```
ENSURES_FINAL_STATE_TAC THEN ASM_REWRITE_TAC[] THEN
CONV_TAC(LAND_CONV BIGNUM_LEXPAND_CONV) THEN ASM_REWRITE_TAC[] THEN
MAP_EVERY EXPAND_TAC ["a"; "b"]
```

The desired conclusion becomes an algebraic equation:

Conclusion by algebra . . . almost

```
  ENSURES_FINAL_STATE_TAC THEN ASM_REWRITE_TAC[] THEN
  CONV_TAC(LAND_CONV BIGNUM_LEXPAND_CONV) THEN ASM_REWRITE_TAC[] THEN
  MAP_EVERY EXPAND_TAC ["a"; "b"]
```

The desired conclusion becomes an algebraic equation:

```
...
150 [‘read (memory :> bytes64 (word_add z (word 88))) s132 = sum_s126‘]
151 [‘read RIP s132 = word (pc + 727)‘]

‘bignum_of_wordlist
 [mullo_s4; sum_s20; sum_s42; sum_s64; sum_s86; sum_s108; sum_s112; sum_s115;
  sum_s118; sum_s121; sum_s124; sum_s126] =
 bignum_of_wordlist [x_0; x_1; x_2; x_3; x_4; x_5] *
 bignum_of_wordlist [y_0; y_1; y_2; y_3; y_4; y_5]‘
```

Custom automation: absence of carries

Algebra *plus* bound reasoning to show some carries are zero, e.g.
$a \times b + c \leq (2^{64} - 1)^2 + (2^{64} - 1) = 2^{64}(2^{64} - 1) < 2^{128}$

```
ACCUMULATOR_POP_ASSUM_LIST(MP_TAC o end_itlist CONJ o DECARRY_RULE) THEN
DISCH_THEN(fun th -> REWRITE_TAC[th]) THEN REAL_ARITH_TAC
```

Custom automation: absence of carries

Algebra *plus* bound reasoning to show some carries are zero, e.g. $a \times b + c \leq (2^{64} - 1)^2 + (2^{64} - 1) = 2^{64}(2^{64} - 1) < 2^{128}$

```
ACCUMULATOR_POP_ASSUM_LIST(MP_TAC o end_itlist CONJ o DECARRY_RULE) THEN
DISCH_THEN(fun th -> REWRITE_TAC[th]) THEN REAL_ARITH_TAC
```

The goal is proved.

```
val it : goalstack = No subgoals
```

Custom automation: irrelevance of carries

In some settings it's easier to prove the range of the mathematical result a priori, so that anything beyond a certain bit is irrelevant. Analogous proof:

```
ACCUMULATOR_POP_ASSUM_LIST(MP_TAC o end_itlist CONJ o DESUM_RULE) THEN
DISCH_THEN(fun th -> REWRITE_TAC[th]) THEN REAL_ARITH_TAC
```

Custom automation: irrelevance of carries

In some settings it's easier to prove the range of the mathematical result a priori, so that anything beyond a certain bit is irrelevant. Analogous proof:

```
ACCUMULATOR_POP_ASSUM_LIST(MP_TAC o end_itlist CONJ o DESUM_RULE) THEN
DISCH_THEN(fun th -> REWRITE_TAC[th]) THEN REAL_ARITH_TAC
```

The approach just proves $y' \equiv y \pmod{2^b}$ by proving that $(y' - y)/2^b \in \mathbb{Z}$

Custom automation: irrelevance of carries

In some settings it's easier to prove the range of the mathematical result a priori, so that anything beyond a certain bit is irrelevant. Analogous proof:

```
ACCUMULATOR_POP_ASSUM_LIST(MP_TAC o end_itlist CONJ o DESUM_RULE) THEN
DISCH_THEN(fun th -> REWRITE_TAC[th]) THEN REAL_ARITH_TAC
```

The approach just proves $y' \equiv y \pmod{2^b}$ by proving that $(y' - y)/2^b \in \mathbb{Z}$

Sometimes one needs to combine with the other approach for carries in lower bit positions ...

Custom automation: irrelevance of modulus multiples

Exactly the same proof automation works for moduli other than powers of 2, proving $y' \equiv y \pmod{m}$ by proving that $(y' - y)/m \in \mathbb{Z}$.

Custom automation: irrelevance of modulus multiples

Exactly the same proof automation works for moduli other than powers of 2, proving $y' \equiv y \pmod{m}$ by proving that $(y' - y)/m \in \mathbb{Z}$.

```
|- nonoverlapping (word pc,0x242) (z,8 * 4)
   ==> ensures x86
        (\s. bytes_loaded s (word pc) (BUTLAST bignum_montmul_p256_mc) /\
             read RIP s = word(pc + 0x09) /\
             C_ARGUMENTS [z; x; y] s /\
             bignum_from_memory (x,4) s = a /\
             bignum_from_memory (y,4) s = b)
        (\s. read RIP s = word (pc + 0x238) /\
             (a * b <= 2 EXP 256 * p_256
              ==> bignum_from_memory (z,4) s =
                  (inverse_mod p_256 (2 EXP 256) * a * b) MOD p_256))
        (MAYCHANGE [RIP; RAX; RBX; RCX; RDX;
                    R8; R9; R10; R11; R12; R13; R14; R15] ,,
         MAYCHANGE [memory :> bytes(z,8 * 4)] ,,
         MAYCHANGE SOME_FLAGS)
```

Questions?