Verified Computer Arithmetic for Cryptography and Elsewhere

John Harrison
Amazon Web Services

ARITH 2023

Mon 4th Sep 2023 (11:30–12:30)

s2n-bignum

An open-source library of bignum arithmetic operations designed for cryptographic applications.

s2n-bignum

An open-source library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: hand-crafted code with competitive performance

s2n-bignum

An open-source library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: hand-crafted code with competitive performance
- Correct: every function is formally verified mathematically

s2n-bignum

An open-source library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: hand-crafted code with competitive performance
- Correct: every function is formally verified mathematically
- Secure: all code is written in “constant-time” style

s2n-bignum

An open-source library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: hand-crafted code with competitive performance
- Correct: every function is formally verified mathematically
- Secure: all code is written in “constant-time” style

`https://github.com/awslabs/s2n-bignum`

s2n-bignum

An open-source library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: hand-crafted code with competitive performance
- Correct: every function is formally verified mathematically
- Secure: all code is written in “constant-time” style

`https://github.com/awslabs/s2n-bignum`

All hand-written or specially generated 64-bit ARM and x86 machine code.

Open-source cryptography at AWS

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

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

Open-source cryptography at AWS

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

- `libtls`: Transport layer security → s2n-tls
  `https://github.com/aws/s2n-tls`
- `libcrypto`: Cryptography

Open-source cryptography at AWS

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

- `libtls`: Transport layer security → s2n-tls

  `https://github.com/aws/s2n-tls`

- `libcrypto`: Cryptography → aws-lc

  `https://github.com/aws/aws-lc`

Open-source cryptography at AWS

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

- `libtls`: Transport layer security → s2n-tls

  `https://github.com/aws/s2n-tls`

- `libcrypto`: Cryptography → aws-lc → s2n-bignum

  `https://github.com/aws/aws-lc`

Open-source cryptography at AWS

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

- `libtls`: Transport layer security → s2n-tls

  `https://github.com/aws/s2n-tls`

- `libcrypto`: Cryptography → aws-lc → s2n-bignum

  `https://github.com/aws/aws-lc`

Bignum arithmetic is fundamental in crypto algorithms like RSA, ECDH, ECDSA, ... Mainly modular operations, with odd modulus.

2006: Verifying floating-point arithmetic at Intel

From arithmetic on $\mathbb{R}$ to arithmetic on $\mathbb{Z}$

Floating-point kernels v cryptographic primitives

- They are *both* intended to be mathematically correct (give the right answer or 'within 0.52 ulps')

Floating-point kernels v cryptographic primitives

- They are *both* intended to be mathematically correct (give the right answer or ‘within 0.52 ulps’)
- They are *both* intended to be fast

Floating-point kernels v cryptographic primitives

- They are *both* intended to be mathematically correct (give the right answer or 'within 0.52 ulps')
- They are *both* intended to be fast
- Crypto bignums often need to be *constant-time* to avoid timing side-channels

Plan for the talk

- Side channels and "constant-time" code
- `s2n-bignum` design and implementation
- `s2n-bignum` formal verification
- Comparison with floating-point numerics

Side channels and "constant-time" code

Cyber-attacks

Attack on the Chappe semaphore system in 1834:

See Tom Standage "The Crooked Timber of Humanity":
`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 $\leftarrow$
- Memory access pattern $\leftarrow$
- 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;
```

Another motivation for working directly in machine code where flags and useful instructions like `CMOV` and `CSEL` are available.

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

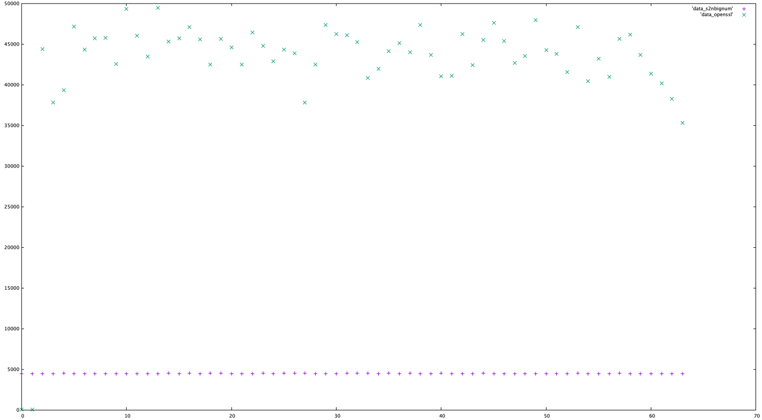
Are the machine instructions constant-time?

- Some definitely not, e.g. division by zero is special
- General assumption that simple things like add, mul mostly are

Recently CPUs have started offering *some* guarantees (DIT bit or DOIT mode).

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.



`s2n-bignum`
design and implementation

Design overview

Typical design questions and tradeoffs:

Design overview

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?

Design overview

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated

Design overview

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated
- Multiplication: schoolbook, Karatsuba, NTT, ...?

Design overview

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated
- Multiplication: schoolbook, Karatsuba, NTT, ...?
  Mix of schoolbook and Karatsuba

Design overview

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated
- Multiplication: schoolbook, Karatsuba, NTT, . . . ?
  Mix of schoolbook and Karatsuba
- Modular reduction: traditional or Montgomery?

Design overview

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated
- Multiplication: schoolbook, Karatsuba, NTT, . . . ?
  Mix of schoolbook and Karatsuba
- Modular reduction: traditional or Montgomery?
  Montgomery except for some special moduli

Design overview

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated
- Multiplication: schoolbook, Karatsuba, NTT, . . . ?
  Mix of schoolbook and Karatsuba
- Modular reduction: traditional or Montgomery?
  Montgomery except for some special moduli
- Use/avoid special instructions and ISA features?

Design overview

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated
- Multiplication: schoolbook, Karatsuba, NTT, ...?
  Mix of schoolbook and Karatsuba
- Modular reduction: traditional or Montgomery?
  Montgomery except for some special moduli
- Use/avoid special instructions and ISA features?
  Occasional use, mostly limited palette and little SIMD

Design overview

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated
- Multiplication: schoolbook, Karatsuba, NTT, ...?
  Mix of schoolbook and Karatsuba
- Modular reduction: traditional or Montgomery?
  Montgomery except for some special moduli
- Use/avoid special instructions and ISA features?
  Occasional use, mostly limited palette and little SIMD
- Specialize for particular microarchitectures?

Design overview

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated
- Multiplication: schoolbook, Karatsuba, NTT, ...?
  Mix of schoolbook and Karatsuba
- Modular reduction: traditional or Montgomery?
  Montgomery except for some special moduli
- Use/avoid special instructions and ISA features?
  Occasional use, mostly limited palette and little SIMD
- Specialize for particular microarchitectures?
  Many functions have two variants for different uarchs

Architecture matters

Recent x86 chips support `MULX`, `ADCX` and `ADOX` instructions specifically designed for integer multiplication, often giving around a 1.3X speedup versus traditional `MUL`/`ADD`/`ADC`.

Architecture matters

Recent x86 chips support `MULX`, `ADCX` and `ADOX` instructions specifically designed for integer multiplication, often giving around a 1.3X speedup versus traditional `MUL`/`ADD`/`ADC`.

Ozturk, Guilford, Gopal and Feghali, *New Instructions Supporting Large Integer Arithmetic on Intel® Architecture Processors*, 2012

Architecture matters

Recent x86 chips support `MULX`, `ADCX` and `ADOX` instructions specifically designed for integer multiplication, often giving around a 1.3X speedup versus traditional `MUL`/`ADD`/`ADC`.

Ozturk, Guilford, Gopal and Feghali, *New Instructions Supporting Large Integer Arithmetic on Intel® Architecture Processors*, 2012

Hence different `s2n-bignum` function variants:

- `bignum_mul_4_8` - 256 $\times$ 256 bit multiplication using new instructions (better performance on recent CPUs)
- `bignum_mul_4_8_alt` - identical functionality using traditional operations (compatibility for older CPUs)

Microarchitecture matters

Even with the same instructions, different ARM®v8 architecture CPUs have significantly different microarchitectural characteristics, in particular throughput of `UMULH`.

Microarchitecture matters

Even with the same instructions, different ARM®v8 architecture CPUs have significantly different microarchitectural characteristics, in particular throughput of `UMULH`.

- `bignum_mul_4_8` - 256 $\times$ 256 bit multiplication using two layers of Karatsuba reduction
- `bignum_mul_4_8_alt` - identical functionality using pure schoolbook multiplication.

Microarchitecture matters

Even with the same instructions, different ARM®v8 architecture CPUs have significantly different microarchitectural characteristics, in particular throughput of `UMULH`.

- `bignum_mul_4_8` - 256×256 bit multiplication using two layers of Karatsuba reduction
- `bignum_mul_4_8_alt` - identical functionality using pure schoolbook multiplication.

Performance ratio on various CPUs can be almost 2X, but not always in the same direction!

Asymptotically efficient multiplication algorithms

- Schoolbook is $O(n^2)$
- Karatsuba is $O(n^{1.5849...})$
- NTT is $O(n \log n \log \log n)$

Asymptotically efficient multiplication algorithms

- Schoolbook is $O(n^2)$
- Karatsuba is $O(n^{1.5849\ldots})$
- NTT is $O(n \log n \log \log n)$

For example (subtractive) Karatsuba trades a multiplication for more additions

$$(Bx_1+x_0)(By_1+y_0) = B^2x_1y_1+B((x_1-x_0)(y_0-y_1)+x_1y_1+x_0y_0)+x_0y_0$$

When do they become worthwhile?

Especially on microarchitectures that have lower multiplier throughputs, these fancier algorithms are more practical than you might think.

When do they become worthwhile?

Especially on microarchitectures that have lower multiplier throughputs, these fancier algorithms are more practical than you might think.

- Karatsuba may be worthwhile even for sizes like 64 or 128 bits. See Liu, Järvinen, Liu and Seo, *Multiprecision Multiplication on ARMv8* in ARITH 2017

When do they become worthwhile?

Especially on microarchitectures that have lower multiplier throughputs, these fancier algorithms are more practical than you might think.

- Karatsuba may be worthwhile even for sizes like 64 or 128 bits. See Liu, Järvinen, Liu and Seo, *Multiprecision Multiplication on ARMv8* in ARITH 2017
- "Arbitrary degree Karatsuba" (ADK) can subdivide size by any factor (e.g. $192 \to 3 \times 64$). See Mike Scott: *Missing a trick: Karatsuba variations*, 2015.

When do they become worthwhile?

Especially on microarchitectures that have lower multiplier throughputs, these fancier algorithms are more practical than you might think.

- Karatsuba may be worthwhile even for sizes like 64 or 128 bits. See Liu, Järvinen, Liu and Seo, *Multiprecision Multiplication on ARMv8* in ARITH 2017
- "Arbitrary degree Karatsuba" (ADK) can subdivide size by any factor (e.g. $192 \rightarrow 3 \times 64$). See Mike Scott: *Missing a trick: Karatsuba variations*, 2015.
- NTT may already be competitive for the sizes typically used in RSA (1024-4096 bits). See Becker, Hwang, Kannwischer, Panny and Yang, *Efficient Multiplication of Somewhat Small Integers using Number-Theoretic Transforms*, IWSEC 2022.

Vector instructions

Many modern CPUs feature SIMD operations that potentially offer higher operation throughputs, e.g. ARM® NEON™ and Intel® AVX2.

Vector instructions

Many modern CPUs feature SIMD operations that potentially offer higher operation throughputs, e.g. ARM® NEON$^{\text{TM}}$ and Intel® AVX2.

- SIMD often works well in unsaturated contexts, good for some moduli like $2^{255} - 19$ or NTT-type approaches.

Vector instructions

Many modern CPUs feature SIMD operations that potentially offer higher operation throughputs, e.g. ARM® NEON™ and Intel® AVX2.

- SIMD often works well in unsaturated contexts, good for some moduli like $2^{255} - 19$ or NTT-type approaches.
- Fine-grained interleaving can share work between scalar and vector units. Juneyoung Lee's recent improvements to s2n-bignum.

Vector instructions

Many modern CPUs feature SIMD operations that potentially offer higher operation throughputs, e.g. ARM® NEON$^{\mathrm{TM}}$ and Intel® AVX2.

- SIMD often works well in unsaturated contexts, good for some moduli like $2^{255} - 19$ or NTT-type approaches.
- Fine-grained interleaving can share work between scalar and vector units. Juneyoung Lee's recent improvements to s2n-bignum.
- Coarse-grained interleaving may help when there is parallelism in the toplevel operations. See Emil Lenngren, *AArch64 optimized implementation for X25519*, 2019.

Vector instructions

Many modern CPUs feature SIMD operations that potentially offer higher operation throughputs, e.g. ARM® NEON™ and Intel® AVX2.

- SIMD often works well in unsaturated contexts, good for some moduli like $2^{255} - 19$ or NTT-type approaches.
- Fine-grained interleaving can share work between scalar and vector units. Juneyoung Lee's recent improvements to s2n-bignum.
- Coarse-grained interleaving may help when there is parallelism in the toplevel operations. See Emil Lenngren, *AArch64 optimized implementation for X25519*, 2019.
- With many SIMD lanes and fairly wide multiplies, these may finally outperform scalar multipliers overall, e.g. `ifma`.

Some performance improvement numbers

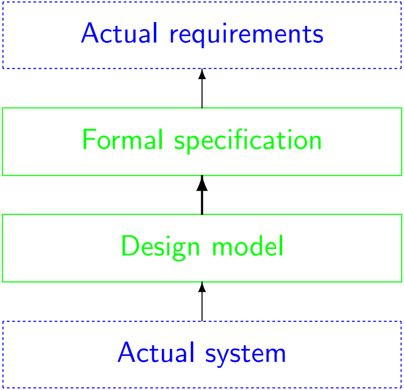
Integrating X25519-related functions from `s2n-bignum` into `aws-lc`, versus code previously used (close to BoringSSL).

AWS® Graviton 2	2.13x
AWS® Graviton 3	1.57x
Apple® M1	1.73x
Modern Intel® and AMD® x86	1.75x - 1.85x
Intel® x86 "Haswell"	1.27x

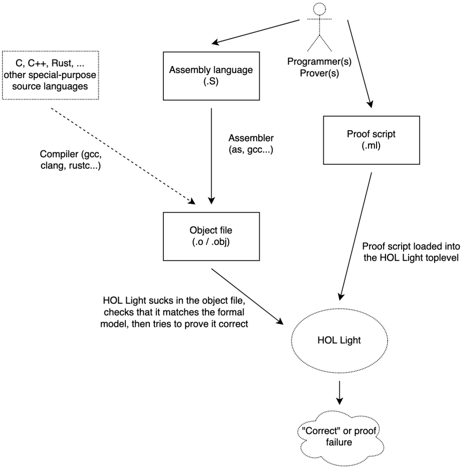
`s2n-bignum`
formal verification

Formal verification

Using a (machine-checked) mathematical proof to verify that an implementation satisfies its mathematical specification.



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 ; .....
```

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 ; .....
```

Run in continuous integration on any github pull request

Modeling instruction decoding and execution

Decoding instruction byte sequences to their semantics:

```
...
| [0b1101011001011111000000:22; 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:

```
...
| [0b1101011001011111000000:22; 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 be nondeterministic (might not be 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 methods:

- Machine code Hoare logic: Myreen, Fox and Gordon, *Hoare Logic for ARM machine code*, FSEN 2007
- Symbolic simulation: Dockins, Folzer, Hendrix, Huffman, McNamee and Tomb, *Constructing Semantic Models of Programs with the Software Analysis Workbench*, VSTTE 2014.

Hoare logic + Symbolic simulation

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

- Machine code Hoare logic: Myreen, Fox and Gordon, *Hoare Logic for ARM machine code*, FSEN 2007
- Symbolic simulation: Dockins, Folzer, Hendrix, Huffman, McNamee and Tomb, *Constructing Semantic Models of Programs with the Software Analysis Workbench*, VSTTE 2014.

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)
```

Proof sizes / Annotation ratio

Stripping out copies of the code from inside the proofs:

	ARM	x86
Lines of source code	98263	79153
Lines of proof	142643	130393
Bytes of source code	3441575	2677243
Bytes of proof	7581703	6888788
Bytes of machine code	751128	657388

Proof sizes / Annotation ratio

Stripping out copies of the code from inside the proofs:

	ARM	x86
Lines of source code	98263	79153
Lines of proof	142643	130393
Bytes of source code	3441575	2677243
Bytes of proof	7581703	6888788
Bytes of machine code	751128	657388

Compare the 'de Bruijn factor' of formalized mathematics

Comparison with floating-point numerics

A contrast: MSB versus LSB

Floating-point numbers, usually being normalized, naturally lend themselves to algorithms based on the ‘most significant bit’.

A contrast: MSB versus LSB

Floating-point numbers, usually being normalized, naturally lend themselves to algorithms based on the 'most significant bit'.

In cryptography, even identifying and manipulating MSBs can be awkward or unnatural to do in constant time.

A contrast: MSB versus LSB

Floating-point numbers, usually being normalized, naturally lend themselves to algorithms based on the ‘most significant bit’.

In cryptography, even identifying and manipulating MSBs can be awkward or unnatural to do in constant time.

More usage of LSB-based algorithms such as *Montgomery multiplication*.

Peter Montgomery

At ARITH 2009 on the cruise:

Montgomery reduction

Montgomery reduction is essentially division by a power of 2, modulo an odd number m.

Montgomery reduction

Montgomery reduction is essentially division by a power of 2, modulo an odd number m.

- Conventional modular reduction: $x = qm + r$ (cancel leading bits by subtracting multiples of m)
  $hhh\cdots hhh\ lll\cdots lll \to 000\cdots 000\ rrr\cdots rrr \to rrr\cdots rrr$

Montgomery reduction

Montgomery reduction is essentially division by a power of 2, modulo an odd number m.

- Conventional modular reduction: $x = qm + r$ (cancel leading bits by subtracting multiples of m)
  $hhh \cdots hhh\ lll \cdots lll \to 000 \cdots 000\ rrr \cdots rrr \to rrr \cdots rrr$
- Montgomery reduction: $x = qm + 2^{\alpha} s$ (cancel *trailing* bits by adding multiples of m)
  $hhh \cdots hhh\ lll \cdots lll \to sss \cdots sss\ 000 \cdots 000 \to sss \cdots sss$

Montgomery reduction

Montgomery reduction is essentially division by a power of 2, modulo an odd number m.

- Conventional modular reduction: $x = qm + r$ (cancel leading bits by subtracting multiples of m)
  $hhh \cdots hhh\ lll \cdots lll \rightarrow 000 \cdots 000\ rrr \cdots rrr \rightarrow rrr \cdots rrr$
- Montgomery reduction: $x = qm + 2^{\alpha} s$ (cancel *trailing* bits by adding multiples of m)
  $hhh \cdots hhh\ lll \cdots lll \rightarrow sss \cdots sss\ 000 \cdots 000 \rightarrow sss \cdots sss$

Can then keep integers systematically in the 'Montgomery domain', multiplied by 2^{α} modulo m and do Montgomery multiplications (multiplication + Montgomery reduction).

An *analogy*: MSB versus LSB

There are meaningful analogies between 'metrical' and 'p-adic' algorithms:

- Over $\mathbb{R}$ where *things get smaller*
- Over $\mathbb{Z}$ where *things get more divisible by* p

An *analogy*: MSB versus LSB

There are meaningful analogies between 'metrical' and 'p-adic' algorithms:

- Over $\mathbb{R}$ where *things get smaller*
- Over $\mathbb{Z}$ where *things get more divisible by* p

One can explicitly cast the latter as a metric, and perform metric space completion to get the 'p-adic numbers'.

An *analogy*: MSB versus LSB

There are meaningful analogies between ‘metrical’ and ‘p-adic’ algorithms:

- Over $\mathbb{R}$ where *things get smaller*
- Over $\mathbb{Z}$ where *things get more divisible by* p

One can explicitly cast the latter as a metric, and perform metric space completion to get the ‘p-adic numbers’.

Brent and Zimmermann, *Modern Computer Arithmetic*, Table 2.1:

classical (MSB)	p-adic (LSB)
Euclidean division	Hensel division, Montgomery reduction
Svoboda’s algorithm	Montgomery-Svoboda
Euclidean gcd	Binary gcd
Newton’s method	Hensel lifting

Using Newton's method for reciprocals

Floating-point computation of $1/a$:

- Form initial approximation $y \approx \frac{1}{a}$
- Then iterate $y' = y \cdot (2 - ay) = y + y \cdot (1 - ay)$

Using Newton's method for reciprocals

Floating-point computation of $1/a$:

- Form initial approximation $y \approx \frac{1}{a}$
- Then iterate $y' = y \cdot (2 - ay) = y + y \cdot (1 - ay)$

If $y = \frac{1}{a}(1 + \epsilon)$ then $y' = \frac{1}{a}(1 - \epsilon^2)$, the classic quadratic convergence where we get twice as many bits of accuracy per iteration.

Modular inverses by Hensel lifting

Consider the 1-word (negated) modular inverse, called `word_negmodinv` in `s2n-bignum`.

Modular inverses by Hensel lifting

Consider the 1-word (negated) modular inverse, called `word_negmodinv` in `s2n-bignum`.

Given a 64-bit unsigned and *odd* integer a, returns another 64-bit integer x such that $ax \equiv -1 \pmod{2^{64}}$, i.e. that `a * x == 0xFFFFFFFFFFFFFFFF` using unsigned silently-wrapping word operations like those on C's `uint64_t`.

Modular inverses by Hensel lifting

Consider the 1-word (negated) modular inverse, called `word_negmodinv` in `s2n-bignum`.

Given a 64-bit unsigned and *odd* integer a, returns another 64-bit integer x such that $ax \equiv -1 \pmod{2^{64}}$, i.e. that `a * x == 0xFFFFFFFFFFFFFFFF` using unsigned silently-wrapping word operations like those on C's `uint64_t`.

It is implemented in a directly similar way using Hensel lifting, the p-adic analog of Newton's method.

Initial approximation

As with the floating-point inverse, we need an initial approximation to start with. The following piece of magic (in C syntax):

```
x = (a - (a<<2))^2
```

happens to give a 5-bit negated modular inverse, assuming `a` is odd.

Hensel lifting step

Given a k-bit approximation $ax \equiv -1 \pmod{2^k}$, do the same Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

Hensel lifting step

Given a k-bit approximation $ax \equiv -1 \pmod{2^k}$, do the same Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

By the initial assumption $ax = 2^k n - 1$ for some integer n

Hensel lifting step

Given a k-bit approximation $ax \equiv -1 \pmod{2^k}$, do the same Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

By the initial assumption $ax = 2^k n - 1$ for some integer n
So $e = ax + 1 = 2^k n$ and

Hensel lifting step

Given a k-bit approximation $ax \equiv -1 \pmod{2^k}$, do the same Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

By the initial assumption $ax = 2^k n - 1$ for some integer n
So $e = ax + 1 = 2^k n$ and $e + 1 = 2^k n + 1$

Hensel lifting step

Given a k-bit approximation $ax \equiv -1 \pmod{2^k}$, do the same Newton step with integers, except for a sign flip because we want a *negated* inverse:

```
e = a * x + 1;
y = e * x + x;
```

By the initial assumption $ax = 2^k n - 1$ for some integer n
So $e = ax + 1 = 2^k n$ and $e + 1 = 2^k n + 1$ so then
$ay = ax(e + 1) = (2^k n - 1)(2^k n + 1) = 2^{2k} n^2 - 1$, i.e.
$ay \equiv -1 \pmod{2^{2k}}$.

The same analogies in verification

Linear (Presburger) arithmetic is a long-established workhorse in program verification.

$$|x - x'| \leq e/2 \wedge |y - y'| < e/2 \Rightarrow |(x + y) - (x' + y')| < e$$

The same analogies in verification

Linear (Presburger) arithmetic is a long-established workhorse in program verification.

$$|x - x'| \leq e/2 \wedge |y - y'| < e/2 \Rightarrow |(x + y) - (x' + y')| < e$$

For a lot of the 'congruential' reasoning a custom decision procedure is a similarly useful workhorse:

$$\begin{array}{l} \mathrm{coprime}(d, a) \wedge \mathrm{coprime}(d, b) \Rightarrow \mathrm{coprime}(d, ab) \\ ax \equiv ay \pmod{n} \wedge \mathrm{coprime}(a, n) \Rightarrow x \equiv y \pmod{n} \\ \mathrm{gcd}(a, n) \mid b \Rightarrow \exists x.\ ax \equiv b \pmod{n} \end{array}$$

The same analogies in verification

Linear (Presburger) arithmetic is a long-established workhorse in program verification.

$$|x - x'| \leq e/2 \wedge |y - y'| < e/2 \Rightarrow |(x + y) - (x' + y')| < e$$

For a lot of the 'congruential' reasoning a custom decision procedure is a similarly useful workhorse:

$$\begin{array}{l} \mathrm{coprime}(d, a) \wedge \mathrm{coprime}(d, b) \Rightarrow \mathrm{coprime}(d, ab) \\ ax \equiv ay \pmod{n} \wedge \mathrm{coprime}(a, n) \Rightarrow x \equiv y \pmod{n} \\ \gcd(a, n) \mid b \Rightarrow \exists x.\ ax \equiv b \pmod{n} \end{array}$$

See Harrison, *Automating elementary number-theoretic proofs using Gröbner bases*, CADE21.

Other verification similarities

Other verification similarities

- Value of general theorem proving framework for reasoning, and even for stating the specification

Other verification similarities

- Value of general theorem proving framework for reasoning, and even for stating the specification
- Specifications are clear and mathematical, without much ambiguity

Other verification similarities

- Value of general theorem proving framework for reasoning, and even for stating the specification
- Specifications are clear and mathematical, without much ambiguity
- Requirement for special-purpose inference rules (e.g. combining interval reasoning and algebra).

Other verification similarities

- Value of general theorem proving framework for reasoning, and even for stating the specification
- Specifications are clear and mathematical, without much ambiguity
- Requirement for special-purpose inference rules (e.g. combining interval reasoning and algebra).
- Can more confidently adopt sophisticated algorithms and subtle optimizations thanks to FV

Other verification similarities

- Value of general theorem proving framework for reasoning, and even for stating the specification
- Specifications are clear and mathematical, without much ambiguity
- Requirement for special-purpose inference rules (e.g. combining interval reasoning and algebra).
- Can more confidently adopt sophisticated algorithms and subtle optimizations thanks to FV
- Typically, experts in both fields can appreciate the meaning and value of formal verification.

Questions?