# The s2n-bignum library of formally verified ECC software

John Harrison
Amazon Web Services

ECC 2024 (remotely)

Wed 30th Oct 2024 (15:30–16:30 in Taipei)

# 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`

Integrated into AWS's open-source cryptographic library that provides top-level operations:

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

# What's in s2n-bignum?

- Basic operations on 64-bit words like counting leading zeros
- Generic-size bignum arithmetic operations like multiplication
- Generic-size data manipulation like constant-time selection
- Generic-size Montgomery operations for any odd modulus

# What's in s2n-bignum?

- Basic operations on 64-bit words like counting leading zeros
- Generic-size bignum arithmetic operations like multiplication
- Generic-size data manipulation like constant-time selection
- Generic-size Montgomery operations for any odd modulus
- Fixed-size ECC operations (for curve25519, edwards25519, P-256, P-384, P-521, secp256k1, SM2).

# What's in s2n-bignum?

- Basic operations on 64-bit words like counting leading zeros
- Generic-size bignum arithmetic operations like multiplication
- Generic-size data manipulation like constant-time selection
- Generic-size Montgomery operations for any odd modulus
- Fixed-size ECC operations (for curve25519, edwards25519, P-256, P-384, P-521, secp256k1, SM2).
  - ECC field operations (i.e. efficient modular arithmetic)
  - ECC group operations (addition and doubling for most curves, Montgomery ladder step for curve25519)
  - Top-level scalar multiplications

# Plan for the talk

- Approach and philosophy
- Design and implementation
- Formal verification
- “Constant time”

# Approach and philosophy

# Notable s2n-bignum characteristics

There are many excellent ‘formally verified cryptography’ projects. Why s2n-bignum and what is distinctive about it?

# Notable s2n-bignum characteristics

There are many excellent 'formally verified cryptography' projects. Why s2n-bignum and what is distinctive about it?

- Generic-size bignum functions with constant-time API

# Notable s2n-bignum characteristics

There are many excellent 'formally verified cryptography' projects. Why s2n-bignum and what is distinctive about it?

- Generic-size bignum functions with constant-time API
- Pure machine code, optimized mechanically and/or by hand

# Notable s2n-bignum characteristics

There are many excellent 'formally verified cryptography' projects. Why s2n-bignum and what is distinctive about it?

- Generic-size bignum functions with constant-time API
- Pure machine code, optimized mechanically and/or by hand
- Verification is separate, not connected to the coding process

# Notable s2n-bignum characteristics

There are many excellent 'formally verified cryptography' projects. Why s2n-bignum and what is distinctive about it?

- Generic-size bignum functions with constant-time API
- Pure machine code, optimized mechanically and/or by hand
- Verification is separate, not connected to the coding process
- Verification takes place in a single general theorem prover

# Constant-time API for generic functions

Basic arithmetic operations have generic-size implementations, where all inputs and outputs can have any number of 64-bit digits.

# Constant-time API for generic functions

Basic arithmetic operations have generic-size implementations, where all inputs and outputs can have any number of 64-bit digits.

OpenSSL and other common libraries offer this functionality with a *simple* and *convenient* interface

```
int BN_mul(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);
```

# Constant-time API for generic functions

Basic arithmetic operations have generic-size implementations, where all inputs and outputs can have any number of 64-bit digits.

OpenSSL and other common libraries offer this functionality with a *simple* and *convenient* interface

```
int BN_mul(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);
```

but as as result it's very difficult (impossible?) to make code *really* constant-time.

# Constant-time API for generic functions

Basic arithmetic operations have generic-size implementations, where all inputs and outputs can have any number of 64-bit digits.

OpenSSL and other common libraries offer this functionality with a *simple* and *convenient* interface

```
int BN_mul(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);
```

but as as result it's very difficult (impossible?) to make code *really* constant-time.

s2n-bignum has a low-level API making all the sizes explicit:

```
void bignum_mul(uint64_t p,uint64_t *z,
                uint64_t m,uint64_t *x,
                uint64_t n,uint64_t *y);
```

# Constant-time API for generic functions

Basic arithmetic operations have generic-size implementations, where all inputs and outputs can have any number of 64-bit digits.

OpenSSL and other common libraries offer this functionality with a *simple* and *convenient* interface

```
int BN_mul(BIGNUM *r, BIGNUM *a, BIGNUM *b, BN_CTX *ctx);
```

but as as result it's very difficult (impossible?) to make code *really* constant-time.

s2n-bignum has a low-level API making all the sizes explicit:

```
void bignum_mul(uint64_t p,uint64_t *z,
                uint64_t m,uint64_t *x,
                uint64_t n,uint64_t *y);
```

The sequence of instructions (hence timing) depends only on the *nominal* sizes.

# 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

# 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 . . .

# Correct-by-construction or separate verification?

Formally verified cryptography projects can be placed on this spectrum:

# Correct-by-construction or separate verification?

Formally verified cryptography projects can be placed on this spectrum:

- Correct-by-construction coding (HACL*, Jasmin)
- ...
- A bit of both or somewhere in between (Fiat)
- ...
- Separate verification (CryptoLine, s2n-bignum)

# Pros and cons of the s2n-bignum approach

# Pros and cons of the s2n-bignum approach

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

# Pros and cons of the s2n-bignum approach

- ☺ 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.

# Pros and cons of the s2n-bignum approach

- ☺ 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.

# Pros and cons of the s2n-bignum approach

- ☺ 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.

# Pros and cons of the s2n-bignum approach

- ☺ 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.

# Verification in a single interactive prover

There is a broad division in theorem proving into:

- Automated theorem provers — try to prove a conjecture without human guidance
  - SAT = propositional satisfiability (Cadical, MiniSat, . . . )
  - First-order logic with quantifiers (Vampire, E, . . . )
  - Equational logic (Waldmeister, . . . )
  - SMT = 'satisfiability modulo theories' (z3, CVC5, . . . )

# Verification in a single interactive prover

There is a broad division in theorem proving into:

- Automated theorem provers — try to prove a conjecture without human guidance
  - SAT = propositional satisfiability (Cadical, MiniSat, ...)
  - First-order logic with quantifiers (Vampire, E, ...)
  - Equational logic (Waldmeister, ...)
  - SMT = 'satisfiability modulo theories' (z3, CVC5, ...)
- Interactive theorem provers — some automation but also human guidance and reliable, checkable proof process (ACL2, Coq, HOL, Isabelle/HOL, Lean, ...)

# Verification in a single interactive prover

There is a broad division in theorem proving into:

- Automated theorem provers — try to prove a conjecture without human guidance
  - SAT = propositional satisfiability (Cadical, MiniSat, ...)
  - First-order logic with quantifiers (Vampire, E, ...)
  - Equational logic (Waldmeister, ...)
  - SMT = 'satisfiability modulo theories' (z3, CVC5, ...)
- Interactive theorem provers — some automation but also human guidance and reliable, checkable proof process (ACL2, Coq, HOL, Isabelle/HOL, Lean, ...)

s2n-bignum proofs all take place in the HOL Light interactive theorem prover, which has a simple logical kernel but is fully programmable with custom inference rules.

# Design and implementation

# s2n-bignum design and implementation

Typical design questions and tradeoffs:

# s2n-bignum design and implementation

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?

# s2n-bignum design and implementation

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated except for occasional internals

# s2n-bignum design and implementation

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated except for occasional internals
- Multiplication: schoolbook, Karatsuba, NTT, . . . ?

# s2n-bignum design and implementation

Typical design questions and tradeoffs:

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

# s2n-bignum design and implementation

Typical design questions and tradeoffs:

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

# s2n-bignum design and implementation

Typical design questions and tradeoffs:

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

# s2n-bignum design and implementation

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated except for occasional internals
- 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?

# s2n-bignum design and implementation

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated except for occasional internals
- 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, mainly integer, increasing use of SIMD

# s2n-bignum design and implementation

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated except for occasional internals
- 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, mainly integer, increasing use of SIMD
- Specialize for particular microarchitectures?

# s2n-bignum design and implementation

Typical design questions and tradeoffs:

- Saturated or unsaturated number representation?
  Saturated except for occasional internals
- 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, mainly integer, increasing use of 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` - $256 \times 256$ 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!

# When is fancy multiplication worthwhile?

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

# When is fancy multiplication worthwhile?

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

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

# When is fancy multiplication worthwhile?

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

- Karatsuba can 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.

# When is fancy multiplication worthwhile?

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

- Karatsuba can 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™ 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 local interleaving can share work between scalar and vector units.

# 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 local interleaving can share work between scalar and vector units.
- Coarse-grained interleaving may help when there is parallelism in the toplevel operations (e.g. Montgomery ladder).

# 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 local interleaving can share work between scalar and vector units.
- Coarse-grained interleaving may help when there is parallelism in the toplevel operations (e.g. Montgomery ladder).
- With many SIMD lanes and fairly wide multiplies, these may finally outperform scalar multipliers overall, e.g. AVX-512 with `ifma`.

# Faster RSA on ARM using local vectorization

# Faster RSA on ARM using local vectorization

- Used subtractive Karatsuba to make the basic integer multiplications faster

# Faster RSA on ARM using local vectorization

- Used subtractive Karatsuba to make the basic integer multiplications faster
- Used a custom autovectorizer to pick possible patterns to vectorize, and automatically pick the ones that maximize measured performance.

# Faster RSA on ARM using local vectorization

- Used subtractive Karatsuba to make the basic integer multiplications faster
- Used a custom autovectorizer to pick possible patterns to vectorize, and automatically pick the ones that maximize measured performance.
- Improved constant-time table selection, also using the SIMD units.

# Faster RSA on ARM using local vectorization

- Used subtractive Karatsuba to make the basic integer multiplications faster
- Used a custom autovectorizer to pick possible patterns to vectorize, and automatically pick the ones that maximize measured performance.
- Improved constant-time table selection, also using the SIMD units.
- Used efficient symbolic equivalence checking to show equivalence to original integer code and hence correctness

# Faster RSA on ARM using local vectorization

- Used subtractive Karatsuba to make the basic integer multiplications faster
- Used a custom autovectorizer to pick possible patterns to vectorize, and automatically pick the ones that maximize measured performance.
- Improved constant-time table selection, also using the SIMD units.
- Used efficient symbolic equivalence checking to show equivalence to original integer code and hence correctness

This resulted in RSA performance gains on Graviton2 of as much as 94% compared to the original code.

# Faster RSA on ARM using local vectorization

- Used subtractive Karatsuba to make the basic integer multiplications faster
- Used a custom autovectorizer to pick possible patterns to vectorize, and automatically pick the ones that maximize measured performance.
- Improved constant-time table selection, also using the SIMD units.
- Used efficient symbolic equivalence checking to show equivalence to original integer code and hence correctness

This resulted in RSA performance gains on Graviton2 of as much as 94% compared to the original code.
June Lee, Hanno Becker and John Harrison: *Formal verification makes RSA faster — and faster to deploy*" (Amazon Science Blog).

# Slothy-optimized Lenngren X25519 code

Started from very fast hand-optimized code for ARM8 combining integer and SIMD units: *AArch64 optimized implementation for X25519*, Emil Lenngren 2019.

# Slothy-optimized Lenngren X25519 code

Started from very fast hand-optimized code for ARM8 combining integer and SIMD units: *AArch64 optimized implementation for X25519*, Emil Lenngren 2019.

- Reverse-engineered a 'clean' non-interleaved version of Lenngren's code

# Slothy-optimized Lenngren X25519 code

Started from very fast hand-optimized code for ARM8 combining integer and SIMD units: *AArch64 optimized implementation for X25519*, Emil Lenngren 2019.

- Reverse-engineered a 'clean' non-interleaved version of Lenngren's code
- Augmented the clean version with annotations to drive the formal proof

# Slothy-optimized Lenngren X25519 code

Started from very fast hand-optimized code for ARM8 combining integer and SIMD units: *AArch64 optimized implementation for X25519*, Emil Lenngren 2019.

- Reverse-engineered a 'clean' non-interleaved version of Lenngren's code
- Augmented the clean version with annotations to drive the formal proof
- Used the SLOTHY superoptimizer to re-schedule the code, preserving annotations

# Slothy-optimized Lenngren X25519 code

Started from very fast hand-optimized code for ARM8 combining integer and SIMD units: *AArch64 optimized implementation for X25519*, Emil Lenngren 2019.

- Reverse-engineered a 'clean' non-interleaved version of Lenngren's code
- Augmented the clean version with annotations to drive the formal proof
- Used the SLOTHY superoptimizer to re-schedule the code, preserving annotations
- Combined the resulting Montgomery ladder inner loop with s2n-bignum's divstep-based modular inverse.

# Slothy-optimized Lenngren X25519 code

Started from very fast hand-optimized code for ARM8 combining integer and SIMD units: *AArch64 optimized implementation for X25519*, Emil Lenngren 2019.

- Reverse-engineered a 'clean' non-interleaved version of Lenngren's code
- Augmented the clean version with annotations to drive the formal proof
- Used the SLOTHY superoptimizer to re-schedule the code, preserving annotations
- Combined the resulting Montgomery ladder inner loop with s2n-bignum's divstep-based modular inverse.

The net result is verified, 12% faster than Lenngren's original and *much* faster than the code in mainstream crypto libraries. We have many other results around the curve25519 family:

# Slothy-optimized Lenngren X25519 code

Started from very fast hand-optimized code for ARM8 combining integer and SIMD units: *AArch64 optimized implementation for X25519*, Emil Lenngren 2019.

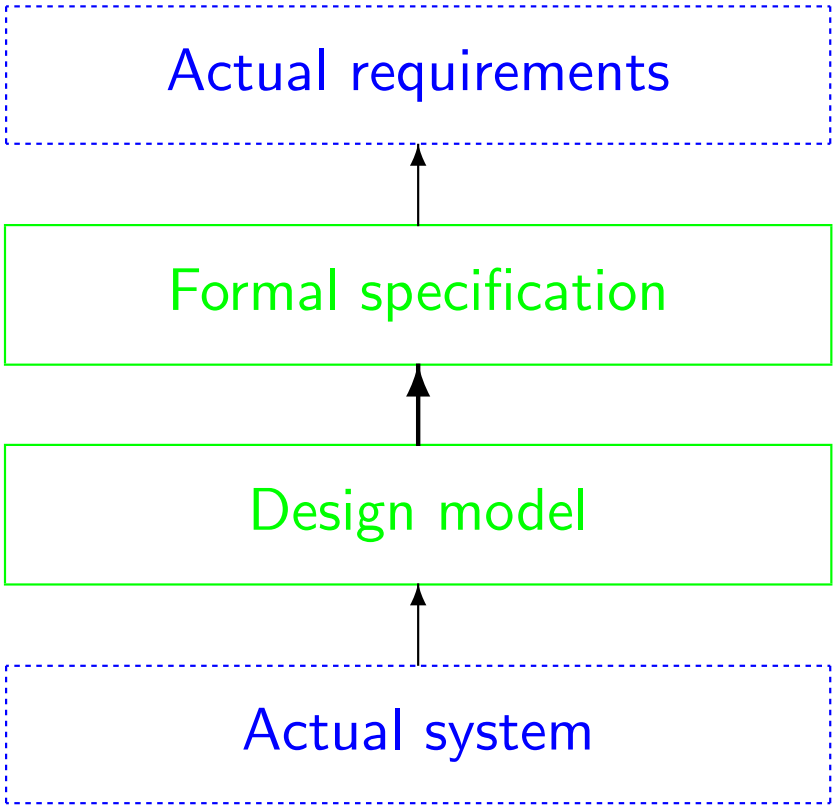
- Reverse-engineered a 'clean' non-interleaved version of Lenngren's code
- Augmented the clean version with annotations to drive the formal proof
- Used the SLOTHY superoptimizer to re-schedule the code, preserving annotations
- Combined the resulting Montgomery ladder inner loop with s2n-bignum's divstep-based modular inverse.

The net result is verified, 12% faster than Lenngren's original and *much* faster than the code in mainstream crypto libraries. We have many other results around the curve25519 family:
Torben Hansen and John Harrison: *Better-performing "25519" cryptography* (Amazon Science Blog).

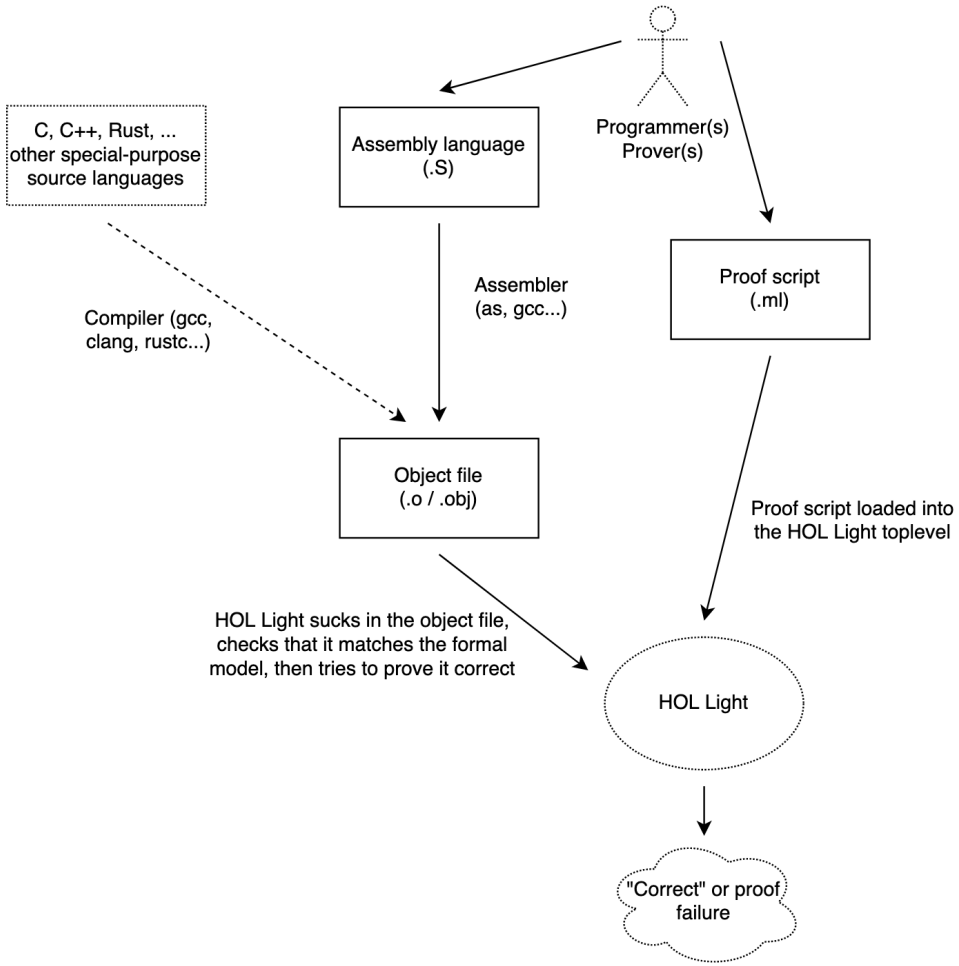
# Formal verification

# 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:

```
...
| [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 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)
```

# Symbolic setup of $6 \times 6$ multiplier

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

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

# Symbolic setup of $6 \times 6$ multiplier

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

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

```
...
 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_of_wordlist [y_0; y_1; y_2; y_3; y_4; y_5] = b`]
10 [`MAYCHANGE [] s0 s0`]
11 [`read (memory :> bytes64 x) s0 = x_0`]
...
22 [`read (memory :> bytes64 (word_add y (word 40))) s0 = y_5`]
```

# Symbolic simulation of $6 \times 6$ multiplier

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

Effectively ‘executes’ code on the *symbolic* values

# Symbolic simulation of $6 \times 6$ multiplier

```
  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]‘
```

# 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
```

# 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
```

# 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
```

In general, it's only because we interleave the bounds reasoning and algebraic simplification that the tactic is as powerful as it is.

# 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
```

# 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 $(y' - y)/2^b \in \mathbb{Z}$

# 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 . . .

# 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 $(y' - y)/m \in \mathbb{Z}$.

# 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)
```

“Constant-time”

# The usual constant-time style

When there is control flow depending on secret data:

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

# The usual constant-time style

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);
```

# Can we *prove* the constant-time property?

In contrast to functional correctness, the constant-time property is *not* yet proved formally.

# Can we *prove* the constant-time property?

In contrast to functional correctness, the constant-time property is *not* yet proved formally.

- We believe that we consistently follow the constant-time style

# Can we *prove* the constant-time property?

In contrast to functional correctness, the constant-time property is *not* yet proved formally.

- We believe that we consistently follow the constant-time style
- We avoid instructions that may have variable latency, and do some static checks to make sure

# Can we *prove* the constant-time property?

In contrast to functional correctness, the constant-time property is *not* yet proved formally.

- We believe that we consistently follow the constant-time style
- We avoid instructions that may have variable latency, and do some static checks to make sure
- In the benchmarking code we deliberately look for correlations with bit densities of input arguments.

# Can we *prove* the constant-time property?

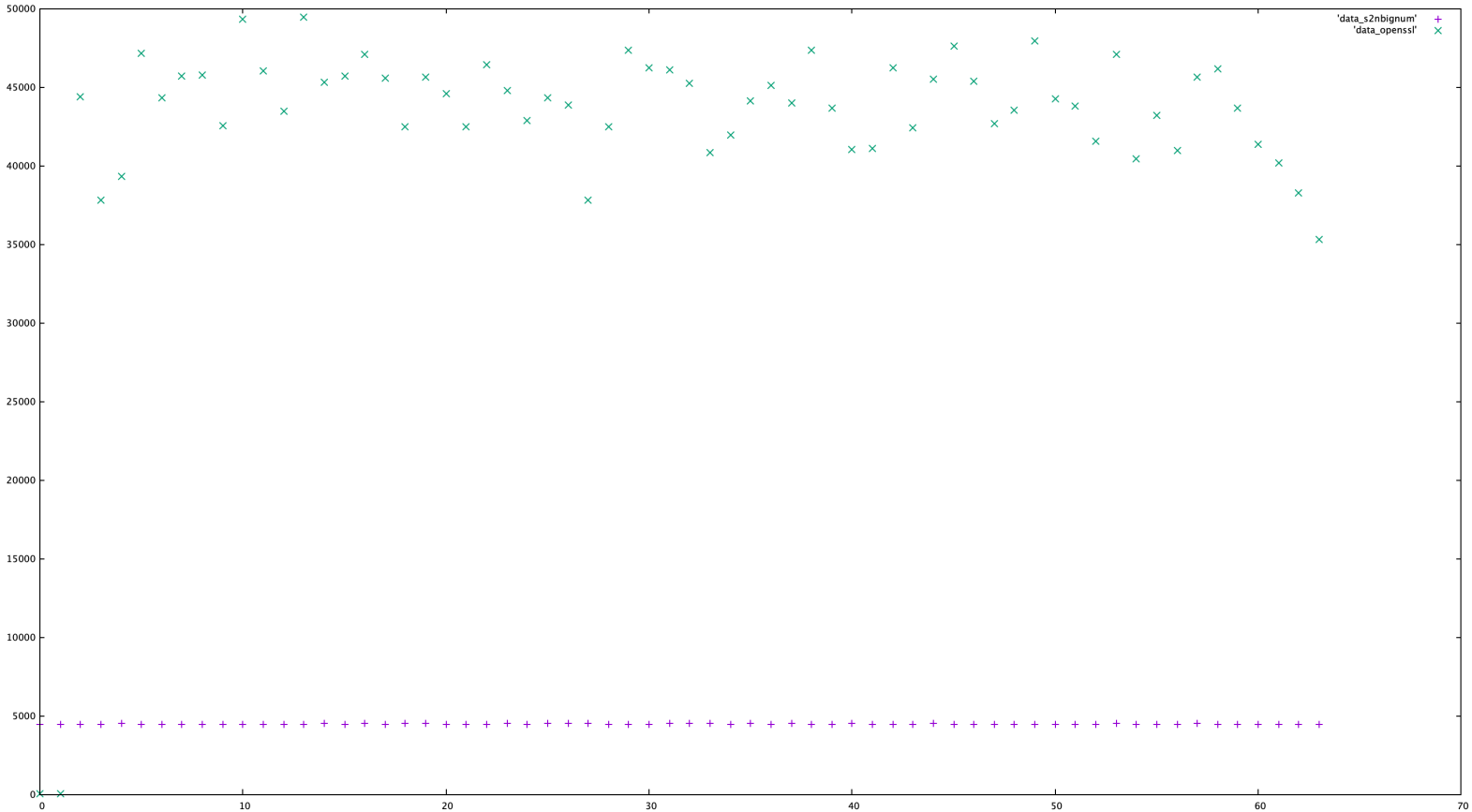
In contrast to functional correctness, the constant-time property is *not* yet proved formally.

- We believe that we consistently follow the constant-time style
- We avoid instructions that may have variable latency, and do some static checks to make sure
- In the benchmarking code we deliberately look for correlations with bit densities of input arguments.

We have work in progress exploring adding 'microarchitectural events' to the model, which could allow us to reason about constant-time properties more formally.

# 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.



Questions?