HISC 2025
HIGH INTEGRITY SOFTWARE CONFERENCE
NOV 13, 2025

Fearless optimization and Formal verification of post-quantum cryptographic code at AWS

Rod Chapman (he/him/his)

AWS Cryptography

The big ideas…

Hybrid Verification

and

Fearless Optimization

But first … some data…

MLKEM Number-Theoretic Transform (NTT)
running on Graviton2 (EC2 c6g instance) performance in clock cycles. Lower is better.

Code language/version	Cycles
PQCrystals/Kyber reference C code	3481
Rod’s very naïve first formally verified version	9110
Rod’s best formally verified and optimized version	1239
Hand-written, formally verified AArch64 assembly language (“Clean” version)	900
AArch64 assembly, auto-super-optimized (also formally verified)	808

The problem…

Cryptographic code is foundational for AWS

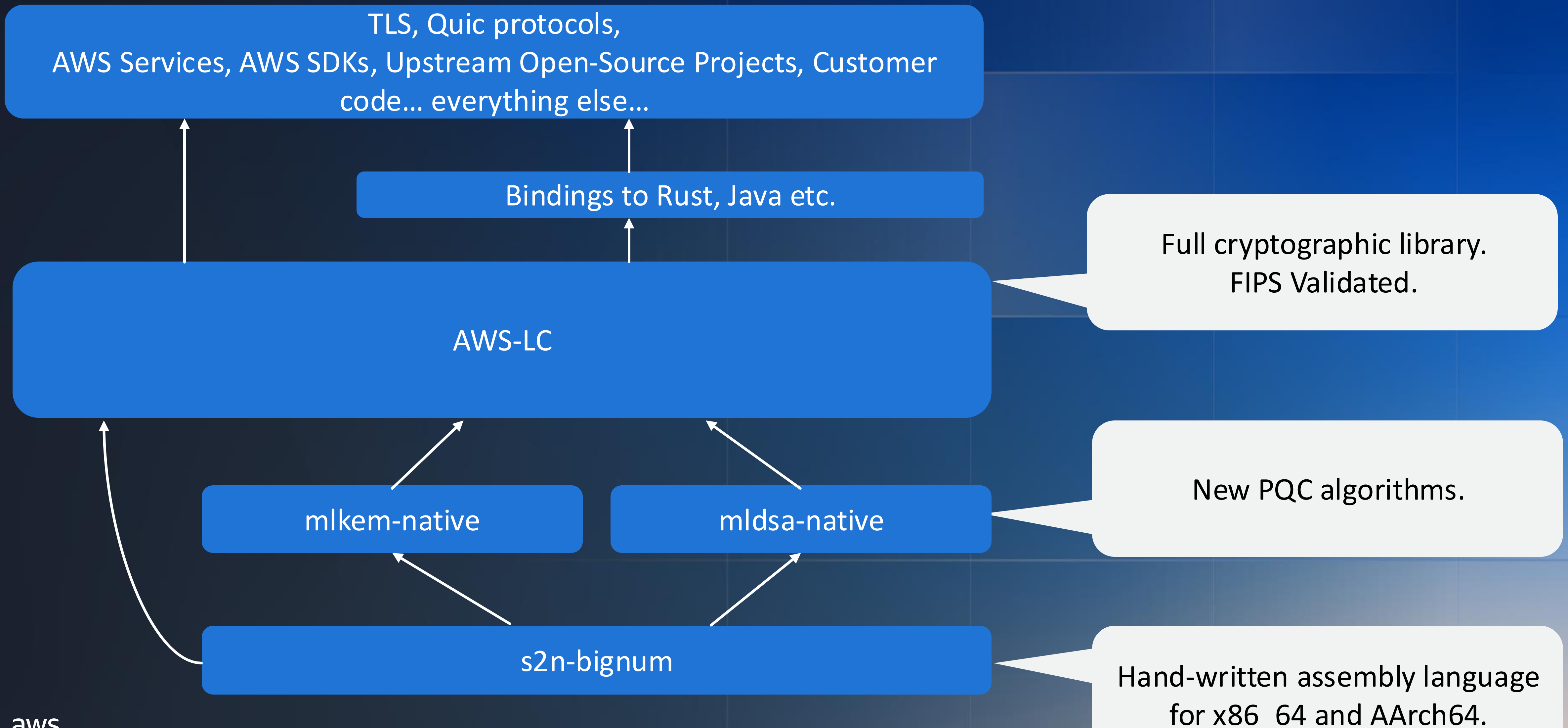
Extraordinarily high bar for verification

- Functional correctness
- Performance (at AWS Scale, this really matters…)
- Freedom from known side-channel attacks

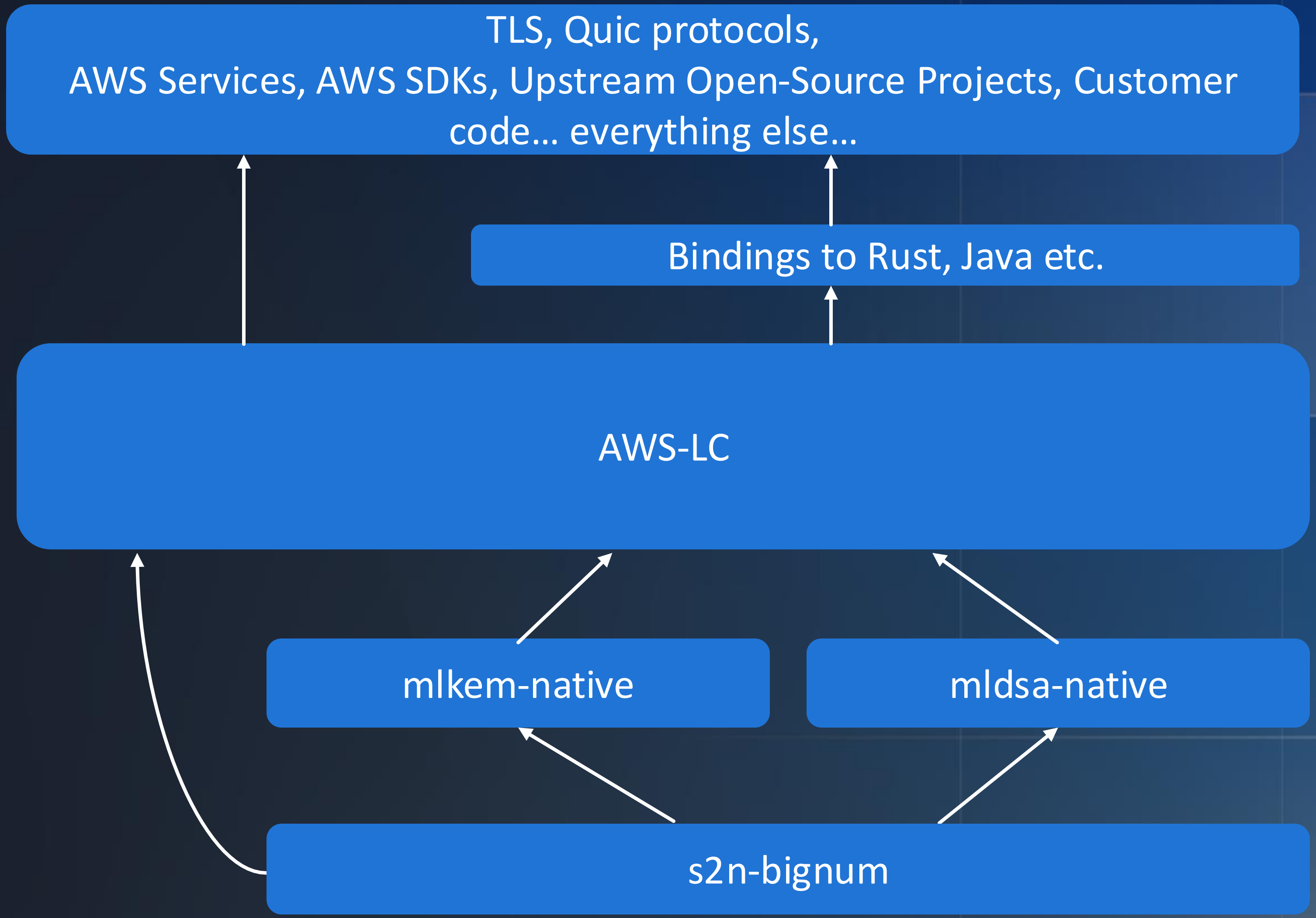
Plus…

- Longevity
- Agility

Crypto Library Layers...



Crypto Library Layers…



Each “Blob” is a stand-alone repository on GitHub, permissively licensed, and may be re-used by you.

Back to the plot…The big ideas…

Hybrid Verification

and

Fearless Optimization

Hybrid Verification? Eh?

Mix dynamic and static verification

Dynamic – aka “testing” – using KATs, fuzz testing, runtime assertion checking, Valgrind (including constant-time checking) etc. etc.

Static – aka “Formal Verification” or “Proof” of program correctness properties.

Hybrid Verification? Eh?

Big idea 2:

Modulate the *depth* of Static Verification to get the most bang-for-the-buck. Trade depth for scale pragmatically.

The SV “Depth Gauge”

Bronze: basic “linting” for dumb bugs plus *all* the dynamic verification.

Silver: Bronze + proof of memory-safety and type-safety.

Gold: Silver + Proof of functional correctness with respect to a formal specification (e.g maths, Cryptol)

Why does this matter?

Fully static verification of type-safety offers the most bang-for-the-buck.

Can be *fully automated* using languages that offer contracts and/or an expressive type system.

Good news: it takes a bit of work, but safety-critical industry have been doing this for years. The tech is well understood.

The big ideas…

Hybrid Verification

and

Fearless Optimization

Why does this work?

Well-typed programs cannot ”go wrong”…

Robin Milner

Why does this work?

Proven type-safe programs run really fast as well...

Contemporary high-level language compilers are incredibly reliable if you only compile type-safe programs.

"Fearless optimization…"
…Supported by Proof

Reference code. Correct but not fast enough.

Terminate when returns diminish…

Optimize

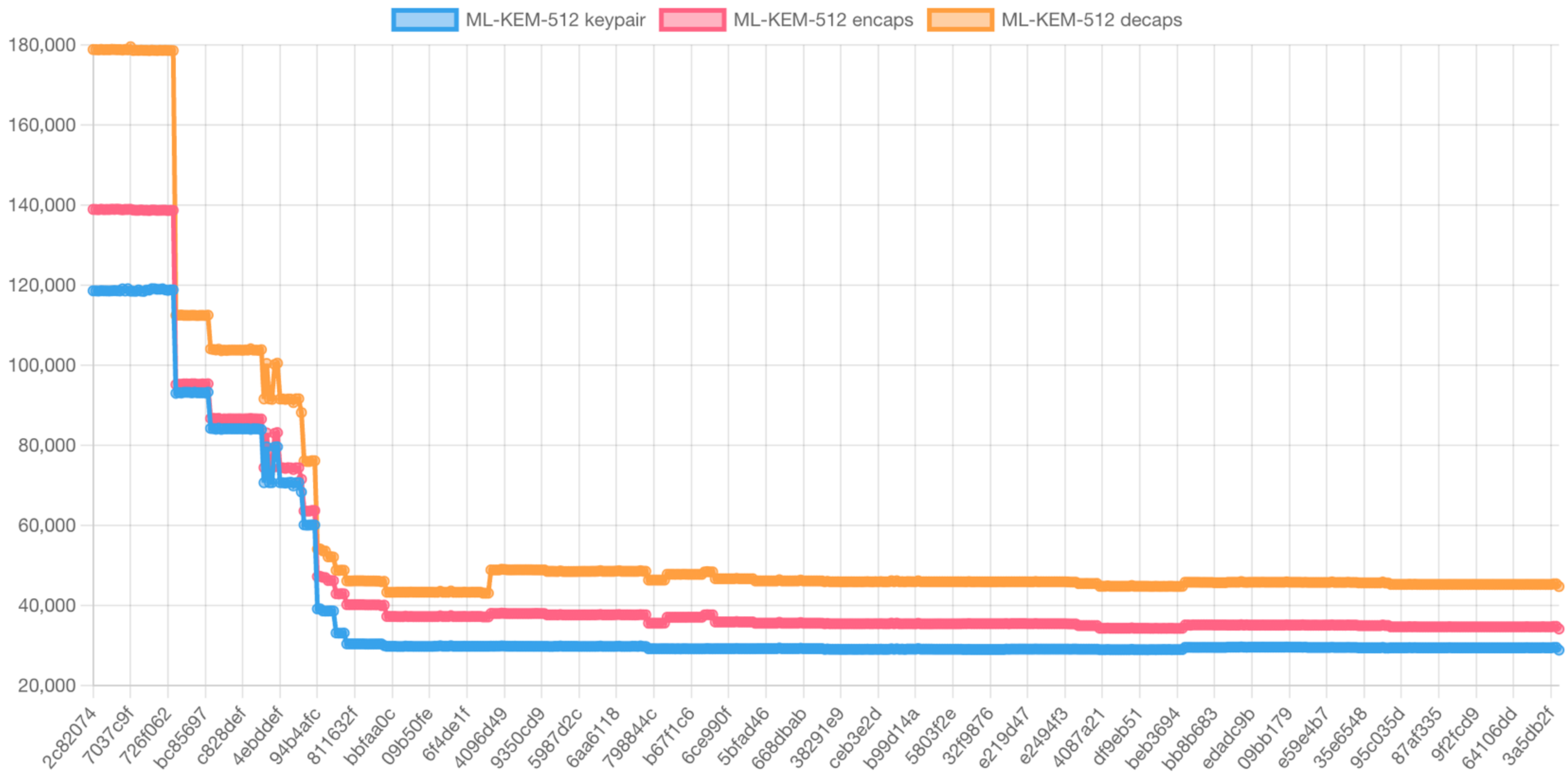
Re-prove

Measure

“Fearless optimization…”

…*Driven* by Proof

Graviton2



ML-KEM-512 keypair
ML-KEM-512 encaps
ML-KEM-512 decaps
180,000
160,000
140,000
120,000
100,000
80,000
60,000
40,000
20,000
2c82074
7037c9f
726f062
bc85697
c828def
4ebddef
94b4afc
811632f
bbfaa0c
09b50fe
6f4de1f
4096d49
9350cd9
5987d2c
6aa6118
798844c
b67f1c6
6ce990f
5bfad46
668dbab
38291e9
ceb3e2d
b99d14a
5803f2e
32f9876
e219d47
e2494f3
4087a21
df9eb51
beb3694
bb8b683
edadc9b
09bb179
e59e4b7
35e6548
95c035d
87af335
9f2fcd9
64106dd
3a5db2f

Introducing mlkem-native

An open-source implementation of FIPS-203 (MLKEM).

Already integrated with

- AWS-LC
- Linux Foundation's LibOQS

Introducing mlkem-native

Designed to deliver strong performance and assurance for server-class hardware (e.g. AWS Data Centres).

NOT designed for high-threat embedded applications (code exhibits cryptographic constant-time, but not designed to resist power-analysis…)

Introducing mlkem-native

“Front end” – portable C code.

“Back-end” – implements time- and security-critical functions.
There are 4 of these…

1. Intel x86_64 AVX2 assembly language
2. AArch64 NEON assembly language
3. RISC-V RV64imv assembly language
4. C (for every other machine)

Introducing mlkem-native

Static Verification status:

All C code proven type-safe using contracts and CBMC tool in “auto-active” style.

AArch64 – proven functionally correct using HOL-Light (15 functions).

Intel AVX2 – will be proved in HOL-Light. Work-in-progress.

RISC-V RV64imv – not yet (not an AWS-led effort anyway…)

Back to the data…

MLKEM Number-Theoretic Transform (NTT)
running on Graviton2 (EC2 c6g instance) performance in clock cycles. Lower is better.

Code language/version	Cycles
PQCrystals/Kyber reference C code	3481
Rod’s best formally verified and optimized C version	1239
Hand-written, formally verified AArch64 assembly language (“Clean” version)	900
AArch64 assembly, auto-super-optimized (also formally verified)	808

How did this get done?

Optimizing MLKEM in C…

There are many “standard tricks”

- Montgomery representation and multiplication of polynomial coefficients.
- Barrett reduction
- ”Layer merging” of the NTT and Inverse NTT
- Deferred modular reduction
- Caching intermediate results in polynomial multiplication

All verified and defended by continuous proof of type-safety in CI.

Back to the data…

MLKEM Number-Theoretic Transform (NTT)
running on Graviton2 (EC2 c6g instance) performance in clock cycles. Lower is better.

Code language/version	Cycles
PQCrystals/Kyber reference C code	3481
Rod’s best formally verified and optimized C version	1239
Hand-written, formally verified AArch64 assembly language (“Clean” version)	900
AArch64 assembly, auto-super-optimized (also formally verified)	808

What’s this?

Introducing…

The Super (Lazy) Optimization of Tricky Handwritten assemblY

aka "SLOTHY"

SLOTHY?

Inputs

- Plain “clean” human-written assembly language.
- Description of target machine micro-architecture.
  (e.g. number and capabilities of each pipeline, instruction latencies etc.)

Output

- Equivalent, but faster code.

Limitation

- Only works for AArch64 code at the moment.

How does SLOTHY work?

1. Register re-allocation. Use more registers to remove apparent data-flow dependencies.
2. Re-order instructions to use more execution pipelines, more of the time.
3. Unwind and re-structure inner loops.

(Under the hood – constraint solving)

SLOTHY results

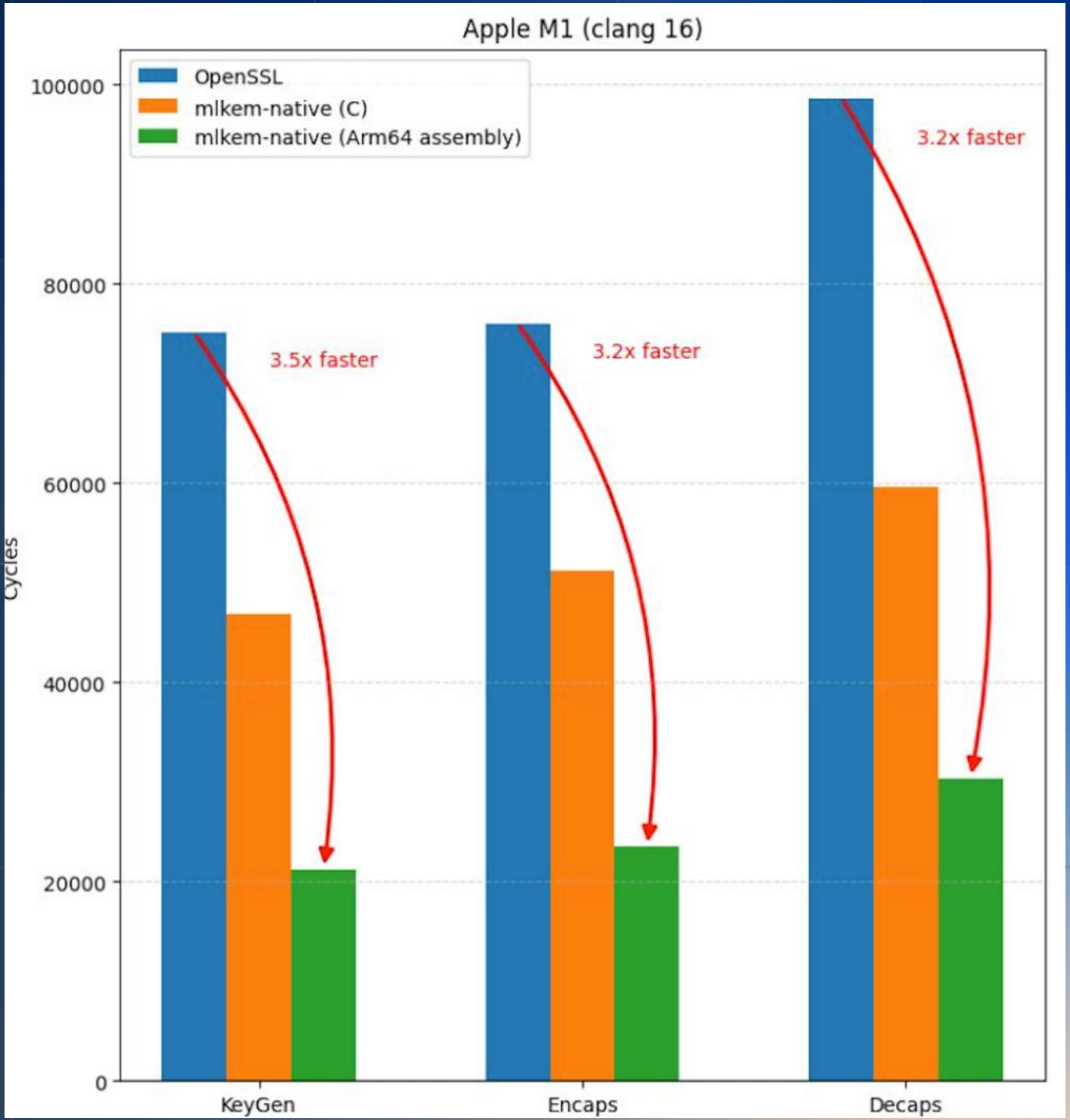
SLOTHY has been used on hand-written code in AWS-LC and s2n-bignum libraries, including RSA, Elliptic-Curves, the AArch64 back-ends for MLKEM and MLDSA.

Noticeable performance improvement on Graviton-2 over original hand-written code.

Seems to work well for most AArch64 implementations, including Cortex-Axx, neoverse, Apple M1 etc.

Putting it all together

mlkem-native
vs
OpenSSL 3.5



Resources…

s2n-bignum: https://github.com/awslabs/s2n-bignum
mlkem-native: https://github.com/pq-code-package/mlkem-native
mldsa-native: https://github.com/pq-code-package/mldsa-native
AWS-LC: https://github.com/aws/aws-lc
SLOTHY: https://github.com/slothy-optimizer/slothy

Questions…

rodchap@amazon.co.uk