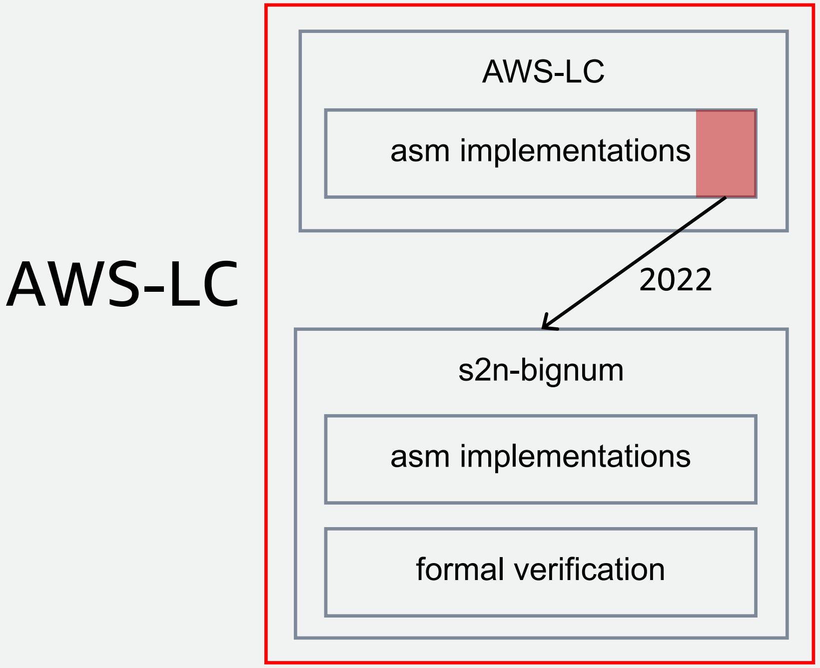
Adoption of High-Assurance and Highly Performant Cryptographic Algorithms at AWS

RWC 2024

Dusan Kostic, Hanno Becker, John Harrison,
Juneyoung Lee, Nevine Ebeid, and **Torben Hansen**
Amazon Web Services

AWS Libcrypto (AWS-LC) and s2n-bignum libraries

- AWS-LC (2020-):
  - Fork of Google's BoringSSL.
  - Optimized for AWS use-cases.
  - FIPS 140-3 validated.
- s2n-bignum (2018-):
  - CPU-specific cryptographic algorithm implementations.
  - Formal verification of correctness.



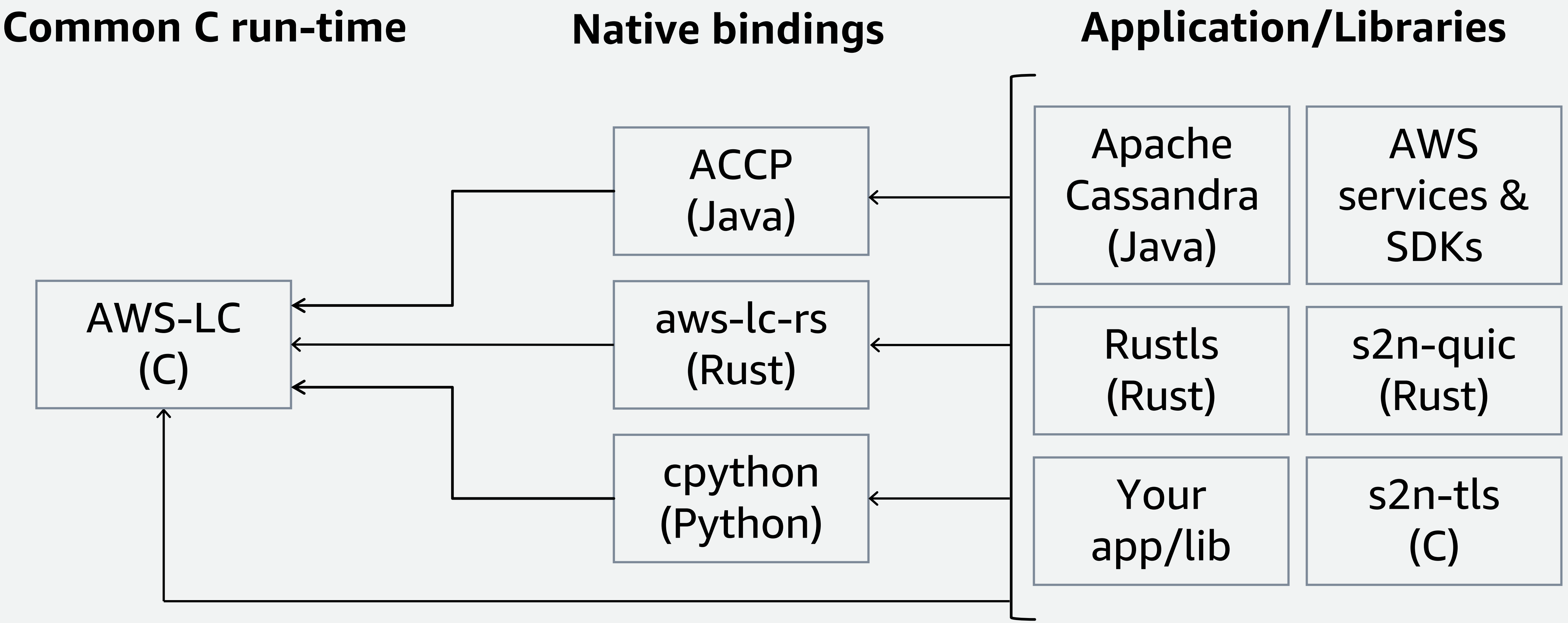
This talk will explore the following 3 topics:

Distribution: How we distribute AWS-LC.

Performance: How we ensure AWS-LC is fast.

Assurance: How we ensure AWS-LC is safe using formal reasoning.

AWS cryptographic algorithm stack



Why not implement cryptographic algorithms natively?

Ensure cryptographic algorithms are available where software is built and run.

Algorithm needs	Availability needs
Data-in-transit	Rust, Java, C, Python
Data-at-rest	x86_64, Arm64
Quantum-safe	FIPS validated

Benefits of common C run-time:

- Implement optimizations **once**.
- Test implementations **once**.
- FIPS: validate **once**.

Common C run-time and native language bindings for scalable distribution of cryptographic algorithm implementations.

This talk will explore the following 3 topics:

Distribution: How we distribute AWS-LC.

Performance: How we ensure AWS-LC is fast.

Assurance: How we ensure AWS-LC is safe using formal reasoning.

Case study: x25519 and Ed25519 (x/Ed25519)

Case study: x25519 and Ed25519 (x/Ed25519)

AWS-LC x/Ed25519 previously used Fiat Crypto.

New implementations in AWS-LC:

1. Written in assembly (x86_64 and Arm64).
2. Consider micro-architectural (μarch) differences in optimizations.
3. High algorithm “scope”.
4. Formal verification.

Harnessing μarch diversity

Dive a bit deeper

AWS EC2 offers a wide varity of instance types:

- Arm64: AWS Graviton 2 and 3.
- x86_64: Many Intel and AMD CPU models.

Why it matters?

Every μarch has unique characteristics:
pipelining, instruction latencies/throughput, …

Therefore unique optimization opportunities.

Focus

Field operations
$a \cdot b \bmod 2^{255}-19$

x/Ed25519 μarch-specific scalar mul on AWS Graviton

Graviton 3 specific

Schoolbook multiplication

Common

Bernstein-Yang divstep modular inverses [3]

“linear” functions e.g. x+y mod 2^{225}-19

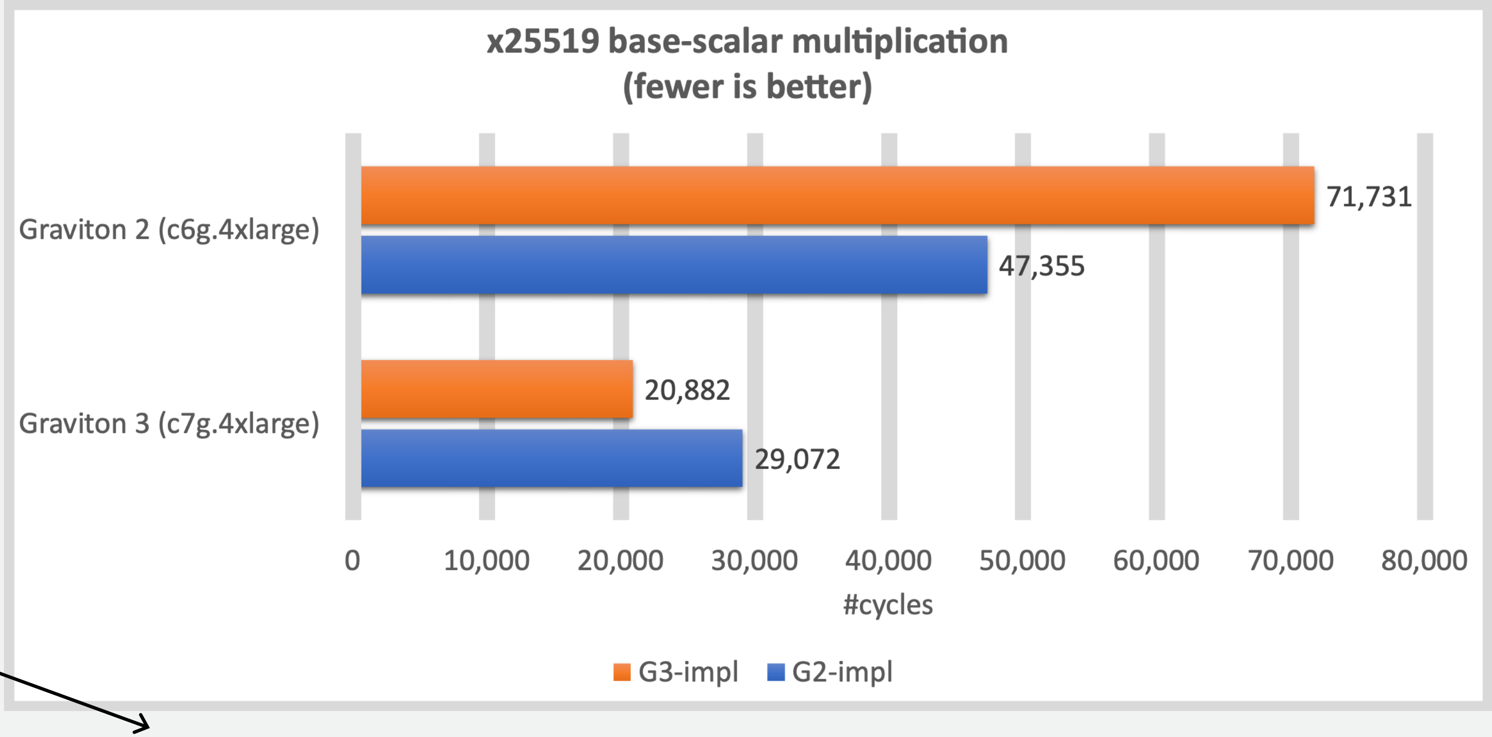
Graviton 2 specific

Karatsuba multiplication

Use scalar+vector units:

- Lenngren “hybridization” [1]
- SLOTHY “super-optimization” [2]

Optimal code is μarch-specific



Graviton 3 optimized

Graviton 2 optimized

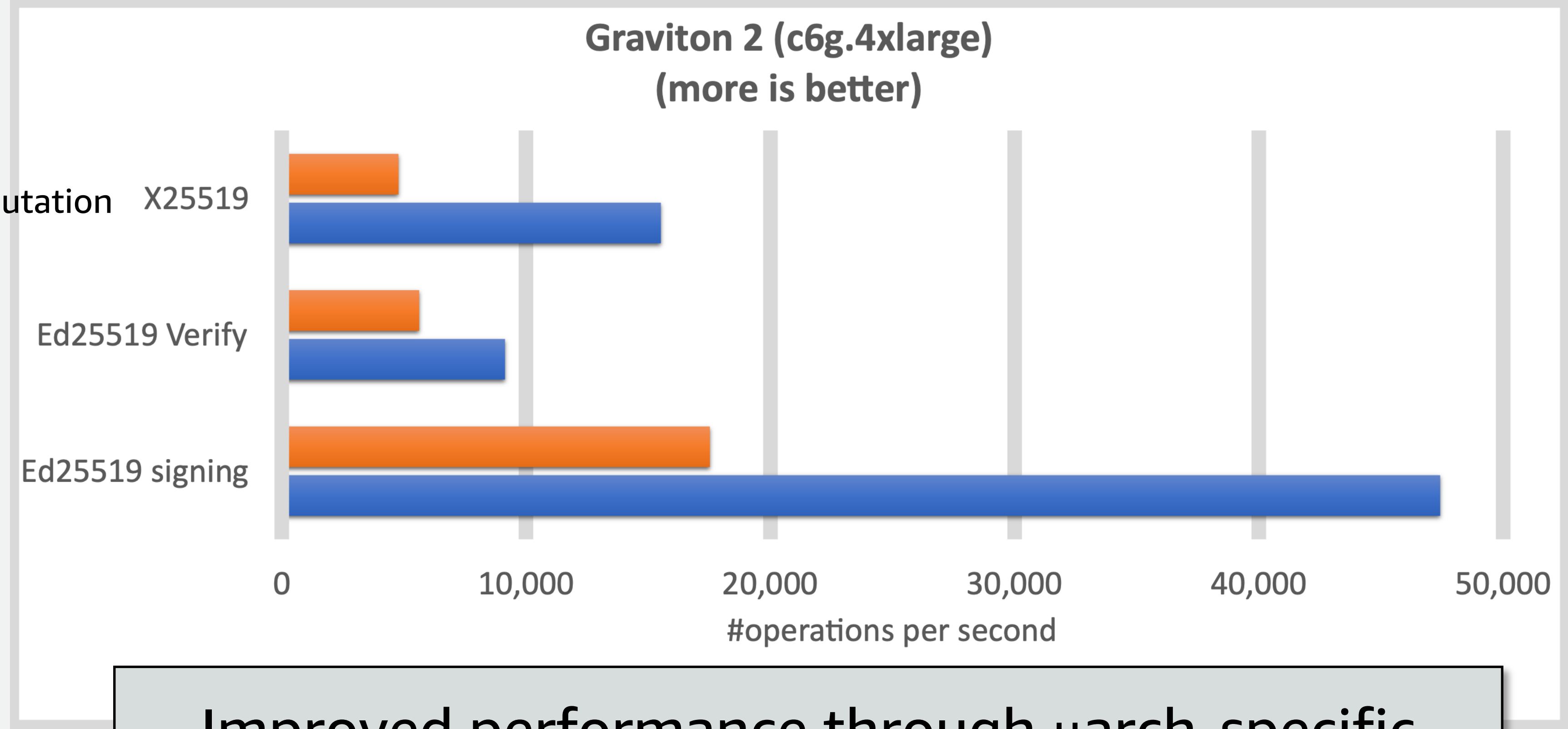
If **G3-impl** was used on **Graviton 2**: 51% slower

If **G2-impl** was used on **Graviton 3**: 39% slower

Difference matters to production work loads

New x/Ed25519 implementations in AWS-LC: Graviton 2

key generation +
shared key computation



Improved performance through µarch-specific implementations for both Arm64 and x86_64

This talk will explore the following 3 topics:

Distribution: How we distribute AWS-LC.

Performance: How we ensure AWS-LC is fast.

Assurance: How we ensure AWS-LC is safe using formal reasoning.

Case study: x25519 and Ed25519 (x/Ed25519)

Existing assurance methods used in AWS-LC

- Code-review ✓✓.
- Unit testing.
- Fuzzing, Cryptofuzz, …
- Memory sanitizers, Valgrind, …
- Wycheproof.
- FIPS and ACVP.

Increase assurance through formal verification: Prove functional correctness.

Proving functional correctness of x/Ed25519

Correctness bug in rare code-path in implementation. **Impact AWS customer experience:**

very low x25519 TLS error-rate
+
50MM TLS requests/second

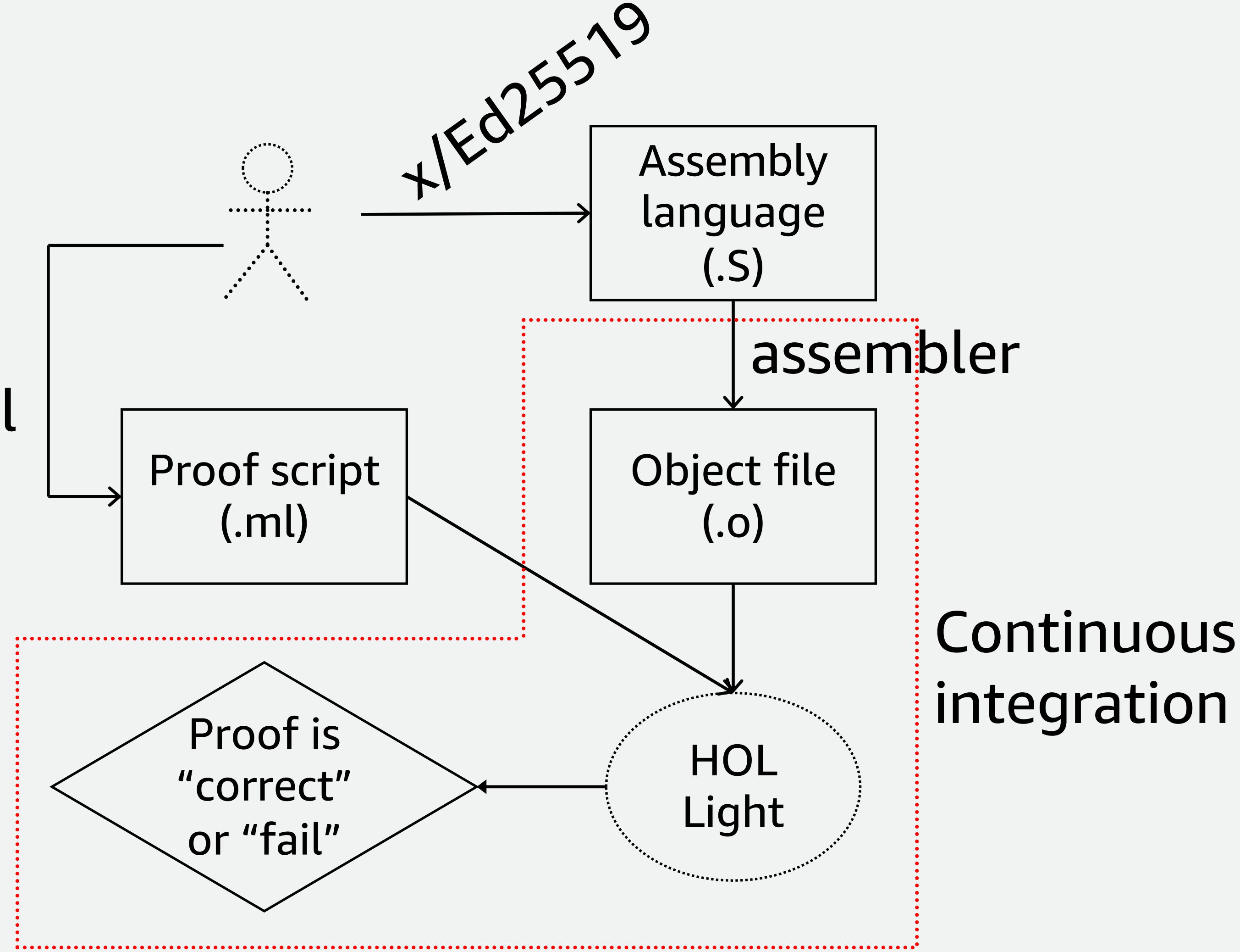
visible Cx impact.

Proof engine requirements:

- Handle x86_64, Arm64, and μarch's.
- Verify object code – ~~compiler assumption.~~

Proof engine: HOL Light and how we use it

- Interactive proof assistant.
- Written and maintained by John Harrison.



- Abstract mathematical specification.
- ISA model.

What do we prove for x/Ed25519?

“Executing x/Ed25519,
with their specific sequence of object code bytes,
on an Arm64 or x86_64 CPU,
will correctly compute the same result as the
abstract mathematical specification”

Main assumptions:

ISA model captures real-world

Specification is correct

HOL Light engine is correct

See all proofs in s2n-bignum project: https://github.com/awslabs/s2n-bignum

Formal verification of functional correctness through HOL Light with no compiler assumptions.

Our experience implementing optimized cryptographic algorithms while using formal reasoning

- Significant time investment was required; 1 person years for end-to-end x/Ed25519 implementation and integration. Worth it at scale.

- Did we hit any major roadblocks? No…

- Mostly non-cryptographic and non-formal verification issues:
  - Portability and build issues: e.g. **.text** ELF section non-readable but stores data tables.
  - Code size from adding multiple implementations for the same CPU.

Summary

Successfully development cryptographic algorithm implementations of x/Ed25519 combining high-performance and formal verification now servicing Trillions of requests a day.

Common C run-time and native language bindings for scalable distribution of cryptographic algorithm implementations.

Improved performance through µarch-specific implementations for both Arm64 and x86_64.

Formal verification of functional correctness through HOL Light with no compiler assumptions.

Resources

- [1] https://github.com/Emill/X25519-AArch64
- [2] https://github.com/slothy-optimizer/slothy
- [3] Daniel J. Bernstein and Bo-Yin Yang, *Fast constant-time gcd computation and modular inversion*, IACR Trans. Cryptogr. Hardw. Embed. Syst., 2019.
- AWS-LC: https://github.com/aws/aws-lc
- s2n-bignum: https://github.com/awslabs/s2n-bignum
- aws-lc-rs: https://github.com/aws/aws-lc-rs
- accp: https://github.com/corretto/amazon-corretto-crypto-provider
- s2n-tls: https://github.com/aws/s2n-tls
- s2n-quic: https://github.com/aws/s2n-quic
- HOL Light: https://github.com/jrh13/hol-light
- Open-source cryptography @ AWS: https://aws.amazon.com/security/opensource/cryptography/
- Automated reasoning @ AWS: https://aws.amazon.com/what-is/automated-reasoning/
- Cryptographic computing @ AWS: https://aws.amazon.com/security/cryptographic-computing/

Open-source cryptography @ AWS

https://aws.amazon.com/security/opensource/cryptography

Thank you!

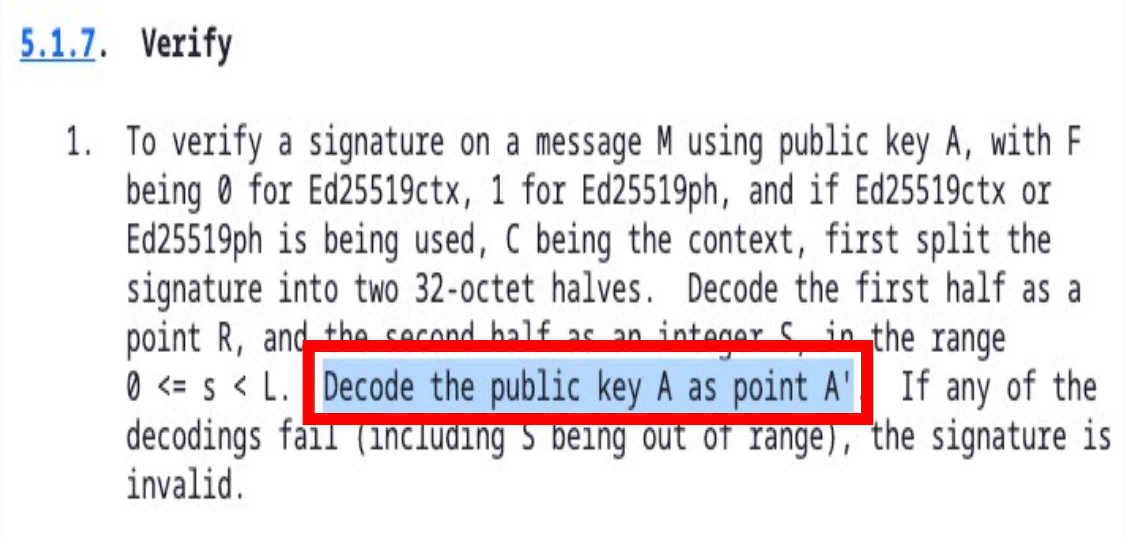
Torben Hansen

htorben@amazon.com

Extra slides in the unlikely event of extra time

Optimizations of x/Ed25519: Algorithm scope

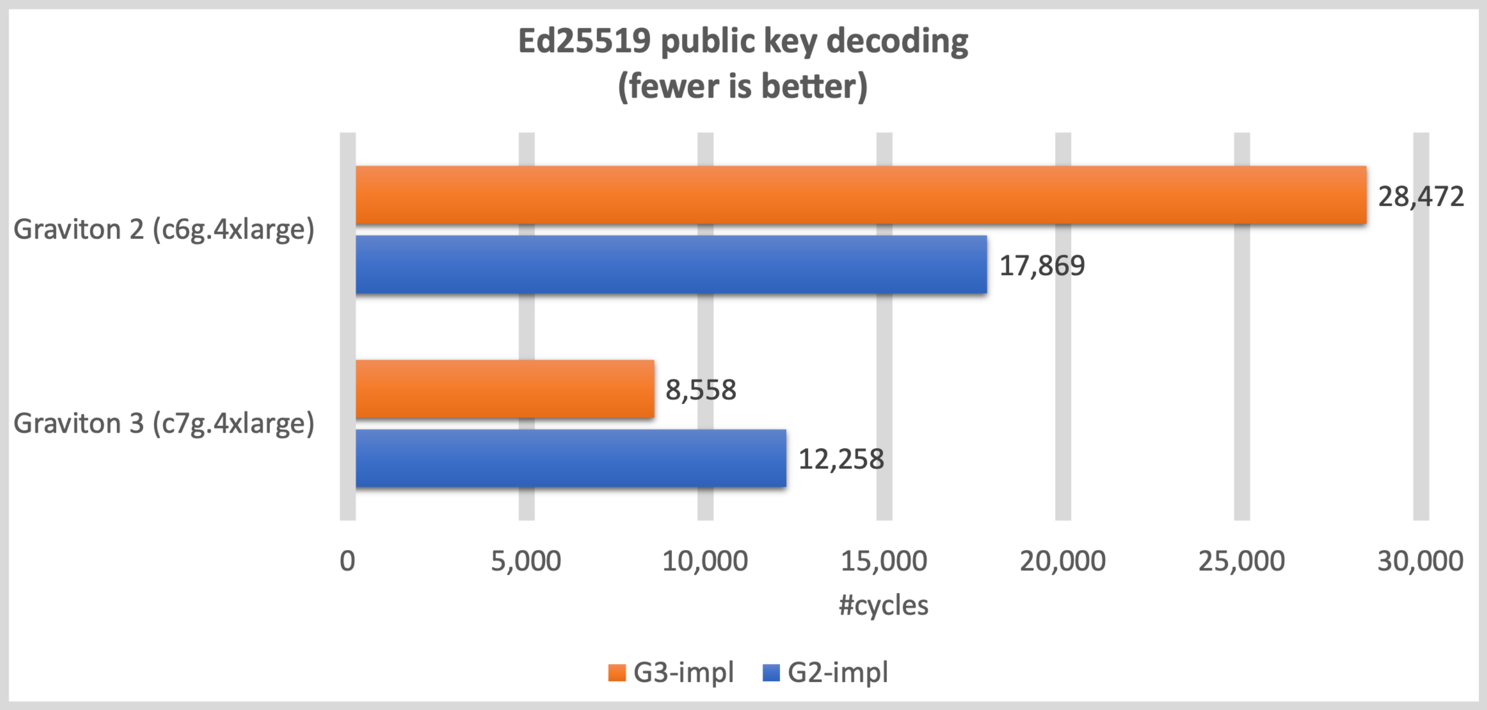
Ed25519 RFC8032 verify step:
Decode public key



5.1.7. Verify

1. To verify a signature on a message M using public key A, with F being 0 for Ed25519ctx, 1 for Ed25519ph, and if Ed25519ctx or Ed25519ph is being used, C being the context, first split the signature into two 32-octet halves. Decode the first half as a point R, and the second half as an integer S, in the range 0 <= s < L. Decode the public key A as point A'. If any of the decodings fail (including S being out of range), the signature is invalid.

Optimal code must consider complete algorithm scope



Ed25519 verify = decoding + other operations.
Using best-μarch implementation: decoding 9% → 6% of total Ed25519 verify.