Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}

Jipeng Zhang[1], Yuxing Yan[2], Junhao Huang[3,4], Çetin Kaya Koç[1,5,6]

[1]Nanjing University of Aeronautics and Astronautics
jp-zhang@outlook.com
[2]Shanghai Aerospace Electronic Technology Institute
yanyuxing7408@163.com
[3]BNU-HKBU United International College [4]Hong Kong Baptist University
huangjunhao@uic.edu.cn
[5]Iğdır University [6]University of California Santa Barbara
cetinkoc@ucsb.edu

Artifact: `https://github.com/Ji-Peng/PQRV/tree/ches2025`

IACR TCHES 2025, to appear

2024-10-23

Outline

Motivations

- Limited Software Implementations on RISC-V.
  - Optimized implementations of PQC on ARM platforms have received significant attention.
  - However, there are comparatively fewer optimized implementations on RISC-V.
- Hardware Implementations Lack Efficient Baseline.
  - Hardware implementations rely on the C reference implementation as their evaluation baseline.
  - Using fully optimized software implementations as a baseline would be more reasonable.
- Challenges of Optimizing PQC for Different RISC-V ISAs.
  - Different RISC-V ISA combinations lead to significant variations in optimization strategies.
  - For example, the Keccak implementation on RV{32,64}I{B}{V} presents eight distinct ISA combinations

Background: FIPS 202, 203, and 204

- In 2015, FIPS 202 established the SHA-3 standard, which includes four hash functions (SHA3-224,256,384,512) and two extendable-output functions (SHAKE128,256).
  - The core component of SHA-3 is the Keccak-f1600 algorithm.
- In 2024, FIPS 203 and 204 introduced lattice-based post-quantum cryptographic standards, ML-KEM and ML-DSA, based on Kyber and Dilithium, respectively.
  - The time-consuming operations in Kyber and Dilithium mainly involve SHA-3 and NTT polynomial multiplication.

Background: Keccak-f1600

- The Keccak-f1600 algorithm operates on a 1600-bit state, consisting of 24 rounds, with each round comprising 5 steps (θ, ρ, π, χ, and ι).

Algorithm 1 A round of Keccak-f1600

Input: The 1600-bit state A; A constant RC
Output: A

1: θ step:
2: $C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4], \quad \forall x \text{ in } 0 \dots 4$
3: $D[x] = C[x-1] \oplus \texttt{ROT}(C[x+1],1), \quad \forall x \text{ in } 0 \dots 4$
4: $A[x,y] = A[x,y] \oplus D[x], \quad \forall (x,y) \text{ in } (0 \dots 4, 0 \dots 4)$
5: ρ and π steps:
6: $B[y, 2x+3y] = \texttt{ROT}(A[x,y], r[x,y]), \quad \forall (x,y) \text{ in } (0 \dots 4, 0 \dots 4)$
7: χ step:
8: $A[x,y] = B[x,y] \oplus ((\texttt{NOT } B[x+1,y]) \texttt{ AND } B[x+2,y]), \quad \forall (x,y) \text{ in } (0 \dots 4, 0 \dots 4)$
9: ι step:
10: $A[0,0] = A[0,0] \oplus \text{RC}$
11: **return** A

Common optimization techniques for Keccak

- Bit interleaving: Changing the state encoding to convert 64-bit rotations into 32-bit rotations.
- Lane complementing: Inverting parts of the state to reduce the number of inversion operations.
- In-place implementation: Keeping the state storage location unchanged after N rounds of computation.
- Lazy rotations: Using the Barrel Shifter feature of ARMv8-A or ARMv7-M to eliminate rotation instructions.

Common optimization techniques for NTT

- Algorithmic or implementation level optimization of modular multiplication algorithms.
  - Montgomery, Barrett, and Plantard.
- Layer merging strategy: reducing memory access.
- Parallelized implementation.
  - SIMD or Vector.

Target platform: CanMV-K230 with C908 core

- RISC-V {32,64}GCBV.
- Supports both 32-bit and 64-bit execution states.
- Supports Vector Extension 1.0 with VLEN=128.
- Supports B Extension 1.0.
- Dual-issue scalar instructions, single-issue vector instructions.

Instruction	Latency	CPI
RV{32,64}I		
arith/logic/compare	1	0.5
lh/lw/ld	3	1
sh/sw/sd	1	1
RV{32,64}M		
mulw on RV64M	3	1
mul{h} on RV64M	4	2
mul{h} on RV32M	3	1
RV{32,64}B		
rori/andn	1	0.5
RV{32,64}V		
vadd/vsub	4	1
vmul{h} with SEW$\leq$16	4	1
vmul{h} with SEW$>$16	5	1
logic/vmerge	4	2
vrgather	5	4
vle	≥ 3	2
vse	1	2.3

Optimized Keccak on RV64I and RV64IB

The 64-bit rotation operation

- RV64I: `slli t,a,n; srli b,a,64-n; xor b,b,t`
- RV64IB: `rori`

Lane complementing

- RV64I: This reduces the number of `NOT` operations required in the χ step of a Keccak-f1600 round from 25 to 8.
- RV64IB: `andn`

In-place implementation: Becker et al.'s in-place method.

- The goal is to slightly offset `loc` $B[]$ from `loc` $A[]$ for the computation of $\rho\pi$ and to move entries back to their original places in χ.

Dual issue optimizations: reducing RAW hazards.

Optimized Keccak on RV32I and RV32IB

Lane complementing: RV32I ✓; RV32IB ✗.
In-place implementation: Becker et al.'s in-place method.
Bit interleaving: RV32I ✓; RV32IB ✗.
Register allocation and optimizations for dual issue

- RV32: $30 \times 32 = 960$ bits < 1600 bits
- Our goal is pipeline friendliness. Using 20 registers to keep the 640-bit state resident in the registers

```
.macro xor5 dst,b,g,k,m,s,tmp
    lw     \dst, \b(a0)
    lw     \tmp, \g(a0)
    xor    \dst, \dst, \tmp
    lw     \tmp, \k(a0)
    xor    \dst, \dst, \tmp
    lw     \tmp, \m(a0)
    xor    \dst, \dst, \tmp
    lw     \tmp, \s(a0)
    xor    \dst, \dst, \tmp
.endm
```

```
lw    tmp0l, 0*8+0(a0) # A[0,0]l
lw    tmp0h, 0*8+4(a0) # A[0,0]h
xor   tmp1l, A[0,1]l, A[0,2]l
xor   tmp1h, A[0,1]h, A[0,2]h
lw    tmp2l, 15*8+0(a0) # A[0,3]l
lw    tmp2h, 15*8+4(a0) # A[0,3]h
xor   tmp1l, \tmp1l, \tmp0l
xor   tmp1h, \tmp1h, \tmp0h
lw    tmp0l, 20*8+0(a0) # A[0,4]l
lw    tmp0h, 20*8+4(a0) # A[0,4]h
xor   tmp1l, \tmp1l, \tmp2l
xor   tmp1h, \tmp1h, \tmp2h
xor   tmp1l, \tmp1l, \tmp0l
xor   tmp1h, \tmp1h, \tmp0h
```

Optimized Keccak on RISC-V Vector

- The implementation on RVV is nearly identical to that on RV64I.
- VLEN=128 $\rightarrow$ 2-way implementation.

Hybrid Keccak implementations

- The C908 has relatively weak capabilities for computing logical vector instructions, with a corresponding CPI of 2.
- `vand; vand` consumes 4 cycles.
- `vand; vand; andx4` also consumes 4 cycles.
- The hybrid implementation on C908 RV64IBV shows no performance improvement over the RV64IB scalar implementation, as the scalar implementation is already sufficiently fast.
- Hybrid implementation on C908 RV32IBV: the hybrid 3-way implementation consisting of an RVV implementation and a scalar implementation performs best.

Keccak: Results and Comparisons

Implementation	Method	Cycles	Instructions	CPI
C on RV32I	ref.	15779	24487	0.64
[Sto19] on RV32I	lane compl. & bit interl. & 4-round in-place & plane-by-plane	8734	12740	0.69
Our RV32I	lane compl. & 1-round in-place & dual-issue opt.	7808	14890	0.52
C on RV32IB	ref.	12341	20460	0.6
Our RV32IB	bit interl. & 1-round in-place & dual-issue opt.	6222	11554	0.54
C on RV64I	ref.	4926	8585	0.57
Our RV64I	lane compl. & 1-round in-place & dual-issue opt.	2591	5049	0.51
C on RV64IB	ref.	2412	4296	0.56
RISCV-Crypto[1] on RV64IB	inline asm for `rori/andn`	2563	4649	0.55
Our RV64IB	1-round in-place & dual-issue opt.	1770	3405	0.52

- RV32I, RV32IB, RV64I, and RV64IB: 12%, 98%, 90%, and 36% faster than previous work, respectively.

Keccak: Results and Comparisons

Our RVVx2	lane compl. & 1-round in-place & dual-issue opt.	9655 (4827)	5462	1.77
Our RV32IVx3	RV32I & RVVx2	11850 (3950)	20273	0.58
Our RV32IVx4		20374 (5093)	35190	0.58
Our RV32IVx5		30544 (6108)	50285	0.61
Our RV32IBVx3	RV32IB & RVVx2	10527 (3509)	17012	0.62
Our RV32IBVx4		16670 (4167)	28472	0.59
Our RV32IBVx5		24299 (4859)	39991	0.61

- RV32IVx3: 22% and 98% faster than RVV and RV32I, respectively.
- RV32IBVx3: 38% and 77% faster than RVV and RV32IB, respectively.

Kyber and Dilithium NTT

- Kyber: 16-bit NTT; Dilithium: 32-bit NTT.
- Plantard multiplication requires $l \times 2l$-bit multiplier.
  - Kyber NTT on RV{32,64}IM; Dilithium NTT on RV64IM.

Algorithm 5 Plantard multiplication by a constant for Kyber on RV32IM [HZZ+24]

Input: 32-bit signed integer $a \in [-137q, 230q]$; $\alpha = 3$; precomputed $2l$-bit integer bq' where b is a constant and $q' = q^{-1} \bmod 2^{2l}$; $q2^l = q \times 2^l$; $l = 16$

Output: $r = ab(-2^{-2l}) \bmod^{\pm} q, r \in (-\frac{q}{2}, \frac{q}{2})$

1: $bq' \leftarrow bq^{-1} \bmod^{\pm} 2^{2l}$ ▷ precomputed
2: **mul** r, a, bq' ▷ $r \leftarrow [abq']_{2l}$
3: **srai** $r, r, \#16$
4: **addi** $r, r, 2^{\alpha}$ ▷ $r \leftarrow ([r]^l + 2^{\alpha})$
5: **mulh** $r, r, q2^l$ ▷ $r \leftarrow [rq2^l]^{2l}$
6: **return** r

Algorithm 6 Montgomery multiplication for Dilithium on RV32IM

Input: Two signed integers a, b satisfying $-\frac{\beta}{2}q \leq ab < \frac{\beta}{2}q$, where $q \in (0, \frac{\beta}{2})$ and $\beta = 2^{32}$; precomputed integer bq' where b is a constant and $b \in (0, q)$; $q' = q^{-1} \bmod \beta$

Output: $r = ab\beta^{-1} \bmod^{\pm} q, r \in (-q, q)$

1: $bq' \leftarrow bq^{-1} \bmod^{\pm} \beta$ ▷ precomputed
2: **mul** r, a, bq'
3: **mulh** t, a, b
4: **mulh** r, r, q
5: **sub** r, t, r
6: **return** r

Kyber and Dilithium NTT

Pipeline optimization:

- A suboptimal case: `mul` r, a, bq'; `srai` $r, r, \#16$; `addi` $r, r, 2^{\alpha}$; `mulh` $r, r, q2^{l}$
- The `mul` consumes 3 cycles on RV32IM.
- Six-way alternation yields optimal performance.
- However, in NTT implementations, we can achieve at most four-way alternation: `mul`x4; `srai`x4; `addi`x4; `mulh`x4 ...
- Kyber NTT: 13218 cycles $\rightarrow$ 5714 cycles

NTT: Results and Comparisons

- Kyber NTT on RV32IM: $1.7\times \sim 2.3\times$
- Kyber NTT on RVV: $2.2\times \sim 3.6\times$ compared to RV32IM

	Impl.	Method	Cycles	Inst.	CPI
Kyber	[HZZ+24] on RV32IM	Plant & 4+3 & single-issue opt.& CT+GS	13218 11427 2891/5900	6774 7317 2572/3663	1.95 1.56 1.12/1.61
	Our on RV32IM	Plant & 4+3 & dual-issue opt.& CT+GS	5714 6005 1673/2668	6870 7398 2412/3536	0.83 0.81 0.69/0.75
	[HZZ+22] on Cortex-M4	Plant & 4+3& CT+GS	4474 4684 2422	- - -	- - -
	Our on RV64IM	Plant & 4+3 & dual-issue opt.& CT+GS	6609 6996 2122/3245	6871 7399 2413/3537	0.96 0.95 0.88/0.92
	Our on RVV	Mont & 1+6 & dual-issue opt& CT+GS	1575 1840 753	1347 1592 738	1.17 1.16 1.02
	[BHK+22] on A72-NEON	Barrett & 4+3& CT+GS	1200 1338 952	- - -	- - -
Dilithium	Our on RV32IM	Mont & 3+3+2 & dual-issue opt.& CT+GS	7054 7561 2026	8692 9206 2349	0.81 0.82 0.86
	Our on RV64IM	Plant & 4+4& CT+GS	8258 8484 2320	7765 8293 2227	1.06 1.02 1.04
	[AHKS22] on Cortex-M4	Mont & 3+3+2& CT+CT	8066 8388 1931	- - -	- - -
	Our on RVV	Mont & 4+4 & dual-issue opt.& CT+GS	3395 3540 668	2899 3106 118	1.17 1.14 5.66
	[BHK+22] on A72-NEON	Barrett & 4+4& CT+GS	2241 2821 1378	- - -	- - -

Kyber and Dilithium: Results and Comparisons

Table 4: Performance of Kyber768 and Dilithium3 on C908 RV{32,64}IM{B}{V}. Most of the cycle counts ($k = 1000$) are determined as the median over 10000 iterations, except that Dilithium `Sign` is obtained as the average over 10000 iterations.

Impl.	Kyber768			Dilithium3		
	KeyGen	Encaps	Decaps	KeyGen	Sign	Verify
[HZZ+24] RV32IM	1052k	1261k	1179k	-	-	-
[HAZ+24] Cortex-M4	604k	732k	674k	2394k	5575k	2302k
Ref RV32IM[1]	1222k	1602k	1691k	4123k	13671k	4145k
Ref RV32IMB[1]	1048k	1377k	1481k	3422k	12635k	3504k
Our RV32IM	496k	606k	578k	1934k	5069k	1889k
Our RV32IMB	447k	550k	532k	1752k	4746k	1720k
Our RV32IMV	312k	419k	371k	1165k	3193k	1165k
Our RV32IMBV	284k	382k	346k	1087k	3054k	1091k
Ref RV64IM[2]	742k	986k	1185k	1841k	8232k	1958k
Ref RV64IMB[2]	603k	803k	1011k	1328k	7217k	1474k
Our RV64IM	278k	326k	357k	920k	3333k	939k
Our RV64IMB	237k	275k	316k	753k	3085k	791k
Our RV64IMV	204k	243k	248k	810k	2406k	800k
Our RV64IMBV	165k	197k	207k	645k	2139k	646k
[BHK+22] A72	99k	127k	120k	515k	1089k	447k

- Kyber: RV32IM 2.0× ∼ 2.1×; RV32IMB 2.4× ∼ 2.8×
- Dilithium: RV32IM 2.1× ∼ 2.7×; RV32IMB 2.0× ∼ 2.7×

ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519

Jipeng Zhang[1], Junhao Huang[2,3], Lirui Zhao[1], Donglong Chen[2], Çetin Kaya Koç[1,4,5]

[1]Nanjing University of Aeronautics and Astronautics, Jiangsu, China
jp-zhang@outlook.com, lirui.zhao@outlook.com
[2]Guangdong Provincial Key Laboratory IRADS, BNU-HKBU United International College
huangjunhao@uic.edu.cn, donglongchen@uic.edu.cn
[3]Hong Kong Baptist University
[4]Iğdır University [5]University of California Santa Barbara
cetinkoc@ucsb.edu

Artifact: `https://github.com/Ji-Peng/eng25519_artifact`

Usenix Security 2024. Distinguished Award

2024-10-23

Outline

Motivations

- How can AVX-512IFMA instructions accelerate ECC?
  - Optimizing ECC using ARM NEON and AVX2 instructions has been thoroughly researched.
  - However, using AVX-512IFMA instructions remains underexplored.
- How can the optimized ECC implementation be integrated into TLS applications?
  - Few works consider integration; most focus solely on optimizing cryptographic implementations.
- How can the cold start issue of vector units be mitigated?
  - The cold start issue can cause some primitives to be up to 3.8 times slower than normal.
- To what extent can our optimized cryptographic implementation improve TLS applications?
  - It is more interesting to understand the improvements to TLS applications rather than just focusing on cryptographic primitive microbenchmarks.

Background: AVX-512 & X25519 and Ed25519

AVX-512

- 32 512-bit registers; Each 512-bit register can be divided into 32 16-bit, 16 32-bit, or 8 64-bit segments.
- AVX-512IFMA supports 52-bit multipliers, whereas AVX2 and AVX-512F only support 32-bit multipliers.

X25519 and Ed25519

- X25519, designed by Daniel J. Bernstein, is a Diffie-Hellman key exchange protocol based on Curve25519.
- Ed25519, designed by Daniel J. Bernstein et al., is an Edwards-curve digital signature algorithm.
- In 2018, RFC 8446 included X25519 and Ed25519 in the supported cipher suites for TLS 1.3.

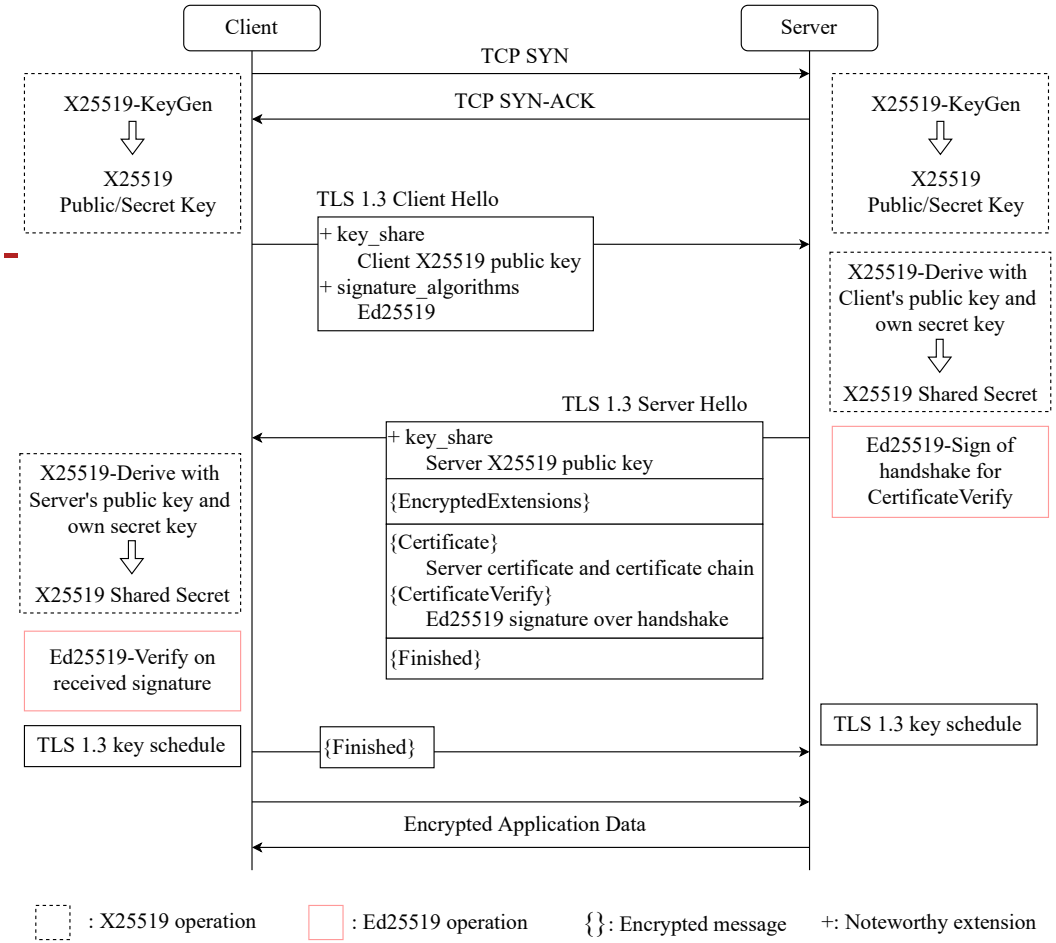
Background: TLS 1.3 handshake & DNS over TLS

TLS 1.3 handshake

- Client op; Server op.
- X25519-KeyGen/KeyGen+X25519-Derive+Ed25519-Sign+X25519-Derive+Ed25519-Verify.

DNS over TLS

- TLS handshake→DNS queries and responses over the TLS connection.



Optimized X25519 and Ed25519 implementation

Field arithmetic

- Radix-2^{51}: A field element $f = f_0 + 2^{51}f_1 + 2^{102}f_2 + 2^{153}f_3 + 2^{204}f_4$.
- 8×1-way: One subroutine performs 8 parallel independent field operations.
- We formally verified our field implementations using CryptoLine.

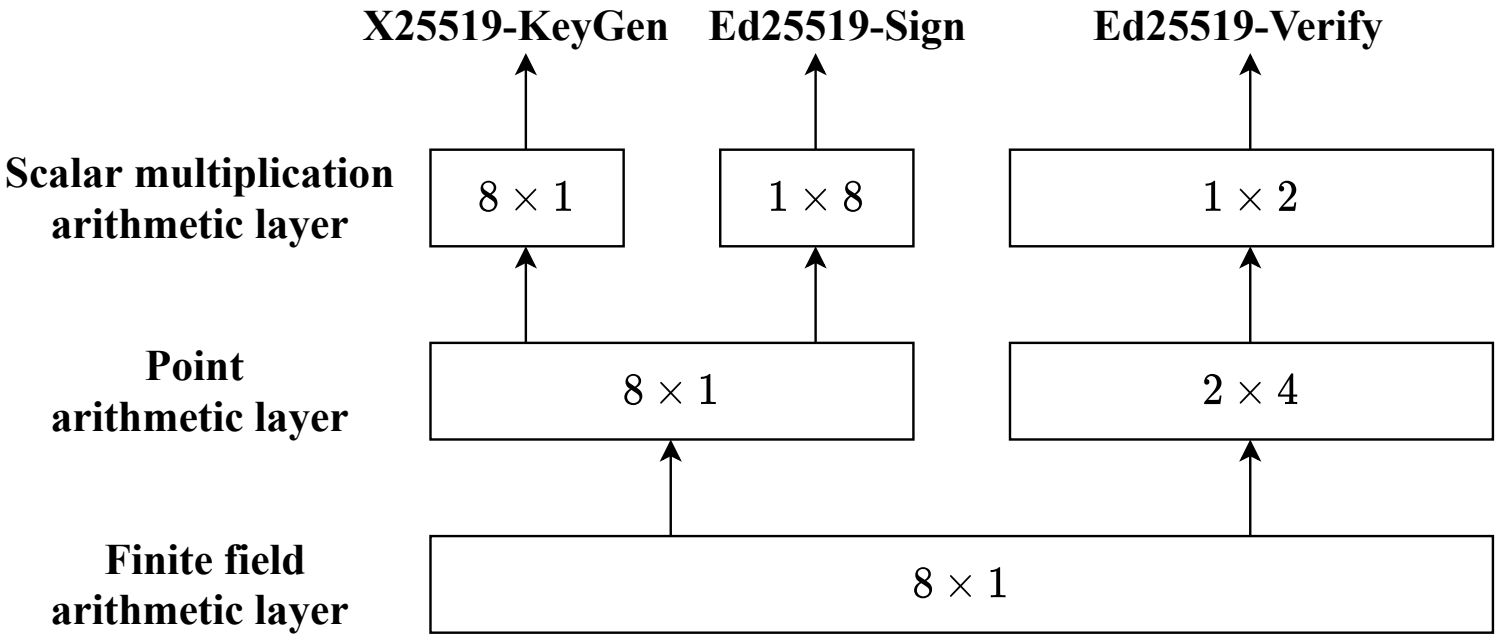


Figure: An overview of our X/Ed25519 implementation.

Optimized X25519 and Ed25519 implementation

Strategy: “finite field arithmetic” $\rightarrow$ “point arithmetic” $\rightarrow$ “scalar multiplication”

- X25519-KeyGen: $8 \times 1 \rightarrow 8 \times 1 \rightarrow 8 \times 1$
  - 12 times that of the OpenSSL implementation and 2.32 times that of Cheng et al.’s implementation.
- X25519-Derive: We don’t provide a faster X25519-Derive implementation than Hisil et al.
- Ed25519-Sign: $8 \times 1 \rightarrow 8 \times 1 \rightarrow 1 \times 8$
  - 3.79 times that of the OpenSSL implementation and 1.18 times that of Faz-Hernández et al.’s implementation.
- Ed25519-Verify: $8 \times 1 \rightarrow 2 \times 4 \rightarrow 1 \times 2$
  - 3.33 times that of the OpenSSL implementation and 1.33 times that of Faz-Hernández et al.’s implementation.

ENG25519: An OpenSSL ENGINE

- ENG25519 is based on OpenSSL ENGINE APIs, libsuola, and engntru.
- Our optimized X/Ed25519 implementations can be transparently integrated into OpenSSL and TLS applications through ENG25519.

Table: Detailed configuration of ENG25519.

Subroutine	Implementation
X25519-KeyGen Ed25519-KeyGen	Our $8 \times 1 \rightarrow 8 \times 1 \rightarrow 8 \times 1$ impl. $batch\text{-}size = 16$
X25519-Derive	$4 \times 2 \rightarrow 1 \times 4$ impl. of Hisil et al.
Ed25519-Sign	Our $8 \times 1 \rightarrow 8 \times 1 \rightarrow 1 \times 8$ impl.
Ed25519-Verify	Our $8 \times 1 \rightarrow 2 \times 4 \rightarrow 1 \times 2$ impl.

ENG25519: How to mitigate the cold-start issue?

Code start issue

- The processor will set the upper parts of the AVX2/AVX-512 vector units to a low-power mode to save power if the units are not in use for about 675 μs, leading to a warm-up phase of approximately 14 μs (56,000 clock cycles at 4 GHz) when an AVX2/AVX-512 instruction is executed in the low-power mode.
- During the warm-up phase, the throughput of the related instructions is 4.5 times slower than usual.
- All X/Ed25519 primitives suffer from varying degrees of performance degradation; especially the X25519-KeyGen takes 3.8 times longer in the DoT scenario than in the warm-start scenario.

ENG25519: How to mitigate the cold-start issue?

We designed a heuristic auxiliary thread that performs different actions based on the application's varying load conditions.

- Low-load scenarios: It takes no action to avoid disrupting the processor's power-saving strategies.
- Medium-load: It periodically executes a vector instruction.
- High-load: The frequent cryptographic operations inherently prevent entering low-power mode.

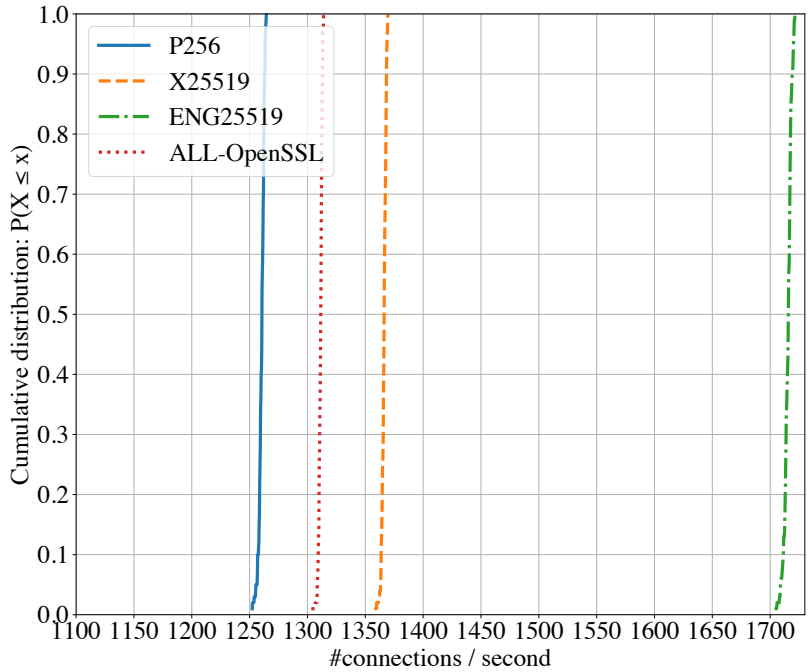
Table: Amortized CPU cycles (CC) to generate a keypair.

Batch size	Amortized CC with auxiliary thread	Amortized CC without auxiliary thread
1	10,315	28,450
4	9,107	19,388
8	9,003	14,108
16	8,980	11,406

ENG25519: Benchmark of TLS handshake

Client: tls_timer $\leftrightarrow$ Server: OpenSSL s_server

On average, the proposed ENG25519 setting (1,707 #connections/second) enables 25% and 35% more handshakes per second than X25519 (1,366) and P256 (1,260), respectively.



ENG25519: Benchmark of DoT query

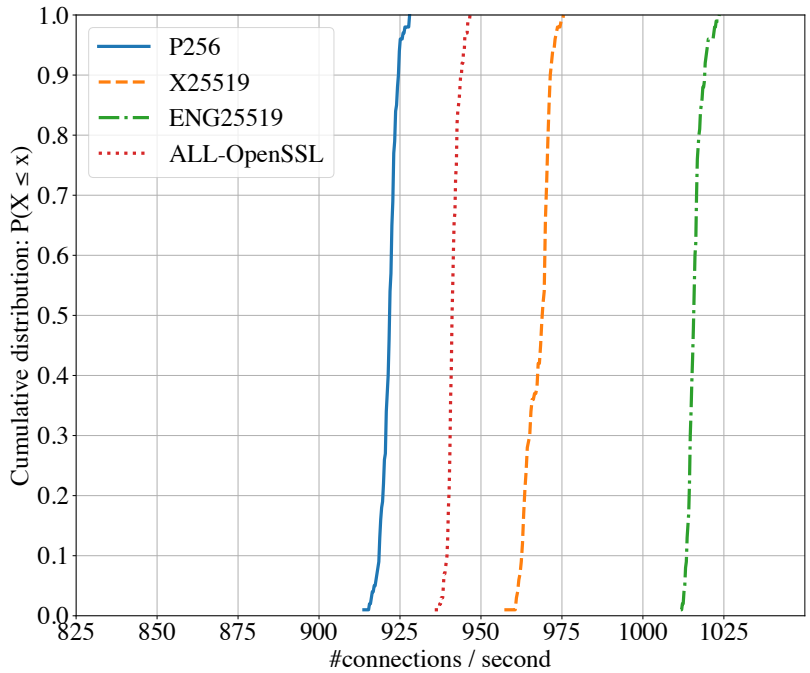
Client: dot_timer $\leftrightarrow$ Server: unbound DoT server[1]

End-to-end experiments

- Our ENG25519 outperforms all other configurations.

Peak throughput

- Our ENG25519 configuration achieved a significant improvement, achieving 290,315 #queries/min, which represents a 41% and 24% increase over P256 (206,275) and X25519 (234,875), respectively.



[1] https://nlnetlabs.nl/projects/unbound/about/

Conclusions

- Faster X/Ed25519 implementation using AVX-512IFMA.
- Integration of optimized X/Ed25519 implementations into TLS; faster TLS 1.3 handshake; increased DNS over TLS throughput.
- Under cold start conditions, some primitives may suffer a performance degradation of up to 3.8 times. If the vector implementation does not achieve significant improvements, a reevaluation of the vector implementation versus the x64 implementation is necessary.
- Open source artifact: `https://github.com/Ji-Peng/eng25519_artifact`.

Publications

- **Jipeng Zhang**, Yuxing Yan, Junhao Huang, Çetin Kaya Koç. Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}. IACR TCHES 2025. CCF-B.
- **Jipeng Zhang**, Junhao Huang, Lirui Zhao, Donglong Chen, Çetin Kaya Koç. ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519. Usenix Security 2024. Distinguished Award. CCF-A.
- **Jipeng Zhang**, Junhao Huang, Xuan Yu, et al. Research on Efficient Implementation of SM2 for Mobile Devices. Acta Electronica Sinica. CCF-A.
- **Jipeng Zhang**, Junhao Huang, et al. Time-memory Trade-offs for Saber+ on Memory-constrained RISC-V. IEEE Trans. on Computers. CCF-A.
- **Jipeng Zhang**, et al. An Efficient and Scalable Sparse Polynomial Multiplication Accelerator for LAC on FPGA. ICPADS2020. CCF-C.
- Junhao Huang, **Jipeng Zhang**, et al. Improved Plantard arithmetic for lattice-based cryptography. IACR TCHES 2022. CCF-B.
- Junhao Huang, Alexandre Adomnici, **Jipeng Zhang**, et al. Revisiting Keccak and Dilithium Implementations on ARMv7-M. IACR TCHES 2024. CCF-B.

Thanks

Thanks for listening