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]CCST, Nanjing University of Aeronautics and Astronautics
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

Abstract

The IETF released RFC 8446 in 2018 as the new TLS 1.3 standard, which recommends using X25519 for key exchange and Ed25519 for identity verification. These computations are the most time-consuming steps in the TLS handshake. Intel introduced AVX-512 in 2013 as an extension of AVX2, and in 2018, AVX-512IFMA, a submodule of AVX-512 to further support 52-bit (integer) multipliers, was implemented on Cannon Lake CPUs.

This paper first revisits various optimization strategies for ECC and presents a more performant X25519/Ed25519 implementation using the AVX-512IFMA instructions. These optimization strategies cover all levels of ECC arithmetic, including finite field arithmetic, point arithmetic, and scalar multiplication computations. Furthermore, we formally verify our finite field implementation to ensure its correctness and robustness.

In addition to the cryptographic implementation, we further explore the deployment of our optimized X25519/Ed25519 library in the TLS protocol layer and the TLS ecosystem. To this end, we design and implement an OpenSSL ENGINE called ENG25519, which propagates the performance benefits of our ECC library to the TLS protocol layer and the TLS ecosystem. The TLS applications can benefit directly from the underlying cryptographic improvements through ENG25519 without necessitating any changes to the source code of OpenSSL and applications. Moreover, we discover that the cold-start issue of vector units degrades the performance of cryptography in TLS protocol, and we develop an auxiliary thread with a heuristic warm-up scheme to mitigate this issue.

Finally, this paper reports a successful integration of the ENG25519 into an unmodified DNS over TLS (DoT) server called `unbound`, which further highlights the practicality of the ENG25519. We also report benchmarks of TLS 1.3 handshake and DoT query, achieving a speedup of 25% to 35% for TLS 1.3 handshakes per second and an improvement of 24% to 41% for the peak server throughput of DoT queries.

1 Introduction

In the TLS 1.3 standard [46] released by IETF in 2018, the elliptic curve cryptographic (ECC) algorithms X25519 [2] and Ed25519 [6] are recommended as the standard key exchange and identity verification algorithms. These two public-key cryptographic (PKC) algorithms are the security-critical and time-consuming steps in the TLS handshake.

In today's society, people are becoming increasingly reliant on online services, such as e-commerce and social media. For these online services, the challenge of slower service responses or even service outages due to increased load is something that service providers have to contend with, especially during times of surging user access. For instance, during events like Black Friday sales, website traffic can surge to levels 30 times higher than usual [51], and social media services can experience crashes due to sudden spikes in user traffic [1, 29]. Additionally, DNS and its privacy-enhanced variant, DNS over TLS (DoT), serving a massive user base, are integral parts of the internet's infrastructure [16].

These online services undoubtedly rely on the TLS protocol to secure communication. However, the TLS handshake, as the core of the entire protocol, involving complex cryptographic computations, poses a potential bottleneck limiting the quality improvements of these online services. This paper aims to address the challenges of mitigating the computational burden introduced by the TLS handshake, especially in high-load server scenarios. This paper takes a bottom-up methodology to solve this problem, starting from optimizing cryptographic algorithms at the lowest level, proceeding to the OpenSSL layer, and ultimately reaching the TLS application layer.

1.1 Motivations

The motivations of this paper, from the lower cryptographic layer to the higher application layer, are as follows:

(1) Previous work has extensively explored the efficient ECC implementation using the advanced Single Instruction

Multiple Data (SIMD) instructions on various CPUs, including NEON instructions on ARM CPUs and AVX2/AVX-512 instructions on Intel CPUs (e.g. [7, 15, 18, 25, 35]). But only a few of them (e.g. [18, 25]) focused on optimizing ECC using the latest AVX-512 on Intel CPUs. Additionally, the existing ECC optimizations using AVX-512 did not maximize the parallelism potential of AVX-512 or utilize the more powerful 52-bit (integer) multipliers in AVX-512IFMA (Integer Fused Multiply-Add). Therefore, we try to fill this gap by utilizing the powerful AVX-512IFMA to speed up cryptographic computations, thus alleviating the computational burden on servers.

(2) While many implementations have sought to improve the efficiency of ECC, they often neglect the integration of optimized ECC into TLS libraries and the broader TLS ecosystem. As a result, the TLS ecosystem cannot actually benefit from these ECC optimizations at large. Therefore, investigating how to integrate the optimized ECC implementation into a TLS application without modifying its source code would enable us to explore the tangible performance enhancements that the low-level ECC optimizations provide for high-level TLS applications. We aim to constitute a systematic study covering not only cryptographic optimization but also the integration into the TLS ecosystem.

(3) When integrating the optimized ECC implementation into the TLS ecosystem, we find that the vector units' (e.g. AVX2 or AVX-512) cold-start issue leads to noticeable performance degradation of cryptographic primitives. Based on our observations, the relevant primitives are even 3.8× slower than usual. This finding motivates us to systematically examine and alleviate the adverse effects of the cold-start issue on real-world TLS applications.

(4) The TLS protocol has become a cornerstone of safeguarding privacy in today's digital landscape. The TLS handshake, as the central component of the TLS protocol, involving complex cryptographic computations, serves as a potential bottleneck, constraining service quality improvements in various contexts like DoT, e-commerce, and social media. This motivates us to address the above-mentioned issues and develop a solution to mitigate the computational burden caused by the TLS handshake.

1.2 Major contributions

Our artifact is available at https://github.com/Ji-Peng/eng25519_artifact. This work has four main contributions.

First, we investigate how to maximize the parallelism potential of AVX-512IFMA and utilize its 52-bit multipliers to speed up the ECC arithmetic, including finite field arithmetic, point arithmetic, and scalar multiplication. To achieve this, we adopt a radix-2^{51} big number representation tailored for the 52-bit multiplier of AVX-512IFMA, we develop an efficient finite field arithmetic implementation, whose correctness is formally verified using the semi-automatic tool CRYPTOLINE. Besides, we show that by adopting the 8-way parallelism in the low-level finite field arithmetic, we can explore more parallelism strategies (see Figure 2 for the overview of our strategies) for the upper layers of ECC. With these optimizations, we achieve new speed records for X25519-KeyGen, Ed25519-Sign, and Ed25519-Verify. However, the X25519-Derive function used to compute the shared secret key is limited by its inherent execution flow and data dependencies; thus, we cannot achieve a faster implementation.

Then, we design and implement an OpenSSL ENGINE called ENG25519 to speed up the TLS ecosystem. The ENG25519 could transparently integrate our optimized cryptographic library into OpenSSL and, ultimately, into TLS applications. We demonstrate the successful integration of ENG25519 into `unbound`, a DNS over TLS (DoT) server, to further illustrate its practicality and conduct a systematic study under the DoT scenario. Compared with other related work ([5, 9]), the ENG25519 is superior to `libsuola` in [9], as an ENGINE template, in integrating ECC implementation to the TLS layer, because it has been verified to successfully reach the TLS application layer.

Then, we evaluate the performance of the cryptographic algorithms in realistic TLS application scenarios rather than just using ideal warm-start benchmark testing. We accomplish this by testing the DoT query scenario, concluding that these cryptographic computations are not warm-started, and performance suffers to varying degrees of degradation compared to the warm-start benchmark. Some subroutines even take 3.8× longer than their warm-start CPU cycles, making the cryptographic implementers "get half the results with twice the effort". To resolve the cold-start issue in realistic TLS application scenarios, we developed an auxiliary thread with a heuristic warm-up scheme for preventing vector units from entering a low-power mode. Besides, our $8 \times$ 1-way X25519-KeyGen is utilized with a focus on cache friendliness and optimal performance. To achieve this, we call the key generation subroutine multiple times when necessary and save the extra keypairs for future use, as the key generation is independent of the communication peer's information and can be precomputed for enhanced cache friendliness[1] and efficiency.

Finally, in the TLS application layer, we achieve a speedup of 25% to 35% for TLS 1.3 handshakes per second and an improvement of 24% to 41% for the peak server throughput of DoT queries. This implies that this paper presents an effective solution to mitigate the computational burden imposed by TLS handshakes for throughput-critical online services. Besides, our study demonstrates that the parallel strategies employed in our ECC implementation, the ENG25519 framework, and the proposed heuristic warm-up scheme for mitigating the cold-start issue of vector units can be extended to related work in this field, and our findings have broad implications and can contribute to parallel computing in ECC and

[1] The cache friendliness involves L1 instruction and L1 data cache of CPU.

other related areas.

2 Background

This section introduces the relevant technical background. In Section 2.1, we briefly introduce the AVX-512 instructions. Next, we describe the algorithmic background of X25519 and Ed25519 in Section 2.2. Finally, we introduce the TLS 1.3 handshake and DNS over TLS (DoT) in Section 2.3.

2.1 AVX-512

As the latest version of the AVX instruction set, AVX-512 provides double-sized vectors compared to AVX2. AVX-512 provides 32 512-bit registers that enable flexible operation for a greater range of parallel data, such as eight 64-bit or four 128-bit lanes.

Apart from the core extension AVX-512F (Foundation), which all AVX-512 implementations require, AVX-512 supports many submodules, such as AVX-512IFMA (Integer Fused Multiply-Add). One of AVX-512IFMA's critical features is the 52-bit integer multiplier, which is the widest multiplier in the SIMD instructions of the current x86-64 series CPUs. Indeed, AVX2 and AVX-512F multiplier have a width of only 32 bits.

AVX-512 introduces new concepts, such as mask operations and broadcasts. The mask operation allows for precise control over the specific behavior of SIMD instructions. The mask instruction specifies the lanes that should be executed or disregarded for a given operand. For more details about AVX-512IFMA instructions, we refer to Appendix A.

2.2 X25519 and Ed25519

X25519 and Ed25519 are constructed from the Montgomery [33] and twisted Edwards [4] curves, respectively. In 2015, RFC 7748 [30] introduced the construction of the Diffie-Hellman protocol based on Curve25519 with 128-bit security, referred to as X25519. X25519 has two stages: key generation (KeyGen) and shared secret calculation (Derive). The core operation of KeyGen is a fixed-point scalar multiplication, and the core operation of Derive is a variable-point scalar multiplication (see [30] for more details).

In 2017, the Edwards-curve Digital Signature Algorithm (EdDSA) named Ed25519, with the same security level as X25519, was introduced in RFC 8032 [28]. It is suggested that Curve25519 is birationally equivalent to Edwards25519, the underlying curve of Ed25519 [6, Sec 2]. Ed25519 consists of three stages: key generation (KeyGen), signature generation (Sign), and signature verification (Verify). The core operation of KeyGen and Sign is a fixed-point scalar multiplication, while the core operation of Verify is a double-point scalar multiplication (see [28] for more details). Later in 2018, RFC 8446 [46] included X25519 and Ed25519 into the cipher suite supported by TLS 1.3.

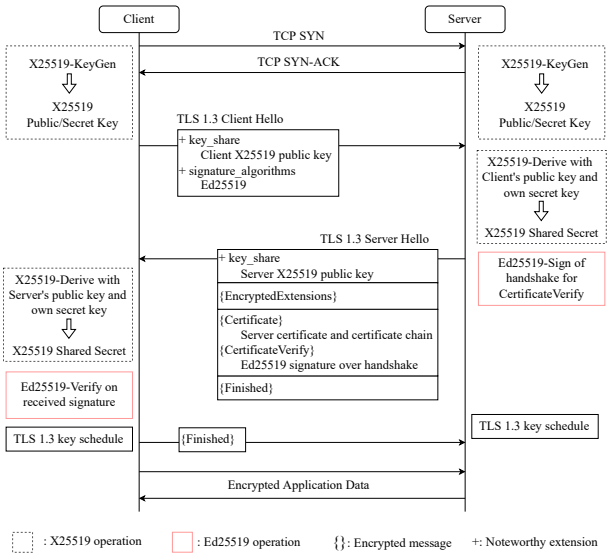


Figure 1: Overview of TLS 1.3 handshake with X25519 for key exchange and Ed25519 for digital signature

The efficient coordinates for implementing the point arithmetic of Edwards25519 are the extended twisted Edwards coordinates [26, Sec 3], which can be expressed as (X,Y,T,Z). Here, T is an auxiliary coordinate that holds $T = XY/Z$. The identity element is $(0,1,0,1)$, and the negative element of (X,Y,T,Z) is $(-X,Y,-T,Z)$.

To avoid confusion among the three terms: Curve25519, X25519, and Ed25519, Bernstein standardized their description in [3], describing "X25519" as the Diffie-Hellman key exchange system on Montgomery curve, "Ed25519" as elliptic curve signature system on twisted Edwards curve, and "Curve25519" as the underlying elliptic curve of X25519/Ed25519.

2.3 TLS 1.3 and DNS over TLS (DoT)

To illustrate the basic flow of the TLS 1.3 handshake, we present a simplified flow chart (see Figure 1) of the TLS 1.3 handshake, assuming that X25519 is used for key exchange and Ed25519 is used for digital signature. The handshake is primarily accomplished through the `Client Hello` and `Server Hello` messages and related extensions. The cryptographic operations performed by the client include X25519-KeyGen, X25519-Derive, and Ed25519-Verify, while the server's cryptographic operations include X25519-KeyGen, X25519-Derive, and Ed25519-Sign. The X25519 public key is transmitted through the `key_share` extension, and the Ed25519 signature is sent via the `CertificateVerify` extension. For more relevant details, we refer readers to the RFC 8446 [46].

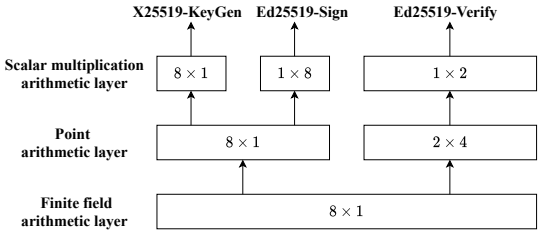


Figure 2: An overview of our ECC implementation. Please refer to Section 3.1 for the meaning of $m \times n$-way operation.

DNS over TLS (DoT) is specified in RFC 7858 [27], which can protect user privacy by encrypting and wrapping DNS queries and responses via TLS. Given the security guarantees of the TLS layer, DoT can impede the monitoring and analysis of DNS traffic. The workflow of DoT is that the client and server establish a TLS connection via a TLS handshake, and then DNS queries and responses are sent and received through this TLS connection. A DoT client can establish a TLS connection with a DoT server via a TLS handshake, only to send a DNS query and close the connection after receiving the corresponding DNS response. In this case, the overhead proportion of the TLS handshake will be more significant, highlighting the role of the handshake's speed optimization.

3 Optimized X25519 and Ed25519 Implementation

This section presents our bottom-up optimized implementation of X25519 and Ed25519. Our optimizations cover all layers of the ECC implementation, including finite field arithmetic, point arithmetic, and scalar multiplication. The finite field arithmetic consists primarily of the basic operations in $\mathbb{F}_p$, such as addition, subtraction, multiplication, square, and inversion modulo p. The point arithmetic includes point addition and point doubling on elliptic curves. The scalar multiplications involves fixed-point, variable-point, and double-point scalar multiplications.

Each layer of ECC implementation has different parallelism strategies. Figure 2 shows an overview of the proposed ECC implementation. To clarify the description, we provide a parallelism hierarchy division method and a vital optimization principle — the MTP principle — in Section 3.1. Overall, we implement the 8×1-way finite field arithmetic with the 52-bit multiplier of AVX-512IFMA in Section 3.2, maximizing its parallelism. Then, we further show that the underlying 8×1-way finite field arithmetic can transfer more parallelism to the upper layer, i.e. 8×1-way and 2×4-way point arithmetic in Section 3.3. Based on these point arithmetic implementations, we further implement the 8×1-way fixed-point scalar multiplication, 1×8-way fixed-point scalar multiplication in Section 3.4, and 2×4-way double-point scalar multiplication in Section 3.5 for X25519-KeyGen, Ed25519-Sign, and Ed25519-Verify, respectively. A comparison with other work is given in Section 3.6.

Key generation is an independent process that does not rely on information from the communication peer during the TLS handshake (see Figure 1). Therefore, one could generate eight different keypairs by calling our 8×1-way X25519-KeyGen subroutine once and save them in the machine. When a new keypair is needed for the next TLS handshake, it can be retrieved directly from the remaining keypairs without executing X25519-KeyGen again. If no keypairs are left, the machine will generate a new batch.

However, X25519-Derive, Ed25519-Sign, and Ed25519-Verify all rely on the message of the communication peer, making their 8×1-way implementation difficult to integrate into the TLS layer. Using an asynchronous programming framework to collect eight requests asynchronously and then perform the 8×1-way operation requires refactoring the TLS applications' source code. This is contrary to the original intention of this paper, namely benefiting the TLS applications from cryptographic optimization without modifying their source code. Thus, integrating other 8×1-way implementations into the TLS layer will be our future work. We only present the optimized 8×1-way X25519-KeyGen, 1×8-way Ed25519-Sign and 1×2-way Ed25519-Verify in this paper.

3.1 Parallelism hierarchy

We use the paradigm "finite field arithmetic" $\rightarrow$ "point arithmetic" $\rightarrow$ "scalar multiplication" to represent our parallelism strategy. For example, our optimized Ed25519-Verify strategy is $8 \times 1 \rightarrow 2 \times 4 \rightarrow 1 \times 2$ (cf. Figure 2), which can also be abbreviated as 1×2-way Ed25519-Verify. X25519-Derive is an exception since the variable-point scalar multiplication (i.e. Montgomery ladder algorithm) is directly contructed on top of the finite field layer. Thus, the paradigm of X25519-Derive is "finite field arithmetic" $\rightarrow$ "Montgomery ladder".

The AVX-512's 512-bit registers allow the 8×64-bit partition (i.e. 8 lanes, each of which holds 64 bits), and these 8 lanes can be used to implement 8×1, 4×2, 2×4, or 1×8-way finite field arithmetic.

- **Interpretation of parallelism**. In the implementation that follows $m \times n$ parallelism, "m" represents the number of independent operations simultaneously executed in parallel within this layer, while "n" refers to the degree of acceleration attained by each independent operation. Taking the 4×2-way finite field implementation as an example, "4" means that four independent finite field operations are executed in parallel, and "2" means that 2-way acceleration is performed inside each finite field operation.
- **Transfer of parallelism.** The "m" indicates the number of independent operations occurring at this layer that

can be transferred to the upper layer; "n" signifies the amount of parallelism digested by this layer. Take the 4×2-way finite field implementation as an example. The four independent arithmetic of the finite field layer can be passed upwards to the point arithmetic layer, namely implementing the point arithmetic in the form of 4×1-way, 2×2-way, or 1×4-way.

- **MTP optimization principle.** We notice that if we digest "n" parallelism in the low-level finite field implementation, we will suffer from performance loss caused by permutation instructions. See Appendix B for a detailed comparison between the 4×2-way and 8×1-way finite field multiplication implementations. In brief, the 4×2-way finite field multiplication needs some awkward permutation instructions to construct the required execution flow, which will suffer a performance penalty due to the high latency of permutation instructions. In contrast, we can naturally construct the execution flow of 8×1-way multiplication, minimizing the permutation performance penalty. We propose the Maximized Transfer Parallelism (MTP) principle which aims to maximize the transfer of SIMD parallelism to the upper layer and avoid wasting parallelism in lower level layer.

3.2 8×1-way finite field arithmetic

Unlike the AVX2 implementation of Cheng et al. [15], which adopts the radix-2^{29} representation to implement their 4×1-way X25519. We employ the radix-2^{51} representation to fully utilize the 52-bit multiply-add instructions in AVX-512IFMA. Note that our method is not a straightforward generalization of [15] because they utilize the 32-bit multiplier of AVX2, while we utilize the 52-bit multiplier of AVX-512IFMA. Therefore, we need to re-design the finite field arithmetic.

The radix-2^{51} method divides the 255-bit elements into five 51-bit limbs and a finite field element f is represented as:

$$f = f_0 + 2^{51} f_1 + 2^{102} f_2 + 2^{153} f_3 + 2^{204} f_4 \tag{1}$$

where $0 \le f_i < 2^{52}$ for $0 \le i < 5$. Among all the finite field arithmetic, modular multiplication is an essential operation. The modular multiplication $f \times g$ consists of two steps: multiplication and modular reduction.

Step 1. Multiplication.

$$\begin{aligned} &h = f \times g = h_0 + 2^{51} h_1 + \cdots + 2^{51\cdot 8} h_8 + 2^{51\cdot 9} h_9, \\ &h_k = \sum_{i+j=k} (f_i g_j)_l + 2 \sum_{i+j=k-1} (f_i g_j)_h, \\ &\quad (0 \le i, j < 5,\ 0 \le k < 10) \end{aligned} \tag{2}$$

Step 2. Modular reduction. Its basic idea is as follows:

$$\begin{aligned} &p = 2^{51\cdot 5} - 19 \ \Rightarrow\ 2^{51\cdot 5} \equiv 19 \mod p \\ &2^{51\cdot 5} h_5 = 19 h_5, \cdots, 2^{51\cdot 9} h_9 = 2^{51\cdot 4} 19 h_9 \\ &h_0 \mathrel{+}= 19 h_5, h_1 \mathrel{+}= 19 h_6, \cdots, h_4 \mathrel{+}= 19 h_9 \end{aligned}$$

Then, the modular reduction can be performed in the following order:

$$\begin{aligned} &①h_5 \to h_6, ②h_6 \to h_7, ③h_7 \to h_8, ④h_8 \to h_9, \\ &⑤19^2(h_9 >> 51) \to h_0|h_1, ⑥19h_5 \to h_0|h_1, \\ &⑦19h_6 \to h_1|h_2, ⑧19h_7 \to h_2|h_3, ⑨19h_8 \to h_3|h_4, \\ &⑩19h_9 \to h_4|h_5, ⑪19h_5 \to h_0 \end{aligned} \tag{3}$$

In step 1, the 52-bit multiplier separates the lower 52-bit and upper 52-bit of the 104-bit product into two registers. In Equation 2, the subscript l and h represents the lower 52-bit and upper 52-bit of the product, respectively. To fit with the radix-2^{51} representation, the upper 52-bit $(f_i g_j)_h$ is converted from $2^{52} 2^{51\cdot(i+j)} (f_i g_j)_h$ to $2^{51\cdot(i+j+1)} 2 (f_i g_j)_h$, and is accumulated to the corresponding limb.

In step 2, the paradigm $h_5 \to h_6$ means replacing (h_5, h_6) with $(h_5 \bmod 2^{51}, h_6 + \lfloor h_5 / 2^{51} \rfloor)$; the paradigm $19^2(h_9 >> 51) \to h_0|h_1$ means replacing (h_0, h_1) with $(h_0 + (19^2(h_9 >> 51))_l, h_1 + 2(19(h_9 >> 51))_h)$. The bit length of $h_i (i = 5, \cdots, 9)$ may exceed 52 before the modular reduction. To utilize the 52-bit multiplier, we must propagate the carry that exceeds 51-bit to the higher limb. After the carry propagation (i.e. ① to ⑤), the modular reduction of ⑥ to ⑪ is performed.

The carry chain has many data dependencies that cause extra latency to the computational process. Therefore, we adjust the carry order and perform it in the following order: ①③⑤②④⑥⑧⑩⑦⑨⑪. Apart from the modular multiplication, the carry propagation of modular addition and subtraction undergoes similar optimization.

In terms of data type, we adopt similar terminology from [15, Sec 3.1]. The fundamental data type used in the 8×1-way finite field arithmetic is the vector set V, which consists of 5 vectors (i.e. 512-bit registers) v_i $(0 \le i < 5)$, and each vector contains 8 limbs from different finite field elements. The vector set V is defined as:

$$\begin{aligned} V &= [a, b, \cdots, g, h] \\ &= \left[\sum_{i=0}^{4} 2^{51 i} a_i, \sum_{i=0}^{4} 2^{51 i} b_i, \cdots, \sum_{i=0}^{4} 2^{51 i} g_i, \sum_{i=0}^{4} 2^{51 i} h_i \right] \\ &= \sum_{i=0}^{4} 2^{51 i} [a_i, b_i, \cdots, g_i, h_i] = \sum_{i=0}^{4} 2^{51 i} v_i \end{aligned} \tag{4}$$

where $v_i = [a_i, b_i, \cdots, g_i, h_i]$, and all of the $a_i, b_i, \cdots, g_i, h_i$ are 51-bit or 52-bit limbs. Each register comprises eight 64-bit lanes, with each lane storing a 51-bit or 52-bit limb. In the 8×1-way finite field arithmetic, eight limbs in one register come from eight different finite field elements. Based on this data structure, we implemented the 8×1-way modular addition, subtraction, multiplication, square, and inversion using AVX-512IFMA.

Formal verification of our finite field implementation The finite field implementation is critical for performance

Algorithm 1 2×4-way point addition.

Input: $[P,J] = [X_P, Y_P, T_P, Z_P, X_J, Y_J, T_J, Z_J]$; $[Q^{pre}, K^{pre}] = [Y_Q - X_Q + 2p, Y_Q + X_Q, 2dT_Q, 2Z_Q, Y_K - X_K + 2p, Y_K + X_K, 2dT_K, 2Z_K]$; P, Q, J, and K are points on Edwards25519; Q^{pre} is the precomputed format of Q, which is generally obtained from a precomputed table.

Output: $[R,S] = [X_R, Y_R, T_R, Z_R, X_S, Y_S, T_S, Z_S]$ such that $R = P+Q$, $S = J+K$.

$M \leftarrow \text{PERM}([P,J], 0\text{xB1})$
$N \leftarrow \text{MADD}([P,J], 0\text{x11}, M, 2p)$
$N \leftarrow \text{MSUB}(N, 0\text{x11}, N, [P,J])$
$N \leftarrow \text{MADD}(N, 0\text{x22}, N, M)$
$U \leftarrow N \times [Q^{pre}, K^{pre}]$ ▷ 8×1-way multiplication
$M \leftarrow \text{PERM}(U, 0\text{xDD})$
$N \leftarrow \text{PERM}(U, 0\text{x88})$
$M \leftarrow \text{MADD}(M, 0\text{x99}, M, 2p)$
$M \leftarrow \text{MSUB}(M, 0\text{x99}, M, N)$
$M \leftarrow \text{MADD}(M, 0\text{x66}, M, N)$
$N \leftarrow \text{PERM}(M, 0\text{x4B})$
$[R,S] \leftarrow M \times N$ ▷ 8×1-way multiplication
return $[R,S]$

and security in ECC. Even minor errors in the finite field implementation can lead to severe consequences such as cryptographic attacks or functional failures [8]. We utilize a semi-automatic formal verification tool, CRYPTOLINE [21, 32], to provide mathematical guarantee of its correctness. For more details, see the artifact of this work.

3.3 8×1-way and 2×4-way point arithmetic

X25519-KeyGen is always computed on the Edwards curve, and then the result is converted to the equivalent Montgomery curve. X25519-Derive generally adopts the Montgomery ladder algorithm, and we can not achieve a new speed record. We therefore only focus on the point arithmetic on the Edwards curve. For the point addition formula on twisted Edwards curves, giving $P = (X_P, Y_P, T_P, Z_P)$ and $Q = (X_Q, Y_Q, T_Q, Z_Q)$ in the format of extended twisted Edwards coordinates [26, Sec 3], $R = P+Q = (X_R, Y_R, T_R, Z_R)$ is calculated as follows:

$$\begin{array}{ll} A \leftarrow (Y_P - X_P) \times (Y_Q - X_Q), & B \leftarrow (Y_P + X_P) \times (Y_Q + X_Q), \\ C \leftarrow 2d \times T_P \times T_Q, & D \leftarrow 2Z_P \times Z_Q, \\ E \leftarrow B - A, & F \leftarrow D - C, \\ G \leftarrow D + C, & H \leftarrow B + A, \\ X_R \leftarrow E \times F, & Y_R \leftarrow G \times H, \\ Z_R \leftarrow F \times G, & T_R \leftarrow E \times H. \end{array} \tag{5}$$

By scheduling eight independent point additions (i.e. above formula) into each lane of the 8×1-way finite field arithmetic, we arrive at $8 \times 1 \rightarrow 8 \times 1$. The 8×1-way point doubling follows a similar approach.

Faz-Hernández et al. [18, Alg 4] implemented a 1×4-way point addition based on 4×1-way finite field arithmetic using AVX2 instructions. However, limited by its calculation flow, the point addition cannot be extended to 1×8-way. Consequently, we design a 2×4-way point addition, shown in Algorithm 1, to handle this shortcoming. In Algorithm 1, $+2p$ ensures that the result of subtraction remains within the non-negative range. Whereby the permutation instruction PERM and mask instruction MADD/MSUB are mentioned in Appendix A. The 2×4-way point doubling is computed in a similar manner.

3.4 8×1-way and 1×8-way fixed-point scalar multiplication

We adopt the terminology of searching a point from precomputed tables in [18, Sec 4.1]. For a fixed-point scalar multiplication kP, where P is known beforehand, precomputing several multiples of P can effectively reduce the computational overhead. In this case, the scalar k is a 253-bit[2] integer. Let $t = \lceil 253/\omega \rceil$ with $\omega > 0$. We precompute a set of t tables; each contains $2^{\omega-1}$ points. These tables are defined as $T_u = \{T_u(v) = 2^{\omega u} vP \mid \text{for } 1 < v \leq 2^{\omega-1}\}$ for $0 \leq u < t$. Searching for a point from these tables is defined as:

$$\phi(T_u, v) = \begin{cases} T_u(v), & \text{if } v > 0 \\ -T_u(-v), & \text{if } v < 0 \\ O, & \text{otherwise .} \end{cases} \tag{6}$$

The scalar k is recoded into signed segments $(k_0, \cdots, k_{t-1})$, such that $k = \sum_{j=0}^{t-1} 2^{\omega j} k_j$ and $-2^{\omega-1} \leq k_j < 2^{\omega-1}$ (see [18, Alg 5] for the recoding algorithm). This work employs $\omega = 4$ and $t = 64$ and only precomputes even-indexed tables (i.e. all T_u for even u) to reduce memory consumption. We utilize a secure and constant-time masking technique to search for points from these tables, which avoids secret-key indexing and potential side-channel attacks.

8×1-way implementation for X25519-KeyGen Using the 8×1-way implementation, eight independent scalar multiplications can be computed in parallel based on our 8×1-way point arithmetic. The calculation of each scalar multiplication is as follows.

$$kP = \sum_{j=0}^{31} \phi(T_{2j}, k_{2j}) + 2^4 \sum_{j=0}^{31} \phi(T_{2j}, k_{2j+1}). \tag{7}$$

Such a formula is viewed as a lane in our 8×1-way point addition. When eight independent scalar multiplications are computed in parallel, we arrive at $8 \times 1 \rightarrow 8 \times 1 \rightarrow 8 \times 1$.

[2] 253 is the bit length of the elliptic curve group order $\mathcal{L}$ of Curve25519. The scalar is guaranteed to lie within the range of $[0, \mathcal{L})$ before executing the scalar multiplication.

Algorithm 2 Optimized double-point scalar multiplication.

Input: (k,l,P,Q), where k and l are 253-bit[4] scalars, P and Q are points on Edwards25519 curve, P is a fixed point, and Q is a variable point. The precomputed table $tableP \leftarrow \{O,P,2P,\cdots,2^{\omega-1}P\}$, where ω is the width of the window, O is the identity element of elliptic curve group, and $tableP$ can be precomputed offline since P is a fixed point.

Output: $R = kP + lQ$.

1: Compute $tableQ \leftarrow \{O,Q,2Q,\cdots,2^{\omega-1}Q\}$ at runtime
2: $(k_0,k_1,\cdots,k_{r-1}) \leftarrow \text{Recoding}_\omega(k)$ ▷ $r = \lceil 253/\omega \rceil + 1$
3: $(l_0,l_1,\cdots,l_{r-1}) \leftarrow \text{Recoding}_\omega(l)$
4: $[M,N] \leftarrow [O,O]$
5: $[A,B] \leftarrow [\pm tableP[|k_{r-1}|], \pm tableQ[|l_{r-1}|]]$ ▷ The sign of the point taken from tables follows the sign of the corresponding scalar fragments (i.e. k_{r-1} and l_{r-1} here).
6: $[M,N] \leftarrow [M,N] + [A,B]$ ▷ Using our 2×4-way point addition.
7: **for** $i \leftarrow r-2$ to 0 **do**
8: $\quad [M,N] \leftarrow 2^\omega [M,N]$ ▷ Using our 2×4-way point doubling ω times.
9: $\quad [A,B] \leftarrow [\pm tableP[|k_i|], \pm tableQ[|l_i|]]$
10: $\quad [M,N] \leftarrow [M,N] + [A,B]$
11: **end for**
12: $R \leftarrow M + N$
13: **return** R

1×8-way implementation for Ed25519-Sign The calculation of kP using our 1×8-way strategy is shown as follows.

$$\begin{aligned} kP = &\sum_{j=0}^{7} \phi(T_{2j}, k_{2j}) + 2^4 \sum_{j=0}^{7} \phi(T_{2j}, k_{2j+1}) \\ &+ \sum_{j=8}^{15} \phi(T_{2j}, k_{2j}) + 2^4 \sum_{j=8}^{15} \phi(T_{2j}, k_{2j+1}) \\ &+ \sum_{j=16}^{23} \phi(T_{2j}, k_{2j}) + 2^4 \sum_{j=16}^{23} \phi(T_{2j}, k_{2j+1}) \\ &+ \sum_{j=24}^{31} \phi(T_{2j}, k_{2j}) + 2^4 \sum_{j=24}^{31} \phi(T_{2j}, k_{2j+1}), \end{aligned} \tag{8}$$

where the eight accumulators are computed in parallel by the 8×1-way point arithmetic and ultimately arrive at $8 \times 1 \rightarrow 8 \times 1 \rightarrow 1 \times 8$.

3.5 2×4-way double-point scalar multiplication

The core operation of Ed25519-Verify is a double-point scalar multiplication, i.e. $kP + lQ$, where k and l are scalars (i.e. 253-bit integers), P is a fixed point, and Q is a variable point.

Faz-Hernández et al. [18, Alg 6] adopted a w-NAF [47] method to compute the double-point scalar multiplication.

Table 1: The number of point addition (PA) and point doubling (PD) of our double-point scalar multiplication with different window widths.

ω	PD	PA	PA+0.97PD[1]
3	257	88	338
4	259	68	320
5	260	57	310
6	267	53	312
7	276	55	323

[1] The proposed 2×4-way point addition and point doubling implementation consume 181 and 176 CPU cycles, respectively in our machine.

Their parallelism strategy is $4 \times 1 \rightarrow 1 \times 4 \rightarrow 1 \times 1$ using AVX2, which shows that the point arithmetic layer digests all the parallelism. It is suboptimal according to the MTP principles. Moreover, they also admit (in [18, Sec 4.2]) that they cannot provide a more efficient strategy due to the inherent sequential pattern of the w-NAF method.

Instead of using the w-NAF method, we design a window-based method to break the parallelism limit. We describe the basic idea of the window-based method [23, Alg 3.41] with a simple example. Calculating scalar multiplication $51P$ with window width $\omega = 3$ and signed encoding is shown as follows. (1) Recode the scalar 51 into signed w-bit segments, i.e. $(1,-2,3)$ such that each value lies within the range of $[-2^{3-1}, 2^{3-1})$ and $1 \cdot 2^{3 \cdot 2} - 2 \cdot 2^{3 \cdot 1} + 3 \cdot 2^{3 \cdot 0} = 51$. (2) Scan segments from left to right and accumulate them. $R \leftarrow O, R \leftarrow R+1P, R \leftarrow 2^3 R, R \leftarrow R+(-2P), R \leftarrow 2^3 R, R \leftarrow R+3P$, where O is the identity element of elliptic curve group, "+" means point addition, and $2^3 R$ is completed by three times of point doublings. $1P$, $2P$, and $3P$ are all obtained from a precomputed table, $-2P$ is obtained by negating $2P$. For a fixed point, precomputation can be performed offline; for a variable point, precomputation has to be run online.

The starting point is to design a window-based method to construct the double-point scalar multiplication. Instead of computing $R \leftarrow R + iP$ solely, we compute $R \leftarrow R + iP$ and $R' \leftarrow R' + jQ$ in parallel for constructing a 2×4-way point arithmetic naturally, where i and j are scalar fragments. Finally, we arrive at $8 \times 1 \rightarrow 2 \times 4 \rightarrow 1 \times 2$. In this way, the finite field layer does not digest parallelism ahead of time. Two parallelisms are passed to the top-most scalar multiplication layer, thus minimizing the permutation performance penalty at the bottom layer.

We present our double-point scalar multiplication in Algorithm 2. The $\text{Recoding}_\omega()$ subroutine in lines 2 and 3 is used to recode a 253-bit scalar k into a series of signed segments such that $k = \sum_{i=0}^{r-1} 2^{\omega i} k_i$, where $r = \lceil 253/\omega \rceil + 1$, ω is the width of the window, and $-2^{\omega-1} \le k_i < 2^{\omega-1}$. See [18, Alg 5] for more details of the $\text{Recoding}_\omega()$ subroutine.

Table 2: The CPU cycles comparison of different X25519 implementations under warm-start conditions. Hisil et al. [25] and Nath et al. [35] only provided optimized X25519-Derive implementation. ISA/ISE stands for Instruction Set Architecture/Extension.

Reference	X25519-KeyGen			X25519-Derive			ISA/ISE
	Cycles	Ratio[1]	Strategy	Cycles	Ratio[2]	Strategy[3]	
OpenSSL [41]	90,809	12.01	$1\times1\rightarrow1\times1\rightarrow1\times1$	90,849	1.47	$1\times1\rightarrow1\times1$	x86-64
Faz-H. [18]	26,420	3.49	$4\times1\rightarrow4\times1\rightarrow1\times4$	71,454	1.15	$2\times2\rightarrow1\times2$	AVX2
Hisil [25]	-	-	-	61,867	1.00	$4\times2\rightarrow1\times4$	AVX-512F
Nath [35]	-	-	-	64,832	1.05	$4\times1\rightarrow1\times4$	AVX2
Cheng [15][4]	70,164	2.32	$4\times1\rightarrow4\times1\rightarrow4\times1$	-	-	-	AVX2
This work	60,479	1.00	$8\times1\rightarrow8\times1\rightarrow8\times1$	-	-	-	AVX-512IFMA

[1] Our 8×1-way X25519-KeyGen can generate eight different keypairs once. The implementation of Cheng et al. is a 4×1-way implementation; others are $1\times n$-way (one keypair once). Therefore, the ratio with Cheng et al. is $(70164/4)/(60479/8)=2.32$. Other calculations are similar to this.

[2] We don't provide a faster X25519-Derive implementation. All comparisons are made with Hisil et al.

[3] The strategy is represented as "finite field" $\rightarrow$ "Montgomery ladder".

[4] Cheng et al. provide a 4×1-way implementation of the X25519-Derive, but as Section 3 said, integrating it into the TLS ecosystem needs to re-engineer the TLS application so we do not present their cycles.

Determines the window's width ω The larger the value of ω, the larger the precomputation overhead of *tableQ* will be, and the number of point addition (PA) and point doubling (PD) in the main loop will be smaller. The computational cost of Algorithm 2 is mainly composed of the precomputation cost of *tableQ* and the main loop cost. The precomputation cost is $(2^{w-3}+1)PD+2^{w-3}PA$. The cost of the main loop is $w(r-1)PD+rPA$. Line 12 is a point addition to calculate the final result. We detail the computational cost for different ω in Table 1, and it is obvious that $\omega=5$ achieves the best performance.

3.6 Comparison

Note that the performance comparison in this section only considers the warm-start case. The cold-start case will be analyzed in Section 4.

Our benchmark machine is an Intel Xeon (Ice Lake) Platinum 8369B with 8 vCPUs and 16 GiB memory[3]. We use GCC 9.2.0 to compile all programs. We did not disable Turbo Boost option for the following reasons: Firstly, and most importantly, we aimed to simulate real-world application scenarios as closely as possible rather than conducting tests under ideal conditions. Secondly, extensive testing revealed that our experimental results remain reproducible even without disabling Turbo Boost, with no significant jitter observed.

The implementations of OpenSSL [40,41], Hisil et al. [25] and Cheng et al. [15], as well as this work are all compiled with the "-O2" flag[4], while the implementations of Faz-Hernández [18] and Nath et al. [35] are compiled with the "-O3" flag. Our experiment shows that the performance difference between "-O2" and "-O3" is negligible.

All performance reported in this section is warm-start performance; we run the subroutine 2,000 times before counting the CPU cycles to avoid the cold-start issue. The reported CPU cycles represent the average time taken to run the subroutine 20,000 times.

Table 2 compares our X25519 implementation with others. Even though OpenSSL's implementation is the slowest one in the table, a comparison with it is still needed. Since none of the previous work has integrated their implementations into OpenSSL, we need to use the OpenSSL implementation as a baseline when comparing metrics related to the TLS handshake. The ratio of CPU cycles of OpenSSL's X25519-KeyGen to our implementation is as high as 12.01; in other words, the throughput (executions per second) of our implementation is 12.01 times that of OpenSSL X25519-KeyGen. The proposed X25519-KeyGen achieves such a significant performance improvement because we pass the 8-way parallelisms of the AVX-512IFMA to the top layer thoroughly according to our MTP principles.

Table 3 provides a comparison of various Ed25519 implementations. Ed25519-Sign and Ed25519-Verify achieve 3.79 and 3.33 times higher throughput than OpenSSL, respectively. We also give the corresponding fixed-point scalar multiplication comparison in Table 4, showing that the throughput of our optimized fixed-point multiplication is 8.06 times and 1.67

[3] Our machine is leased from Alibaba Cloud. The family and instance type are compute optimized type c7 and ecs.c7.2xlarge, respectively. The 8 vCPUs can be approximately understood as 4 physical cores & 8 hyperthreads.

[4] For the implementation of OpenSSL, we need to extract the X25519 and Ed25519 implementation from the huge OpenSSL project to form a new small project for benchmark, so its compilation options are consistent with our implementation. For the implementation of Hisil et al., their compiling script did not work on our machine, so we refactored their project, and its compile options were also consistent with our implementation. Cheng et al. originally used the "-O2" option; they used the Clang compiler by default, and we changed it to GCC for a fair comparison.

Table 3: The CPU cycles comparison of different Ed25519 implementations under warm-start conditions. The comparison of Ed25519-KeyGen is not provided, partly because it is not used in TLS handshake and its performance is close to X25519-KeyGen.

Reference	Ed25519-Sign			Ed25519-Verify			ISA/ISE
	Cycles	Ratio	Strategy	Cycles	Ratio	Strategy	
OpenSSL [40]	93,606	3.79	$1\times1 \rightarrow 1\times1 \rightarrow 1\times1$	271,445	3.33	$1\times1 \rightarrow 1\times1 \rightarrow 1\times1$	x86-64
Faz-H. [18]	29,252	1.18	$4\times1 \rightarrow 4\times1 \rightarrow 1\times4$	108,082	1.33	$4\times1 \rightarrow 1\times4 \rightarrow 1\times1$	AVX2
This work	24,723	1.00	$8\times1 \rightarrow 8\times1 \rightarrow 1\times8$	81,506	1.00	$8\times1 \rightarrow 2\times4 \rightarrow 1\times2$	AVX-512IFMA

Table 4: The CPU cycles comparison of the corresponding fixed-point scalar multiplication.

Reference	Cycles	Ratio
OpenSSL [40]	71,561	8.06
Faz-H. [18]	14,813	1.67
This work	8,880	1.00

times that of OpenSSL and Faz-Hernández et al., respectively.

4 Reaching TLS and Application layers

The goal of reaching TLS and the application layers is to explore the performance of cryptographic operations in realistic scenarios, including the impact of the cold-start issue of vector units. In Section 4.1, we implement ENG25519 to transparently integrate our optimized implementation into TLS applications. Section 4.2 explores the cold-start issue and presents our solution for alleviating this issue. We conduct several experiments in Section 4.3 and Section 4.4 to benchmark TLS handshakes and DoT queries, respectively, through the successful integration of ENG25519.

4.1 The ENG25519 ENGINE

We follow the methodology suggested in [49] for incorporating the optimized X25519 and Ed25519 implementation into the OpenSSL ENGINE framework. Interested readers can refer to this paper for more detailed descriptions.

In the OpenSSL framework, an ENGINE acts as a container for implementing cryptographic algorithms. OpenSSL has a built-in ENGINE called "dynamic" that loads other ENGINEs, such as our ENG25519, in the form of a shared library, dynamically at runtime. Thus, through the OpenSSL configuration file, the "dynamic" ENGINE loads ENG25519, and the optimized implementations of X25519 and Ed25519 are transparently integrated into both OpenSSL and TLS applications.

The ENG25519 is built based on `libsuola` [50] in [49] and `engntru`[5] [43] in [5]. However, while `libsuola` provided

[5] An OpenSSL ENGINE that can integrate the batched implementation of a post-quantum key exchange scheme, NTRU Prime, into TLS and TLS applications.

Table 5: Detailed configuration of ENG25519.

Subroutine	Implementation
X25519-KeyGen Ed25519-KeyGen	Our $8\times1 \rightarrow 8\times1 \rightarrow 8\times1$ impl. *batch-size* $= 16$ (Section 3.4)
X25519-Derive	$4\times2 \rightarrow 1\times4$ impl. of Hisil et al. ([25])
Ed25519-Sign	Our $8\times1 \rightarrow 8\times1 \rightarrow 1\times8$ impl. (Section 3.4)
Ed25519-Verify	Our $8\times1 \rightarrow 2\times4 \rightarrow 1\times2$ impl. (Section 3.5)

simple test cases to verify its correctness, it did not reach the TLS ecosystem. When we attempted to reach the TLS layer, we encountered two bugs in `libsuola` (see Appendix C for the fixes). Consequently, our ENG25519 is a better ENGINE template to integrate ECC implementations into the TLS layer than `libsuola`, as ENG25519 has been verified to reach both the TLS and application layers.

The X25519 and Ed25519 are implemented through the EVP_PKEY_meth and the corresponding EVP_PKEY_ASN1_meth APIs within the OpenSSL framework, which are abstract data types provided by OpenSSL for public key cryptographic algorithms. The former API provides `keygen()` and `derive()` methods for X25519 implementation; provides `keygen()`, `sign()`, and `verify()` for Ed25519 implementation. The latter one provides encoding and decoding methods for the public and secret keys. Table 5 presents the detailed configuration of ENG25519.

To integrate our 8×1-way X25519-KeyGen into ENG25519, we implement a BATCH_STORE structure, as described in [5]. This structure holds a certain number of keypairs, i.e., 128 in our case. The `keygen()` method of X25519 accesses this structure to determine if there are any remaining keypairs. If so, it fetches them directly. Otherwise, it uses our 8×1-way X25519-KeyGen 16 times to regenerate the keypairs and repopulate the structure. We will discuss the design

Table 6: Warm-start performance vs cold-start performance in the DoT query scenario. CC is the abbreviation of CPU cycles.

Subroutine	Warm-start CC	Cold-start CC (DoT)	Cold-start CC/ warm-start CC	DoT warm CC[3]	DoT warm CC/ warm-start CC
Our X25519-KeyGen[1]	7,560[2]	28,450	3.8	10,315[4]	1.4
Our Ed25519-Sign	24,723	52,157	2.1	28,515	1.2
Our Ed25519-Verify	81,506	239,176	2.9	90,956	1.1
Hisil et al. X25519-Derive [25]	61,867	97,789	1.6	64,395	1.0

[1] Here, we set the batch size to 1 to see the effect of the auxiliary thread more intuitively.
[2] In Table 2, our 8×1-way X25519-KeyGen takes 6,0479 cycles for generating eight keypairs. Here, we report the average cost for generating one keypair.
[3] The performance using our auxiliary warm-up thread for mitigating the cold-start issue.
[4] Part of the overhead is consumed by the `BATCH_STORE` frame.

decisions for *batch-size* $= 16$ later. The `derive()` method is a wrapper around the 1×4-way X25519-Derive implementation by Hisil et al. [25]. The `keygen()` method for Ed25519 is similar to the X25519 case. The `sign()` and `verify()` methods are wrappers of our 1×8-way Ed25519-Sign and 1×2-way Ed25519-Verify implementation, respectively.

4.2 How to mitigate the cold-start issue?

The so-called "cold-start" approach measures the direct running time of the cryptographic algorithm a few thousand times during cryptographic benchmark tests. In contrast, the "warm-start" approach involves an initial warm-up phase where the cryptographic algorithm is executed a few hundred times before measurement. According to [19, Sec 11.9], 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.

After a careful review of the previous implementations with AVX2/AVX-512, we found that the cryptographic implementers tend to report only warm-start performance (e.g. [12, 13, 15, 18, 25, 35] and their software [10, 11, 14, 17, 24, 34]), without considering the adverse effects of the cold-start issue of vector units. Actually, the cold-start issue has never been discussed in previous related work in the cryptographic engineering field. This finding motivates us to explore the actual effect the cold-start issue brings to the ECC implementation with AVX2/AVX-512. Furthermore, as our objective is to incorporate the AVX-512IFMA-optimized ECC implementation into TLS, the vector units could operate in the low-power mode during certain stages of the TLS protocol, which could result in a much worse cold-start performance than what has been observed during warm-start testing.

As a result, we intend to systematically examine and alleviate adverse effects of the cold-start issue on real-world TLS applications. We measured the CPU cycles of cryptographic operations under the DoT query benchmark conditions (cf. Section 4.4). These results are more qualified than the warm-start performance reported in Table 2 and 3 to illustrate the performance of cryptographic operations in realistic TLS applications.

We present the results in Table 6. The "warm-start CC" column is derived from Table 2 and 3. The "Cold-start CC (DoT)" column is measured under DoT query benchmark conditions, and the results show that cryptographic operations are cold-started. The "Cold-start CC/warm-start CC" column indicates the performance degradation of cryptographic operations in the DoT query conditions (i.e. cold-start scenario) compared to the warm-start scenario. The last two columns will be analyzed later. All subroutines 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.

Initially, we hypothesized that the performance degradation was caused by cache misses due to the larger executable file than the L1 cache (as mentioned in [55, Sec 5.1]). However, after careful analysis through the top-down method [54] with `pmu-tools` [39], we concluded that cache misses could not cause such significant performance degradation. Subsequently, we discovered the cold-start issue of vector units, which is also mentioned in [19, 20, 52]. As this is a mechanism inside the CPU without any interface to disable it, we have to find a workaround to mitigate this issue.

The warm-up phase typically lasts around 14 µs (56,000 cycles under 4GHz), after which all vector units are fully activated, as demonstrated in [19, Sec 11.9]. The vector units will enter low-power mode if no vector instructions are executed for approximately 675 µs. To tackle this issue, one can execute a vector instruction 14 µs prior to cryptographic operations to initiate the warm-up phase. However, predicting precisely when a client initiates a TLS handshake with a server is impossible.

To address this problem, we propose an auxiliary warm-up thread that executes a dummy vector instruction every 500 µs to prevent the vector units from entering low-power mode. The time interval is set as 500 µs to allow for timer errors. Nonetheless, continuously preventing the vector units from entering low-energy mode will increase the CPU's power consumption, which contradicts the CPU designer's intentions. Therefore, we propose a heuristic warm-up scheme for the auxiliary thread, which operates as follows: It wakes up every 500 µs, executes a vector instruction if necessary, and then goes to sleep. Otherwise, it goes to sleep directly, resulting in minimal consumption of CPU resources because it sleeps most of the time.

Heuristic warm-up scheme The `unbound` server provides the capability to report statistics [45], allowing us to determine the number of DoT queries within a 60-second period[6], denoted as Q. We record the maximum number of DoT queries that `unbound` can resolve in 60 seconds as Q_{max}, which is approximately 290,000 with ENG25519 support. We classify DoT workload into three categories: high, medium, and low. In high workload scenarios ($Q \geq 120,000$), at least one DoT query is resolved on average every 500 µs, and the auxiliary thread does not require any warm-up actions. For medium workload scenarios ($60,000 \leq Q < 120,000$), the auxiliary thread executes one vector instruction every 500 µs to prevent the vector unit from entering low-power mode. For low workload scenarios ($Q < 60,000$), the auxiliary thread does not need to execute any warm-up actions. Compared to consistently executing a vector instruction every 500 µs to warm up vector units, regardless of the workload scenario, our heuristic warm-up scheme considers the various scenarios' requirements. Unnecessary warm-up actions are removed for high-workload scenarios, periodic warm-up actions ensure faster TLS handshakes for medium-workload scenarios, and for low-workload scenarios, we avoid disrupting the power-saving mechanism inside the CPU. We describe the heuristic warm-up scheme as follows.

$$T = \begin{cases} +\infty, & Q \geq 120,000 \\ 500\,\mu s, & 60,000 \leq Q < 120,000 \\ +\infty, & Q < 60,000 \end{cases} \quad (9)$$

where the time interval T denotes the duration for the auxiliary thread to execute a vector instruction for warm-up vector units, and the symbol $+\infty$ indicates no instruction execution. In our implementation, the thresholds for dividing the three workload cases and the corresponding time intervals are adjustable, enabling the scheme to be expanded to other application scenarios. See Appendix D for the guidelines of parameters adjustment.

The last two columns in Table 6 show the effectiveness of our auxiliary thread. These results are measured after equipping our auxiliary thread with $T = 500\,\mu s$ for both the server and the client. In realistic scenarios, we only equip the server with the auxiliary thread. The client is only equipped with the auxiliary thread in the experiments of Table 6; in all other experiments, the client is not equipped with the auxiliary thread. In practice, requests initiated by clients are scattered and unpredictable. We forgo the performance sweetspot brought by auxiliary threads on the client side to simulate a more realistic scenario. In this way, among the cryptographic operations performed by the client, the batching technique described below can mitigate the cold-start issue of X25519-KeyGen. However, X25519-Derive and Ed25519-Verify will inevitably suffer from the cold-start issue. Benchmarks in Sections 4.3 and 4.4 demonstrate that our configuration (i.e. the server using ENG25519 with our auxiliary thread and the client using ENG25519 without our auxiliary thread) still outperforms other configurations.

[6]The time interval is adjustable in the unbound configuration file.

Table 7: Amortized CPU cycles (CC) to generate a keypair using our 8×1-way X25519-KeyGen with different batch sizes. The data presented here encompasses the memory management overhead introduced by the OpenSSL ENGINE layer.

Batch size	Amortized CC with auxiliary thread	Amortized CC without auxiliary thread
1	10,315	28,450
2	9,903	24,977
4	9,107	19,388
8	9,003	14,108
16	8,980	11,406

Power consumption One might think that using AVX-512 and bypassing energy-saving mechanisms could potentially increase CPU power consumption and even carbon emissions. We argue that our faster cryptographic computations not only don't increase power consumption but actually reduce it due to the overall time reduction. Consider the cryptographic operations in a TLS 1.3 handshake on the server side, i.e. X25519-KeyGen+X25519-Derive+Ed25519-Sign. This work (97.1k CC) achieves an 184% and 31% improvement over OpenSSL (275.3k CC) and Faz-Hernandez (127.1k CC), respectively. Professional power consumption tests [31] indicate that AVX-512 only increases average power consumption by 17% and peak power consumption by 10% compared to AVX2. Additionally, compared to the implementation without any AVX instructions, AVX-512 increases the peak power consumption by 9.6%, but reduces the average power consumption. Considering the efficiency improvement, utilizing AVX-512 remains beneficial in terms of power consumption.

Online-KeyGen and offline-KeyGen We only consider the online KeyGen scenario. The offline-KeyGen pattern is commonly used in scenarios such as TLS/SSL certificate au-

thorities and cryptocurrencies with dedicated security devices for protecting offline keypairs. Our focus is on the online-KeyGen (for X25519 key exchange) scenario. Most server software using the TLS protocol adopts the online-KeyGen mechanism, such as `unbound` and `nginx`. The precomputed keypairs are securely stored in memory, eliminating the need for dedicated security devices. Key management involves simple memory operations at the ENGINE level, which is transparent to the server.

Batch X25519-KeyGen As previously mentioned, X25519-KeyGen can be performed without relying on information from the communication peer. This is the underlying premise that allows our 8×1-way X25519-KeyGen to be utilized. Calling our 8×1-way X25519-KeyGen once generates eight different keypairs. We call it 16 times when needed, denoted as *batch-size* $= 16$, for two reasons. First, when auxiliary thread support is not available, such as on the client side mentioned above, multiple calls can amortize the impact of the cold-start issue. Second, when auxiliary thread support is present, as on the server side, multiple calls are more cache-friendly and can improve performance. Table 7 presents the amortized cost of each keypair under different batch sizes. Compared to *batch-size* $= 1$, the performance improvement of *batch-size* $= 16$ is 13% or 60% with or without our auxiliary thread, respectively. Further increasing *batch-size* beyond 16 yields diminishing returns, so we adopt 16 as our choice. The time taken to generate a batch is approximately 0.22 to 0.33 milliseconds (under 4GHz), which is imperceptible to users.

4.3 Benchmark of TLS handshake

We used two machines with the same configuration as the client and server to benchmark TLS handshake, the same as in Section 3.6. Additionally, the two machines communicate over an internal network with a bandwidth of up to 10Gbps. Our OpenSSL version is `1.1.1q`. Our benchmark method is similar to Bernstein et al. [5, Sec 4.4] in Usenix Security 2022. We ran a generic TLS server on the server side that listens for connections on a given port using TLS via the `openssl s_server` tool [42]. On the client side, we measured handshakes per second using `tls_timer` [44]. In the main loop of `tls_timer`, it records a timestamp, and then performs a certain number of TLS connections, then records another timestamp, resulting in the elapsed time. For each TLS connection, it just performs TLS 1.3 handshake without sending any application data, and then the client properly shuts down the connection. The elapsed time covers the computational overhead of cryptographic operations, the time for the packet to travel over the network, and the time to travel from kernel space to user space.

Our experiments with cumulative distributions are presented in Figure 3a. We used Ed25519 as the signature algorithm for all experiments. As a baseline for comparison, we report results with both client and server using NIST P-256 [36] and X25519 in OpenSSL `1.1.1q` for key exchange, namely "P256" and "X25519" in legend. Both experiments use the cryptographic implementation built into OpenSSL instead of using any ENGINEs. In the setting of the "ENG25519" legend, the server adopts our ENG25519 supported by our auxiliary warm-up thread, and the client adopts our ENG25519 not supported by the auxiliary thread. The "ALL-OpenSSL" legend indicates that the ENGINE is implemented using X25519 and Ed25519 implementation in OpenSSL. When it is compared with the "X25519" configuration, the impact of the ENGINE framework on performance can be known.

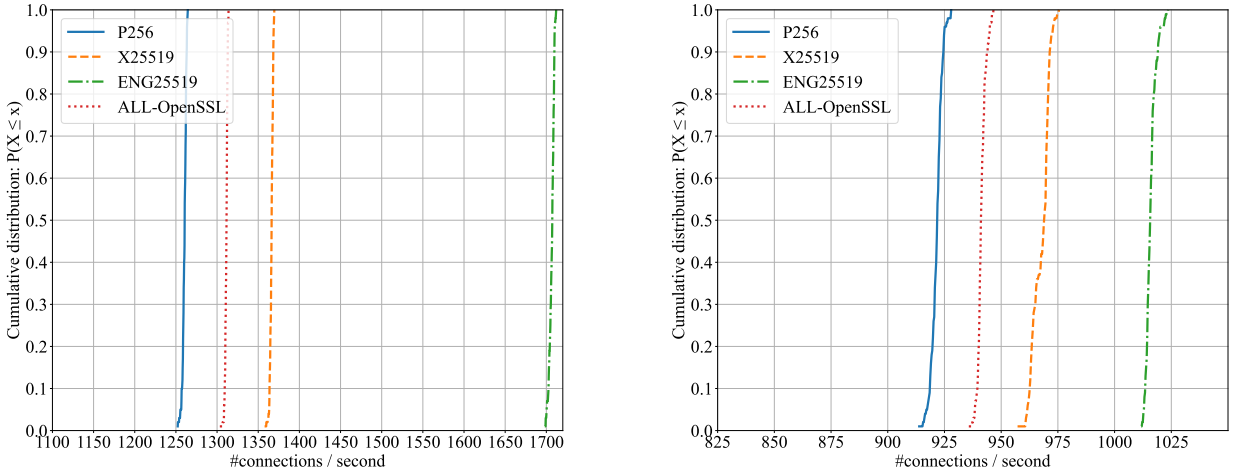
On average, the proposed ENG25519 setting (1,707 #connections/second) enables 25% and 35% more handshakes per second than X25519 (1,366) and P256 (1,260), respectively.

4.4 Benchmark of DoT query

We conducted two benchmarking approaches. The first involved end-to-end experiments derived from [5, Sec 4.4] in Usenix Security 2022, allowing us to showcase optimizations on both client and server paths. This approach is in align with users' perception. The second measured the server's peak throughput by bombarding it with a sufficient number of clients, quantified as the number of completed DoT queries per 60 seconds (#queries/min). Large-scale service providers are particularly interested in this metric, as higher throughput can potentially reduce hardware acquisition costs.

End-to-end experiments On the server side, we utilized `unbound` [37] DoT server, which listens for TLS connections on a given port. On the client side, we developed the `dot_timer` tool based on the `tls_timer`. Once the TLS 1.3 handshake is completed, `dot_timer` sends a DNS query to `unbound`, waits for the corresponding DNS response over the established TLS connection, and then properly shuts down the connection. Other configurations remain unchanged from Section 4.3. Compared to `tls_timer`, the elapsed time recorded by `dot_timer` additionally covers the time taken by the client to send the DNS query and receive the DNS response over the TLS connection. Figure 3b visualizes our experimental results, with the same legends as in Section 4.3. 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. This enhancement demonstrates the effectiveness of our optimizations in achieving higher server throughput.



(a) TLS 1.3 handshake performance

(b) DoT query performance

Figure 3: Cumulative distributions of TLS handshake and DoT query performance under different configurations. For each configuration, we collected 100 samples in terms of the average number of TLS connections and DoT queries per second. For each sample, we measure the elapsed time over 10000 sequentially TLS 1.3 handshakes and DoT queries.

5 Security

We confirm that all instructions used in this work are constant-time, and we avoid secret-data-dependent memory access and branches to protect against cache-timing attacks and branch prediction attacks. The implementation of X25519-KeyGen and Ed25519-Sign involves accessing precomputed tables. Our secure table lookup method traverses the entire (sub)table and uses masking techniques to obtain the desired table entry. Attackers cannot observe any useful cache access patterns or expose any secret information. The precomputed keypairs are saved directly into memory. Memory security is ensured through operating system (OS) measures, e.g., virtual memory, access control, and memory isolation. Our X25519 keypairs follow the "one-time pad" pattern, with no secret information being revealed to a specific cache line. Different private keys are generated using random number generation APIs, ensuring no correlations between different keys. Thus, our implementation does not introduce any potential attack surface and does not affect the security assumptions relied upon by cryptographic primitives.

6 Availability and Scalability

The availability of AVX-512IFMA AVX-512IFMA was first implemented in Cannon Lake CPUs in 2018. Since then, it has gained widespread support in high-performance server environments. After 2018, most Intel Xeon series processors, except Cascade Lake (2019) and Cooper Lake (2020), have provided support for AVX-512IFMA. On the client side, AVX-512 support has been somewhat contentious due to power consumption concerns. However, it is important to note that our optimizations are of greater interest to large-scale service providers.

For hardware without AVX-512IFMA support AVX2 enjoys more widespread support compared to AVX-512, with almost all CPUs released after Haswell (2013) supporting AVX2. For hardware that only supports AVX2, one can derive a solution with additional engineering efforts: At the cryptographic primitive layer, employ $4 \times$ 1-way X25519-KeyGen from [15], X25519-Derive from [25], Ed25519-Sign, and Ed25519-Verify from [18]. In the OpenSSL ENGINE layer, our ENG25519 framework (including batching mechanisms for X25519-KeyGen) can be reused with some engineering effort to align with the interfaces of cryptographic primitives. Our auxiliary threads with the heuristic wake-up strategy can also be used to mitigate the cold start issue associated with AVX2.

Scalability of the heuristic warm-up scheme Our implementation of the heuristic warm-up scheme depends on the reporting statistics feature of `unbound`. However, for TLS applications without this feature, the scheme is still available by implementing it at the ENGINE layer.

Scalability of MTP principle for high-end hardware For wider SIMD/vector instruction sets, such as the vector length

supported by the ARM SVE(2) is up to 2,048 bits [48], and supported by the vector extension of RISC-V is up to 16,384 bits [22]. The design concepts proposed in this paper to enhance the parallelism of ECC-related arithmetic from 4-way to 8-way can serve as a valuable guide when contemplating hardware with greater parallelism. For instance, guided by the MTP principle, for a machine that supports 1,024-bit vectors, according to the ideas in this work, the optimization scheme is $16\times1 \to 16\times1 \to 16\times1$ X25519-KeyGen, $4\times4 \to 1\times4$ X25519-Derive, $16\times1 \to 16\times1 \to 1\times16$ Ed25519-Sign, and $16\times1 \to 2\times8 \to 1\times2$ Ed25519-Verify. Then, utilize the methodology similar to this paper to reach the TLS layer and TLS applications.

Application scenarios In addition to the example of DoT provided in this paper, we list here more potential application scenarios that could benefit from this paper: (1) E-commerce, especially during promotional discount periods. During events like Black Friday sales, website traffic can surge to levels 30 times higher than usual [51]. A report by the WTO [53, Sec 7] also mentions that with the rise of e-commerce, network capacity and higher bandwidth services have proved to be crucial. (2) Social media platforms that experience sudden spikes in activity due to major news events can also be vulnerable to performance issues related to the TLS handshake and could experience crashes [1, 29]. (3) Industrial Internet of Things (IoT) servers. It's predicted that by 2025, the number of connected devices per minute will reach 152,000 [38]. For instance, there can be thousands of devices connected to the server at any given time, each requiring a secure TLS handshake. This underscores the critical importance of improving server throughput. The common feature of these scenarios is their strong emphasis on peak throughput because it can even affect hardware resource procurement costs.

Session resumption mechanism This mechanism can effectively expedite TLS handshakes using pre-shared keys. However, to ensure forward secrecy when conducting fast handshakes with this mechanism, ECDHE calculations (i.e., X25519-KeyGen and X25519-Derive in our case) are still required. As mentioned in TLS 1.3 specification [46, Sec 2.2]: "When a client offers resumption via a PSK, it SHOULD also supply a "key_share" extension to the server to allow the server to decline resumption and fall back to a full handshake if needed. The server responds with a "pre_shared_key" extension to negotiate the use of PSK key establishment and can (as shown here) respond with a "key_share" extension to do (EC)DHE key establishment, thus providing forward secrecy." Therefore, this work can also accelerate scenarios where session resumption is used while ensuring forward secrecy.

7 Conclusions

In this study, we redesign and implement all layers of ECC arithmetic to improve the performance of X25519 and Ed25519 using the AVX-512IFMA instruction set. At the OpenSSL level, we proposed ENG25519, based on the OpenSSL ENGINE API and `libsuola`, which can actually benefit the TLS application from the optimized ECC implementations. In the end, we choose TLS handshakes and DoT for application-level benchmarking. During this process, we also uncover and address the cold-start issue of vector units, which, to our knowledge, had not been reported in the field of cryptographic engineering before.

Our solution achieves a speedup of 25% to 35% for TLS handshakes per second and improves peak server throughput for DoT queries by 24% to 41%. This means that our solution is indeed effective in mitigating the computational burden of TLS handshakes for throughput-critical scenarios. Additionally, we discuss the scalability of our solution, which can extend to high-end hardware, low-end hardware, and various CPU architectures. This suggests that our solution offers guidance for complex real-world environments.

Acknowledgments

This work is supported by the Jiangsu Province 100 Foreign Experts Introduction Plan (BX2022012), TÜBİTAK Projects (2232-118C332 and 1001-121F348), Guangdong Provincial Key Laboratory IRADS (2022B1212010006, R0400001-22), and Guangdong Province General Universities Key Special Fund (New Generation Information Technology) (2023ZDZX1033).

We would like to express our gratitude to Hao Cheng, Bo Zhao, Nicola Tuveri, Matt Caswell, Wouter Wijngaards, Hao Wang, and Xuan Yu for their valuable discussions. We also extend our appreciation to the anonymous reviewers for their thoughtful discussions and constructive feedback. We also express our gratitude for the partial financial support provided by Zhe Liu for leasing cloud servers.

References

[1] Levi Adkins. Just yesterday, the Weibo server blew up! Twitter programmers need a break. https://www.mo4tech.com/just-yesterday-the-weibo-server-blew-up-twitter-programmers-need-a-break.html, 2023. Accessed: 2023-10-17.

[2] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography - PKC 2006*, volume 3958, pages 207–228. Springer, 2006.

[3] Daniel J. Bernstein. 25519 naming. https://mailarchive.ietf.org/arch/msg/cfrg/

-9LEdnzVrE5RORux3Oo_oDDRksU/, 2014. Accessed: 2023-07-01.

[4] Daniel J Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted Edwards Curves. In *International Conference on Cryptology in Africa*, pages 389–405. Springer, 2008.

[5] Daniel J Bernstein, Billy Bob Brumley, Ming-Shing Chen, and Nicola Tuveri. OpenSSLNTRU: Faster post-quantum TLS key exchange. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 845–862, 2022.

[6] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[7] Daniel J. Bernstein and Peter Schwabe. NEON Crypto. In *Cryptographic Hardware and Embedded Systems - CHES 2012*, volume 7428, pages 320–339. Springer, 2012.

[8] Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug attacks. In *Advances in Cryptology - CRYPTO 2008*, volume 5157, pages 221–240. Springer, 2008.

[9] Billy Bob Brumley, Sohaib ul Hassan, Alex Shaindlin, Nicola Tuveri, and Kide Vuojärvi. Batch Binary Weierstrass. In *Progress in Cryptology - LATINCRYPT 2019*, volume 11774, pages 364–384. Springer, 2019.

[10] Hao Cheng. Artifact of the paper "Batching CSIDH group actions using AVX-512". https://gitlab.uni.lu/APSIA/AVX-CSIDH, 2021. Accessed: 2023-07-01.

[11] Hao Cheng. Artifact of the paper "Highly vectorized SIKE for AVX-512". https://gitlab.uni.lu/APSIA/AVXSIKE, 2022. Accessed: 2023-07-01.

[12] Hao Cheng, Georgios Fotiadis, Johann Groszschädl, and Peter YA Ryan. Highly vectorized SIKE for AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2022(2):41–68, 2022.

[13] Hao Cheng, Georgios Fotiadis, Johann Groszschädl, Peter YA Ryan, and Peter Roenne. Batching CSIDH group actions using AVX-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(4):618–649, 2021.

[14] Hao Cheng, Johann Großschädl, and Jiaqi Tian. Artifact of the paper "High-Throughput Elliptic Curve Cryptography Using AVX2 Vector Instructions". https://gitlab.uni.lu/APSIA/AVXECC, 2021. Accessed: 2023-07-01.

[15] Hao Cheng, Johann Großschädl, Jiaqi Tian, Peter B. Rønne, and Peter Y. A. Ryan. High-Throughput Elliptic Curve Cryptography Using AVX2 Vector Instructions. In *Selected Areas in Cryptography - SAC 2020*, volume 12804, pages 698–719. Springer, 2020.

[16] Alexander Dupuy. Google Public DNS turns 8.8.8.8 years old. https://security.googleblog.com/2018/08/google-public-dns-turns-8888-years-old.html, 2018. Accessed: 2023-10-17.

[17] Armando Faz-Hernández. Artifact of the paper "High-performance Implementation of Elliptic Curve Cryptography Using Vector Instructions". https://github.com/armfazh/fld-ecc-vec, 2019. Accessed: 2023-07-01.

[18] Armando Faz-Hernández, Julio César López-Hernández, and Ricardo Dahab. High-performance Implementation of Elliptic Curve Cryptography Using Vector Instructions. *ACM Transactions on Mathematical Software*, 45(3):25:1–25:35, 2019.

[19] Agner Fog. The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers. https://www.agner.org/optimize/microarchitecture.pdf, 2022. Accessed: 2023-07-01.

[20] Agner Fog. Test results for Broadwell and Skylake. https://www.agner.org/optimize/blog/read.php?i=415, December 26, 2015. Accessed: 2023-07-01.

[21] Yu-Fu Fu, Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Signed cryptographic program verification with typed cryptoline. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1591–1606. ACM, 2019.

[22] Tom R. Halfhill. RISC-V Vectors Know No Limits. https://www.linleygroup.com/newsletters/newsletter_detail.php?num=6154, 2020. Accessed: 2023-07-01.

[23] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[24] Hüseyin Hisil, Berkan Egrice, and Mert Yassi. Artifact of the paper "Fast 4 way vectorized ladder for the complete set of Montgomery curves". https://github.com/crypto-ninjaturtles/montgomery4x, 2020. Accessed: 2023-07-01.

[25] Hüseyin Hisil, Berkan Egrice, and Mert Yassi. Fast 4 way vectorized ladder for the complete set of Montgomery curves. *IACR Cryptology ePrint Archive*, page 388, 2020.

[26] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards Curves Revisited. In *Advances in Cryptology - ASIACRYPT 2008*, volume 5350, pages 326–343. Springer, 2008.

[27] Zi Hu, Liang Zhu, John Heidemann, Allison Mankin, Duane Wessels, and Paul E. Hoffman. RFC 7858: Specification for DNS over Transport Layer Security (TLS). https://www.rfc-editor.org/info/rfc7858, May 2016.

[28] Simon Josefsson and Ilari Liusvaara. RFC 8032: Edwards-Curve Digital Signature Algorithm (EdDSA). https://www.rfc-editor.org/info/rfc8032, January 2017.

[29] Raza Aslam Lakhani. Why Do Websites Crash, and How to Rescue Your Clients From a Website Crash. https://www.cloudways.com/blog/why-do-websites-crash/, 2022. Accessed: 2023-10-17.

[30] Adam Langley, Mike Hamburg, and Sean Turner. RFC 7748: Elliptic Curves for Security. https://www.rfc-editor.org/info/rfc7748, January 2016.

[31] Michael Larabel. AVX / AVX2 / AVX-512 Performance + Power On Intel Rocket Lake. https://www.phoronix.com/review/rocket-lake-avx512/6, 2021. Accessed: 2023-07-01.

[32] Jiaxiang Liu, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying arithmetic in cryptographic C programs. In *34th IEEE/ACM International Conference on Automated Software Engineering*, pages 552–564. IEEE, 2019.

[33] Peter L Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of computation*, 48(177):243–264, 1987.

[34] Kaushik Nath. Artifact of the paper "Efficient 4-Way Vectorizations of the Montgomery Ladder". https://github.com/kn-cs/vec-ladder, 2022. Accessed: 2023-07-01.

[35] Kaushik Nath and Palash Sarkar. Efficient 4-Way Vectorizations of the Montgomery Ladder. *IEEE Transactions on Computers*, 71(3):712–723, 2022.

[36] NIST. Digital Signature Standard (DSS), FIPS 186-4. https://csrc.nist.gov/publications/detail/fips/186/4/final, 2013. Accessed: 2023-07-01.

[37] NLnetLabs. Unbound, a validating, recursive, and caching DNS resolver. https://nlnetlabs.nl/projects/unbound/about/, 2022. Accessed: 2023-07-01.

[38] Christo Petrov and Lorie Tonogbanua. 26 Insightful Internet of Things Statistics 2023. https://techjury.net/blog/internet-of-things-statistics/, 2023. Accessed: 2023-10-17.

[39] Intel-PMU Project. pmu-tools. https://github.com/andikleen/pmu-tools, 2023. Accessed: 2023-07-01.

[40] OpenSSL Project. Ed25519 C language implementation. https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/crypto/ec/curve25519.c, 2016. Accessed: 2023-07-01.

[41] OpenSSL Project. X25519 x64 assembly language implementation. https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/crypto/ec/asm/x25519-x86_64.pl, 2018. Accessed: 2023-07-01.

[42] OpenSSL Project. OpenSSL s_server tool. https://beta.openssl.org/docs/manmaster/man1/openssl-s_server.html, 2022. Accessed: 2023-07-01.

[43] OpenSSLNTRU Project. Engntru OpenSSL Engine. https://opensslntru.cr.yp.to/engntru-20210608.tar.gz, 2021. Accessed: 2023-07-01.

[44] OpenSSLNTRU Project. tls_timer tool. https://opensslntru.cr.yp.to/tls_timer-20210608.tar.gz, 2021. Accessed: 2023-07-01.

[45] Unbound Project. Howto Statistics. https://www.nlnetlabs.nl/documentation/unbound/howto-statistics/, 2022. Accessed: 2023-07-01.

[46] Eric Rescorla. RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3. https://www.rfc-editor.org/info/rfc8446, August 2018.

[47] Jerome A. Solinas. Efficient Arithmetic on Koblitz Curves. *Designs, Codes and Cryptography*, 19:195–249, 2000.

[48] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The ARM Scalable Vector Extension. *IEEE Micro*, 37(2):26–39, mar 2017.

[49] Nicola Tuveri and Billy Bob Brumley. Start Your ENGINEs: Dynamically Loadable Contemporary Crypto. In *2019 IEEE Cybersecurity Development, SecDev 2019*, pages 4–19. IEEE, 2019.

[50] Nicola Tuveri, Billy Bob Brumley, and Iaroslav Gridin. Libsuola OpenSSL Engine. https://github.com/romen/libsuola, 2019. Accessed: 2023-07-01.

[51] Mike Wager. High Traffic Surges Got Your Website Down? Here's Why. https://www.keysight.com/blogs/tech/software-testing/2022/11/15/high-traffic-surges-got-your-website-down, 2022. Accessed: 2023-10-17.

[52] WikiChip. Skylake (client) - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client), 2015. Accessed: 2023-07-01.

[53] WTO. E-commerce, trade and the COVID-19 pandemic. https://www.wto.org/english/tratop_e/covid19_e/ecommerce_report_e.pdf, 2020. Accessed: 2023-10-17.

[54] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.

[55] Jipeng Zhang, Junhao Huang, Zhe Liu, and Sujoy Sinha Roy. Time-memory Trade-offs for Saber+ on Memory-constrained RISC-V Platform. *IEEE Transactions on Computers*, 71(11):2996–3007, 2022.

A Details of AVX-512IFMA instructions

The 52-bit multiplication instructions One of AVX-512IFMA's critical features is the 52-bit integer multiplier, with corresponding instructions like `vpmadd52luq` and `vpmadd52huq`. For instance, the "`vpmadd52luq a_l, b, c`" and "`vpmadd52huq a_h, b, c`" instructions[7] can multiply the unsigned 52-bit integers in each 64-bit lane of `b` and `c` and produce a 104-bit intermediate result utilizing the 52-bit multiplier. The first instruction gets the low 52-bit result and accumulates it to the corresponding lane of `a_l`. The second instruction gets the high 52-bit result and accumulates it to `a_h`.

Mask addition and subtraction instructions Consider the inline[8] mask addition instruction `_mm512_mask_add_epi64`, abbreviated as VMADD. The syntax of the instruction is as follows:

```
__m512i _mm512_mask_add_epi64(__m512i src,
        __mmask8 k, __m512i a, __m512i b);
```

The instruction adds packed eight 64-bit integers in `a` and `b`, stores the results in `dst` (the return value), and employs an 8-bit mask `k` to determine whether each corresponding lane of `dst` should be a copy of the corresponding lane of `src` or the result of the addition. Specifically, when the mask bit in `k` is 0, the corresponding lane in `dst` is a copy of the corresponding lane in `src`, and when the mask bit is 1, the corresponding lane in `dst` is the sum of the corresponding lanes in `a` and `b`. The mask subtraction `_mm512_mask_sub_epi64`, abbreviated as VMSUB, operates in a similar manner.

Suppose $a = [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$ and $b = [b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7]$. In line 2 of Algorithm 1, the mask k is 0x11, which is represented in binary as 00010001. The result of the MADD$(a, 0\text{x}11, a, b)$ instruction is $[a_0 + b_0, a_1, a_2, a_3, a_4 + b_4, a_5, a_6, a_7]$. This means that the 0-th and 4-th lanes are the sum of the corresponding lanes in a and b, while the remaining lanes are a copy of the corresponding lanes in a.

Permutation instructions Apart from the mask instructions, the permutation instruction, abbreviated as PERM, is also versatile. With the following syntax, it shuffles the 64-bit integers in `a` within each 256-bit lane using the control bits in `imm8` and stores the results in `dst`.

```
__m512i _mm512_permutex_epi64(__m512i a,
        const int imm8);
```

For example, in line 1 of Algorithm 1, where `imm8` is 0xB1, and its binary representation is 10110001, which is equivalent to "2 3 0 1" in two-bit integers. The result of PERM$(a, 0\text{x}B1)$ is $[a_1, a_0, a_3, a_2, a_5, a_4, a_7, a_6]$. Every two-bit integer in "2 3 0 1" is used to index a 64-bit integer in each 256-bit lane of a 512-bit register. The 8-bit values 0xDD, 0x88, and 0x4B in Algorithm 1 follows the same pattern as `imm8`.

B Performance penalty caused by permutation instructions

Using 4×2-way modular multiplication as an example, let $a, b, \cdots, g, h \in \mathbb{F}_p$ where $p = 2^{255} - 19$. The 4×2-way modular multiplication calculates four modular multiplications in parallel, i.e., $a \cdot e, b \cdot f, c \cdot g, d \cdot h \pmod{p}$. The four elements a, b, c, and d can be represented using three 512-bit registers with the radix-2^{51} representation as $[a_0, a_3, b_0, b_3, c_0, c_3, d_0, d_3]$, $[a_1, a_4, b_1, b_4, c_1, c_4, d_1, d_4]$, and $[a_2, 0, b_2, 0, c_2, 0, d_2, 0]$.

[7] Each of the AVX-512 instruction has a corresponding mask operand to control the specific behavior of the instruction, here we ignore it for the convenience of understanding.

[8] Intel Intrinsics Guide: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

The elements e, f, g, and h can also be represented in a similar way. We need to construct all the product terms, i.e. a_ie_j, b_if_j, c_ig_j, and d_ih_j where $i, j = 0, \cdots, 4$, using the following computation sequences:

$$
\begin{aligned}
&[a_0, a_0, b_0, b_0, c_0, c_0, d_0, d_0] \times [e_0, e_3, f_0, f_3, g_0, g_3, h_0, h_3] \\
&[a_0, a_0, b_0, b_0, c_0, c_0, d_0, d_0] \times [e_1, e_4, f_1, f_4, g_1, g_4, h_1, h_4] \\
&[a_0, a_0, b_0, b_0, c_0, c_0, d_0, d_0] \times [e_2,\ 0,\ f_2, 0,\ g_2,\ 0,\ h_2, 0] \\
&\cdots \\
&[a_4, a_4, b_4, b_4, c_4, c_4, d_4, d_4] \times [e_0, e_3, f_0, f_3, g_0, g_3, h_0, h_3] \\
&[a_4, a_4, b_4, b_4, c_4, c_4, d_4, d_4] \times [e_1, e_4, f_1, f_4, g_1, g_4, h_1, h_4] \\
&[a_4, a_4, b_4, b_4, c_4, c_4, d_4, d_4] \times [e_2,\ 0,\ f_2, 0,\ g_2,\ 0,\ h_2, 0].
\end{aligned}
$$

Obviously, we need to use permutation instructions to transform the form $[a_0, a_3, b_0, b_3, c_0, c_3, d_0, d_3]$ (i.e. input form) into the form $[a_0, a_0, b_0, b_0, c_0, c_0, d_0, d_0]$. In addition, permutation instructions are also required when accumulating these product terms. Overall, twenty permutation instructions are used (12 `vpshufd` instructions and 8 `vpblendmq` instructions) in our 4×2-way modular multiplication.

For the 8×1-way modular multiplication, the eight finite field elements can be represented with radix-2^{51} representation as:

$$
\begin{aligned}
&[a_0, b_0, c_0, d_0, e_0, f_0, g_0, h_0] \\
&\cdots \\
&[a_4, b_4, c_4, d_4, e_4, f_4, g_4, h_4].
\end{aligned}
$$

We can construct the execution flow without permutation instructions by adopting a similar computation sequence as 1-way implementation.

The performance penalty caused by awkward permutation instructions in the bottom-level finite field arithmetic will be magnified due to its heavy usage by the upper-level arithmetic (e.g. point arithmetic and scalar multiplication).

C Fixing two bugs in libsuola

The first bug prevents the TLS layer from finding a suitable signature algorithm for the TLS handshake when we want TLS to select Ed25519. During the execution flow of the TLS handshake, OpenSSL checks the digital signature algorithm and its corresponding message digest algorithm. If the check fails, OpenSSL refuses to use the corresponding algorithm, resulting in the failure of the TLS handshake. In the `libsuola` implementation, OpenSSL is told that the message digest scheme for Ed25519 is "mandatory"[9], which makes TLS refuse to use the Ed25519 algorithm, causing the TLS handshake to fail. To fix this issue, our ENG25519 tells OpenSSL that the message digest scheme for Ed25519 is "advisory"[10] instead of "mandatory".

[9] Return value is 2 in https://github.com/romen/libsuola/blob/c055fd0f546b8a257293e7794055886081deddfd/meths/suola_asn1_meth.c#L208-L212

[10] Return value is 1 instead of 2

The second bug resulted from an incorrect `EVP_MD` implementation for Ed25519, causing incorrect signature results. Before calling the `sign()` subroutine during the TLS handshake, OpenSSL copies a temporary `EVP_MD` and uses it to execute `sign()`. The message cached in the original `EVP_MD` is not cleared after the `sign()` subroutine is executed, which leads to residual contents being included in the signed messages. We fix the issue by clearing the message in the original `EVP_MD` after it is copied by OpenSSL.

After fixing these two bugs, our ENG25519 works successfully after integrating it into TLS applications.

D Parameter adjustment guidelines

The parameters used for categorizing three workload cases The configuration of these parameters determines how the workload is categorized into three different cases. We consider $Q \geq 120,000$ as high load because estimates suggest that under such load, there is an average of one DoT request every 500 microseconds, simulating a wake-up operation. Therefore, there's no need for an additional wake-up. The division between medium and low loads is primarily based on empiricism. For medium loads, our default configuration enable wake-up operations to expedite server-side TLS handshake computations, proactively preparing for potential transitions to high load conditions due to sudden user surges. Experienced developers can fine-tune the categorization of the three workload cases based on their own empirical insights. For example, if one aims to accelerate TLS handshake calculations even under low load conditions, it is possible to remove the low load scenario by considering $0 \leq Q < 120,000$ as a medium load condition.

The time interval parameter Recall that for Intel CPUs, if no vector instructions are executed for approximately 675 microseconds, vector units enter a low-power mode. Setting this parameter to 500 microseconds primarily addresses potential timer inaccuracies. If CPUs of other architectures exhibit similar cold-start issues due to internal designs, this parameter should be adjusted based on the distinct cold-start behaviors.