

Measuring Parallel Execution of Classical-PQC Component Pairs in Hybrid TLS 1.3

Siwoo Eum, Minho Song, Seung-Won Lee, and Hwajeong Seo*

Hansung University, South Korea
shuraatum@gmail.com, smino0906@gmail.com
dkajdfhd1@gmail.com, hwajeong84@gmail.com

Abstract. Hybrid key exchange and Composite ML-DSA signatures in TLS 1.3 split one operation into a classical component and a post-quantum cryptography (PQC) component that do not depend on each other. Using a cryptographic workload of the TLS handshake rather than a full TLS connection, this paper measures whether running the two components at the same time on two pthread worker threads reduces latency compared with sequential execution. We evaluate the three ECDHE-ML-KEM groups of RFC 10024 with the matching Composite ML-DSA schemes, each with server authentication and mutual authentication (mTLS). Parallel execution keeps the order of operations, overlaps only the component pair of one operation, and is timed with all synchronization cost included. In an ARM and an x86 environment, parallel execution was faster in all six conditions, with speedups of $1.157\times$ – $1.281\times$ and $1.041\times$ – $1.235\times$. The gain was large in the signature stages, where the two components take similar time, and small or negative in the key exchange stages, where one component dominates. In a low-power x86 environment, however, the same source code was slower in all six conditions ($0.800\times$ – $0.917\times$). There, the overhead of waking and waiting for the worker threads exceeded the time saved, and it grew with the stage length, so a fixed cost per pair does not explain it. Independence of the components therefore does not guarantee a gain, and parallelization should be applied only after measuring in the target environment.

Keywords: Post-quantum cryptography · TLS 1.3 · Hybrid key exchange · ML-KEM · Composite ML-DSA · Mutual authentication · pthread

1 Introduction

A large enough quantum computer will break the elliptic-curve key exchange and digital signatures that TLS uses today, so NIST standardized the lattice-based key encapsulation scheme ML-KEM and the digital signature scheme ML-DSA [5,4]. Until the new algorithms are trusted enough, hybrid constructions combine classical cryptography and post-quantum cryptography (PQC) so that

* Corresponding author: hwajeong84@gmail.com.

security holds even if one of them is broken [12,2]. In TLS 1.3 [10], three hybrid key exchange groups that combine ECDHE and ML-KEM were standardized as RFC 10024 [3], and Composite ML-DSA, which joins ML-DSA and a classical signature into one signature algorithm, has been proposed for authentication [7,9].

In a hybrid construction, a TLS endpoint performs key generation, encapsulation and decapsulation, signature generation, and signature verification once with the classical algorithm and once with the PQC algorithm, and mutual authentication (mutual TLS, mTLS) adds one more signature generation and verification. The two components of a hybrid construction do not use each other’s output, so running them at the same time on different cores can ideally reduce the latency of one operation to the time of the slower component. The real gain, however, depends on the time balance of the two components and on the synchronization cost of handing the work to worker threads and waiting for them. The asymmetric operations of a handshake take tens to hundreds of μs , so if waking a thread costs about as much as the saving, parallel execution can even be slower. Existing studies of post-quantum TLS performance mainly dealt with algorithm choice and network conditions [8,11], and to the best of our knowledge, measurements of the gain and cost of running hybrid components at the same time have rarely been reported.

This paper analyzes whether running the classical component and the PQC component at the same time on two worker threads reduces latency compared with sequential execution. We narrowed the scope of parallelization on purpose: the order of operations is the same as in sequential execution, and only the classical-PQC component pair of one operation is overlapped. This narrows the cause of an observed time difference down to one factor, “running a component pair at the same time.” The contributions of this paper are as follows.

- We implement parallel execution in units of component pairs and compare its execution time, including all synchronization cost, with the sequential execution time.
- We report the total time and the five stage times for six conditions (three algorithm combinations, two authentication modes) and analyze where a gain or a loss appears, in terms of the components’ time balance and the synchronization cost.
- We show that this parallel execution reduces latency in all conditions on ARM and x86, and present a case where the same source code is slower in all conditions on low-power x86. In that case the overhead grows with the stage length, which a fixed cost per pair does not explain.

We measure a cryptographic workload that runs the asymmetric operations of a handshake in the standard order, not a full TLS connection (Section 6).

2 Background and Related Work

2.1 Hybrid Key Exchange

Hybrid key exchange in TLS 1.3 concatenates the public values of the two schemes, and also their shared secrets, and treats the result like a single

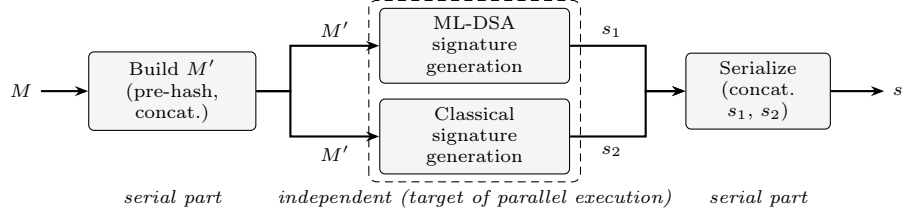


Fig. 1. Signature generation of Composite ML-DSA (s_1 : ML-DSA, s_2 : classical, s : composite signature).

group [12]. RFC 10024 defines three such groups: X25519-MLKEM768, SecP256r1-MLKEM768, and SecP384r1-MLKEM1024 [3]. The client generates an ECDHE key pair and an ML-KEM key pair and sends the public values. The server generates its own ECDHE ephemeral key pair, computes the ECDHE shared secret, and separately performs ML-KEM encapsulation with the client’s public key. The client then computes the ECDHE shared secret and performs ML-KEM decapsulation. The final shared secret is the two results concatenated in a fixed order, so the two computations before the concatenation are independent.

2.2 Composite ML-DSA

Composite ML-DSA builds a signature input M' from the message M to be signed, and ML-DSA and the classical signature algorithm each sign M' [7] (Fig. 1). M' concatenates a fixed prefix string, a label that names the algorithm combination, an application context with its length, and the pre-hash of M . In TLS 1.3, M is the content that CertificateVerify signs [10]. The final signature is the ML-DSA signature followed by the classical signature, and verification succeeds only when both component signatures are valid. Building M' and serializing the final signature are serial parts shared by the two components, and the two signature generations (or verifications) between them are independent. The use of Composite ML-DSA in CertificateVerify is defined by a separate draft [9]. Both documents are still drafts, and our workload implements version 19 and version 10, respectively.

2.3 Related Work

Paquin et al. measured the connection establishment time of hybrid key exchange and post-quantum signatures on an emulated network: above a 3–5% loss rate, algorithms whose messages are split into several packets become much slower, and with large network latency the effect of computationally slow algorithms is mostly hidden [8]. Sikeridis et al. analyzed the effect of post-quantum signature algorithms on connection establishment latency and TLS session throughput under realistic network conditions [11]. These studies deal with algorithm choice and network conditions, not with how to run the operations inside one endpoint.

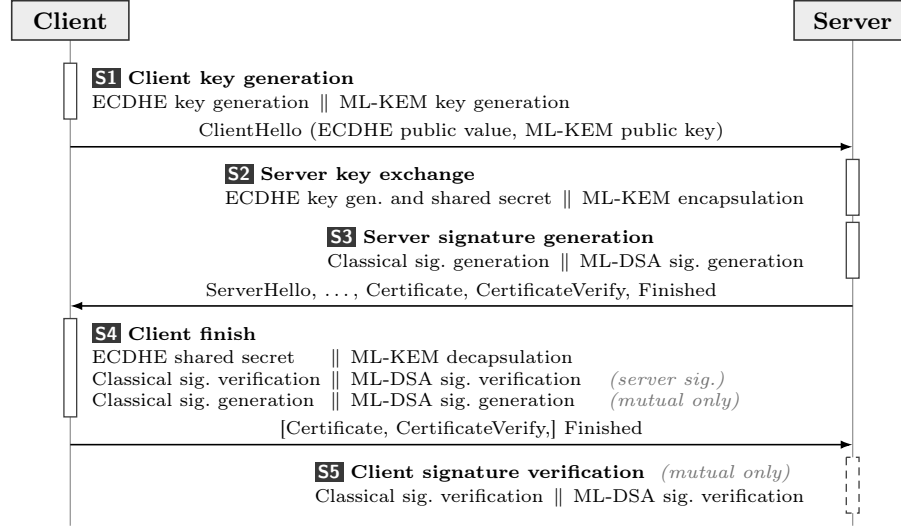


Fig. 2. TLS 1.3 handshake message flow and the five stages of the workload. $A \parallel B$ is the pair of the classical component A and the PQC component B of one operation.

The structure assumed by security studies of hybrid constructions [2], that the two components are computed independently and then combined, is the basis of the parallelism in this paper. We measure the gain from this minimum parallelism alone, using two worker threads.

3 Workload and Execution Model

3.1 Stages

The workload divides the asymmetric cryptographic operations of a TLS 1.3 handshake into five stages, following the message flow (Fig. 2). A pair consists of the classical component and the PQC component of one operation, and the pairs inside a stage run in the order shown in the figure, from top to bottom. The workload does not exchange messages: it performs the client and server operations one after another, in this order, inside one process on one device. The total time is therefore not the latency seen by one side but the sum of the cryptographic operation times of the two endpoints. Server authentication has no S5 and no client signature generation in S4, so it has 5 component pairs, and mutual authentication (mTLS) has 7. The classical component of S2 combines the server's ECDHE ephemeral key generation and the shared secret computation, so it does more work than the PQC component, which is one ML-KEM encapsulation. S4 combines two or three pairs, so its speedup is not the gain of one pair.

Long-term signing keys are generated once, outside the measured interval. Ephemeral key generation, the preparation of cryptographic contexts from stored

keys, building M' , and signature serialization are inside the measured interval in both sequential and parallel execution.

3.2 Sequential Execution and Pair-Parallel Execution

In *sequential execution*, one calling thread runs the classical and then the PQC component of each operation. *Component-pair parallel execution* (parallel execution for short) uses two worker threads created in advance. The calling thread puts the two components of one pair into a work queue, wakes the worker threads with a condition variable, and waits until both components are finished. The next pair is submitted only after one pair is finished, so the order of operations is the same as in sequential execution, and only the two components of the same operation ever run at the same time. The worker threads were not pinned to cores, and waiting used a condition variable, not busy-waiting. The time to acquire the mutex, handle the work queue, wake the worker threads, and wait for completion is all included in the parallel execution time; we call it the *synchronization cost*.

3.3 Upper Bound on the Expected Gain

For one pair, let t_c and t_q be the execution times of the classical and the PQC component, and o the synchronization cost of the pair. The sequential execution time is $t_c + t_q$, and if the execution time of a component is the same no matter which thread runs it, the parallel execution time is $\max(t_c, t_q) + o$. The *speedup* is the sequential execution time divided by the parallel execution time, and for one pair it satisfies

$$\frac{t_c + t_q}{\max(t_c, t_q) + o} \leq \frac{t_c + t_q}{\max(t_c, t_q)} \leq 2. \quad (1)$$

Parallel execution is faster if the speedup is larger than 1 and slower if it is smaller. Eq. (1) tells us three things. First, the upper bound is close to 2 when the two components take similar time and close to 1 when one side dominates. Second, the time saved is $\min(t_c, t_q) - o$, so parallel execution is slower if $o > \min(t_c, t_q)$. Third, the time that parallel execution can add is at most o per pair and does not depend on how long the components take.

A stage also contains serial parts such as building M' , serializing the signature, and restoring keys, so by Amdahl's law [1] the speedup of a stage and of the total time is smaller than the upper bound of a pair. The assumption that the execution time of a component does not depend on the thread may also fail, depending on the core's operating frequency, cache state, and type. This paper records time per stage and does not separate t_c , t_q , and o , so Eq. (1) is used only to interpret the observations in Section 5.

Table 1. Algorithm combinations.

Combination	Key exchange	Signature
ed25519	X25519 + ML-KEM-768	Ed25519 + ML-DSA-65
p256	ECDH P-256 + ML-KEM-768	ECDSA P-256 + ML-DSA-65
p384	ECDH P-384 + ML-KEM-1024	ECDSA P-384 + ML-DSA-87

4 Implementation and Measurement Method

4.1 Implementation and Experimental Conditions

The workload is implemented in C with POSIX threads (pthread). Classical operations use the OpenSSL provided by the operating system of each environment [13], ML-KEM and ML-DSA use liboqs in all environments [6], and the code is compiled with `-O2`. Before the measurement, we checked that a normal handshake succeeds and that it is rejected when the ciphertext or the signature of either component is tampered with.

We paired each of the three groups of RFC 10024 with the Composite ML-DSA scheme of the matching security level, giving three algorithm combinations (Table 1). Below, *ed25519*, *p256*, and *p384* mean not a single classical algorithm but the whole combination of key exchange and signature in this table. Applying server authentication and mutual authentication (mTLS) to each combination gives six conditions, and in mutual authentication the client uses the same signature scheme as the server.

For each condition, we ran the handshake 100 times as warm-up and then measured it 1,000 times, running sequential and parallel execution once each in every round. To reduce bias from the execution order, the order of the two was switched every round. Each stage was timed from its start to its end, and the whole measurement was repeated five times. Times are the median of the five per-repeat medians, and the speedup is the median of the per-repeat ratios.

4.2 Measurement Environments

Table 2 shows the three measurement environments. The main targets are the ARM environment (Apple M2) and the x86 environment (the desktop processor Intel Core i7-13700KF). The low-power x86 environment is a small PC (Intel NUC) with the low-power mobile processor Intel Core i7-1165G7. Below, ARM, x86, and low-power x86 are names for these three environments and do not represent all processors of each instruction set. The source code is the same in the three environments, and the SHA-256 hash of all source files matches. The x86 and low-power x86 environments have the same versions of OpenSSL, liboqs, and the compiler and differ in the kernel version, while ARM differs in the operating system, OpenSSL, and the compiler.

Table 2. Measurement environments. P is a performance core and E is an efficiency core.

	ARM	x86	low-power x86
Processor	Apple M2	Intel Core i7-13700KF	Intel Core i7-1165G7
Cores	8 (4P+4E)	16 (8P+8E)	4
Threads	8	24	8
Instruction set	arm64	x86-64	x86-64
Compiler	Apple clang 17.0.0	GCC 13.3.0	GCC 13.3.0
OpenSSL	3.6.2	3.0.13	3.0.13
liboqs	0.15.0	0.15.0	0.15.0

Table 3. Total time and speedup for each condition. Time: μs , lower is better. Speedup: sequential/parallel time, higher is better (> 1 : parallel is faster). Range: min–max over repeats.

Environment	Combination	Auth.	Seq. (μs)	Par. (μs)	Speedup	Range
ARM	ed25519	server	727.0	619.5	$1.174\times$	1.148–1.208
ARM	ed25519	mutual	1315.5	1106.0	$1.189\times$	1.155–1.204
ARM	p256	server	699.0	604.0	$1.157\times$	1.149–1.175
ARM	p256	mutual	1290.0	1103.5	$1.177\times$	1.164–1.183
ARM	p384	server	1754.0	1431.0	$1.227\times$	1.220–1.230
ARM	p384	mutual	2839.5	2207.0	$1.281\times$	1.273–1.291
x86	ed25519	server	627.8	516.2	$1.208\times$	1.200–1.228
x86	ed25519	mutual	1106.0	897.8	$1.235\times$	1.223–1.260
x86	p256	server	733.7	612.2	$1.194\times$	1.161–1.207
x86	p256	mutual	1190.5	976.1	$1.220\times$	1.176–1.243
x86	p384	server	3420.0	3287.0	$1.041\times$	1.037–1.050
x86	p384	mutual	4652.0	4389.7	$1.057\times$	1.054–1.060
low-power x86	ed25519	server	810.8	1003.7	$0.808\times$	0.801–0.837
low-power x86	ed25519	mutual	1412.3	1746.6	$0.807\times$	0.803–0.848
low-power x86	p256	server	906.1	1118.4	$0.809\times$	0.802–0.828
low-power x86	p256	mutual	1485.3	1857.6	$0.800\times$	0.790–0.830
low-power x86	p384	server	5496.9	5994.4	$0.917\times$	0.905–0.920
low-power x86	p384	mutual	7283.7	8046.1	$0.907\times$	0.903–0.911

5 Performance Evaluation

We first analyze ARM and x86; low-power x86, where parallel execution became slower, is discussed in Section 5.4. The tables show all three environments.

5.1 Total Time

In Table 3, parallel execution was faster in all six conditions on ARM ($1.157\times$ – $1.281\times$) and x86 ($1.041\times$ – $1.235\times$). The whole range was above 1 in every condition, so the result was consistent over the five repeats, but the speedup stayed far below the upper bound of 2 in Eq. (1).

The two environments showed opposite trends across combinations: p384 had the largest speedup on ARM but the smallest on x86. For server authentication, the sequential total time of p384 is 1754.0 μs on ARM but 3420.0 μs on x86, and 61% of this difference comes from the key exchange stages S1 and S2. Absolute times include software stack differences and cannot be compared across environments, but if P-384 operations dominate the key exchange stages on x86, the

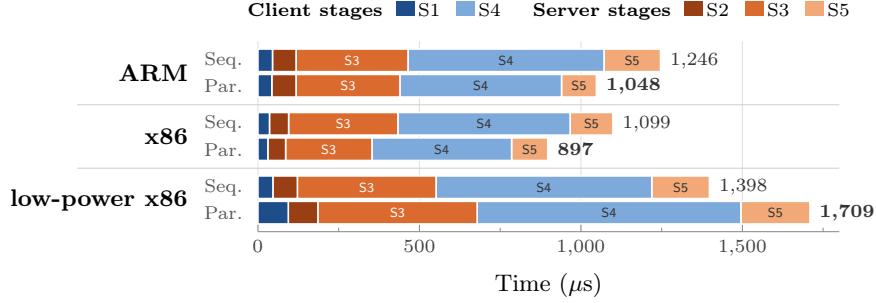


Fig. 3. ed25519 combination (Table 1), mutual authentication: stage times in the three environments (μs , lower is better). Blue: client stages; orange: server stages; bar-end number: sum of the five stages.

upper bound of Eq. (1) is itself close to 1. The effect of the authentication mode was small: mutual authentication had a slightly higher speedup than server authentication for all three combinations in both environments, but the two ranges sometimes overlap, so we do not claim it as a confirmed effect.

5.2 Results by Stage

Fig. 3 shows the stage times for the ed25519 mutual authentication condition. In all three environments, the signature stages S3, S4, and S5 take most of the time and account for most of the gain and loss. Each bar is the sum of the five stage times, which is slightly shorter than the total time in Table 3 because the total time also includes the processing between stages. Table 4 shows the speedup of each stage. A speedup smaller than 1 appeared in 4 of 27 stages on ARM and in none on x86. Except for S3 of ed25519 mutual authentication on ARM (range 0.997–1.126), the per-repeat range was on the same side of 1 in every stage. Table 5 shows the time change caused by parallel execution in S1 and S2. These stages have only one component pair, so the change corresponds to $o - \min(t_c, t_q)$ in Eq. (1).

ARM environment. The four cases that became slower were all S2 of ed25519 and p256. For example, S2 of p256 server authentication grew from 78.0 μs to 84.0 μs . The short length of the stage alone does not explain this: S1 of the same condition (40.0 μs) is shorter than S2 and still showed a speedup of 1.333 $\times$. As Fig. 2 shows, the classical component of S2 combines key generation and the shared secret computation, so the two components are unbalanced, and the time that can be overlapped, $\min(t_c, t_q)$, was probably smaller than the synchronization cost. The timer resolution of this environment is 1 μs and the change in these stages is 2–6 μs , so the size of the loss itself is not precise. The gain in the signature stages (up to 1.620 $\times$) was much larger than the loss in S2, so the total time decreased in all conditions.

Table 4. Speedup by stage (higher is better).

Combination	Stage	ARM		x86		low-power x86	
		server	mutual	server	mutual	server	mutual
ed25519	S1	1.023	1.047	1.165	1.171	0.550	0.499
ed25519	S2	0.973	0.973	1.086	1.090	0.851	0.826
ed25519	S3	1.102	1.107	1.254	1.267	0.885	0.875
ed25519	S4	1.412	1.219	1.173	1.235	0.775	0.817
ed25519	S5	–	1.620	–	1.185	–	0.839
p256	S1	1.333	1.317	1.167	1.189	0.512	0.459
p256	S2	0.929	0.929	1.042	1.051	0.876	0.807
p256	S3	1.143	1.125	1.302	1.308	0.871	0.867
p256	S4	1.293	1.202	1.124	1.226	0.794	0.802
p256	S5	–	1.404	–	1.180	–	0.815
p384	S1	1.119	1.110	1.021	1.029	0.847	0.843
p384	S2	1.029	1.032	1.006	1.009	0.946	0.946
p384	S3	1.405	1.401	1.115	1.123	0.927	0.922
p384	S4	1.254	1.318	1.035	1.064	0.902	0.902
p384	S5	–	1.410	–	1.080	–	0.869

x86 environment. The largest improvement was S3 of p256 mutual authentication ($1.308\times$) and the smallest was S2 of p384 server authentication ($1.006\times$). In Table 5, the time saved in S1 and S2 is similar ($4.6\text{--}7.7\ \mu\text{s}$) whether the sequential execution time of the stage is $31.9\ \mu\text{s}$ or $1085.1\ \mu\text{s}$. The time saved cannot exceed the time of the shorter component, so this is explained if the shorter side is the ML-KEM component and its time changes little across combinations. In that case, S1 and S2 of p384 are long but dominated by the classical component, so the upper bound $(t_c + t_q)/\max(t_c, t_q)$ is close to 1.

5.3 Results by Endpoint

In a real connection, the client performs S1 and S4 and the server S2, S3, and S5. Table 6 shows the speedup of each endpoint, computed from the sum of its stage times.

In server authentication, the two endpoints gained different amounts, and which one gained more was opposite in the two environments. For ed25519 and p256 on ARM, the client (1.335, 1.299) gained more than the server (1.080, 1.098): the gain of S4, which contains signature verification, is large, while the server’s S2 becomes slower and the gain of S3 is small (Table 4). On x86, the server gained more than the client for all three combinations, because the largest gain is in the server’s signature generation (S3). In both cases the per-repeat ranges of the two endpoints did not overlap. In mutual authentication, both endpoints generate and verify signatures, so the difference shrinks and the two ranges overlapped for all combinations in both environments. On low-power x86, the client slowed down more than the server in all six conditions, and in server authentication the client’s S1 was the stage with the largest slowdown.

5.4 A Case Where Parallel Execution Is Slower: Low-Power x86

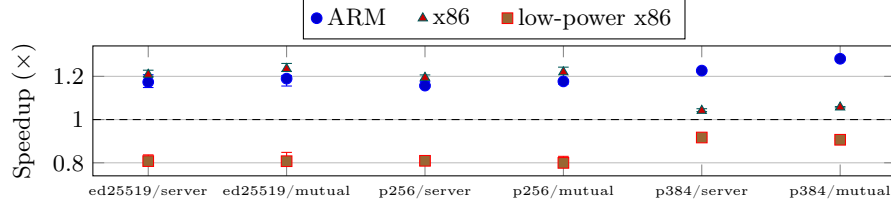
The two environments above suggest that parallel execution of component pairs is always a gain, but the same source code and procedure on low-power x86

Table 5. Sequential execution time of S1 and S2 and the change caused by parallel execution (parallel minus sequential; μ s, server authentication). Negative: parallel is faster.

Combination	Stage	ARM		x86		low-power x86	
		Seq.	Change	Seq.	Change	Seq.	Change
ed25519	S1	45.0	-1.0	34.9	-4.9	45.6	+37.5
ed25519	S2	73.0	+2.0	58.7	-4.7	74.8	+13.2
p256	S1	40.0	-10.0	31.9	-4.8	40.9	+38.2
p256	S2	78.0	+6.0	114.8	-4.6	136.8	+19.1
p384	S1	141.0	-15.0	423.3	-7.7	705.8	+134.0
p384	S2	352.0	-10.0	1085.1	-5.8	1799.0	+102.8

Table 6. Speedup by endpoint (higher is better). C is the client (S1, S4) and S is the server (S2, S3, S5).

Combination	Auth.	ARM		x86		low-power x86	
		C	S	C	S	C	S
ed25519	server	1.335	1.080	1.172	1.225	0.730	0.880
ed25519	mutual	1.205	1.198	1.231	1.223	0.784	0.857
p256	server	1.299	1.098	1.127	1.224	0.746	0.872
p256	mutual	1.210	1.152	1.223	1.216	0.769	0.840
p384	server	1.228	1.233	1.033	1.047	0.886	0.939
p384	mutual	1.294	1.285	1.057	1.057	0.891	0.924

**Fig. 4.** Speedup of the total time for each condition in the three environments (sequential/parallel time, higher is better). Points are medians and bars are min-max ranges.

give the opposite result (Table 3, Fig. 4). Parallel execution was not faster in any condition: the speedup was $0.800\times$ – $0.917\times$, with the whole range below 1 in every condition. All 27 stages also became slower (Table 4). The largest slowdown was S1 of p256 mutual authentication ($0.459\times$), and the stage closest to 1 was S2 of p384 ($0.946\times$ with both authentication modes). This environment has the same instruction set as x86 and the same versions of OpenSSL, liboqs, and the compiler, so the effect of parallelization depends not on the instruction set or the libraries but on the individual environment.

Overhead of parallel execution. For every pair, the work is put into the queue, the two sleeping worker threads are woken, and the calling thread is woken again after both components finish; in a short stage this time appears directly as a loss. For example, S1 of ed25519 server authentication takes only $45.6\ \mu$ s in sequential execution but took $83.1\ \mu$ s in parallel execution: the cost of waking the threads

and waiting was larger than the time of the shorter component, the most that overlapping can save. The same stage became slightly faster in the other two environments (Table 5).

Stage length and the time change. This overhead is, however, more than a fixed setup time paid once per pair. In the model of Eq. (1), the time added to one pair is at most o and independent of the component length. But in Table 5, the change in this environment is $+37.5 \mu\text{s}$ and $+13.2 \mu\text{s}$ for ed25519 but $+134.0 \mu\text{s}$ and $+102.8 \mu\text{s}$ for p384, so it grew with the stage length. Either the synchronization cost grew with the waiting time, or the components themselves ran more slowly on the worker threads. The former happens if a longer wait lets the core of the calling thread enter a deeper idle state that takes longer to wake from. The latter happens if the operating frequency of a core woken from the idle state did not rise enough, or if the maximum frequency per core dropped because two cores were active at the same time. Both can happen in a low-power processor with tight power and thermal limits, and our measurement cannot tell them apart. Even so, this case shows that the effect of parallel execution must be measured in the target environment before it is applied.

6 Limitations

Our workload reproduces the cryptographic operations and their order from the standard documents, without the TLS record layer, socket I/O, or certificate path validation, so the results cannot directly predict the connection latency or throughput of a real service. The parallel execution is also only one possible implementation: the results may change if the calling thread runs one component itself or if busy-waiting or core pinning is used, and we did not consider the extra CPU time and energy that parallel execution consumes. Differences between environments are not the effect of the processor alone. ARM differs from the other two environments in the operating system, OpenSSL, and the compiler, the two x86 environments differ in the kernel version and the core configuration, and only one device was measured per environment, so an environment’s name cannot be generalized to all processors of that family. For low-power x86, we only listed candidate causes of the slowdown without verifying them. Each environment was measured once with five repeats, and the ARM timer resolution is $1 \mu\text{s}$, so the speedup of stages that take tens of μs has low precision.

7 Conclusion

This paper measured, for three algorithm combinations and two authentication modes, whether the simplest parallelization, running one classical-PQC component pair of hybrid key exchange and Composite ML-DSA at the same time on two worker threads, reduces latency. On ARM and x86, latency decreased in all six conditions, with speedups of $1.157\times$ – $1.281\times$ and $1.041\times$ – $1.235\times$, far below the ideal upper bound of 2. The stage results are better explained by the time

balance of the two components than by the stage length: for ed25519 and p256 on ARM, the unbalanced S2 lost time while the shorter S1 gained, and on x86 the long key exchange stages of p384 gained the least.

This gain did not appear in every environment. On low-power x86, the same source code was slower in all six conditions ($0.800\times$ – $0.917\times$) and in every stage: the overhead of waking and waiting for the worker threads was larger than the time saved. The added time also grew with the stage length, so at least one assumption failed there: a constant synchronization cost, or a component execution time independent of the thread. This environment has the same instruction set and main library versions as x86, so the effect cannot be generalized by instruction set. Independence of the components only makes parallelization possible; it does not guarantee a gain, which must be checked in the target environment.

Future work will verify the model of Section 3 by separately measuring each component’s execution time and the per-pair synchronization cost, find the cause of the low-power x86 slowdown with performance counters and scheduler tracing, and compare with a design in which the calling thread runs one component itself.

References

1. Amdahl, G.M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: AFIPS Spring Joint Computer Conference. pp. 483–485 (1967). <https://doi.org/10.1145/1465482.1465560>
2. Bindel, N., Brendel, J., Fischlin, M., Goncalves, B., Stebila, D.: Hybrid Key Encapsulation Mechanisms and Authenticated Key Exchange. In: Post-Quantum Cryptography – PQCrypto 2019. pp. 206–226. LNCS, Springer (2019). https://doi.org/10.1007/978-3-030-25510-7_12
3. Kwiatkowski, K., Kampanakis, P., Westerbaan, B., Stebila, D.: Post-Quantum Traditional (PQ/T) Hybrid Key Agreement Mechanisms for TLS 1.3. RFC 10024 (2026). <https://doi.org/10.17487/RFC10024>
4. National Institute of Standards and Technology: Module-Lattice-Based Digital Signature Standard. Tech. Rep. FIPS 204, NIST (2024). <https://doi.org/10.6028/NIST.FIPS.204>
5. National Institute of Standards and Technology: Module-Lattice-Based Key-Encapsulation Mechanism Standard. Tech. Rep. FIPS 203, NIST (2024). <https://doi.org/10.6028/NIST.FIPS.203>
6. Open Quantum Safe: liboqs 0.15.0 (2025), <https://github.com/open-quantum-safe/liboqs/releases/tag/0.15.0>
7. Ounsworth, M., Gray, J., Pala, M., Klaußner, J., Fluhrer, S.: Composite Module-Lattice-Based Digital Signature Algorithm (ML-DSA) for use in X.509 Public Key Infrastructure. Internet-Draft, draft-ietf-lamps-pq-composite-sigs-19 (work in progress) (2026), <https://datatracker.ietf.org/doc/draft-ietf-lamps-pq-composite-sigs/>
8. Paquin, C., Stebila, D., Tamvada, G.: Benchmarking Post-quantum Cryptography in TLS. In: Post-Quantum Cryptography – PQCrypto 2020. pp. 72–91. LNCS, Springer (2020). https://doi.org/10.1007/978-3-030-44223-1_5
9. Reddy, T., Hollebeek, T., Gray, J., Fluhrer, S., Van Geest, D.: Use of Composite ML-DSA in TLS 1.3. Internet-Draft, draft-reddy-tls-composite-mldsa-10 (work in progress) (2026), <https://datatracker.ietf.org/doc/draft-reddy-tls-composite-mldsa/>

10. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (2018). <https://doi.org/10.17487/RFC8446>
11. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Post-Quantum Authentication in TLS 1.3: A Performance Study. In: Network and Distributed System Security Symposium (NDSS) (2020). <https://doi.org/10.14722/ndss.2020.24203>
12. Stebila, D., Fluhrer, S., Gueron, S.: Hybrid Key Exchange in TLS 1.3. RFC 9954 (2026). <https://doi.org/10.17487/RFC9954>
13. The OpenSSL Project Authors: OpenSSL: Cryptography and SSL/TLS Toolkit (2026), <https://www.openssl.org/>, versions 3.0.13 and 3.6.2