PQMagic: Towards Secure and Efficient Post Quantum Cryptography Implementations

Yituo He[1(✉)[0009−0003−0563−9022]], Xinpeng Hao[1], Juanru Li[3], and Yu Yu[1,2]

[1] Shanghai Qi Zhi Institute, Shanghai, China
yituohe@163.com
haoxinpeng@sqz.ac.cn
[2] Shanghai Jiao Tong University, Shanghai, China
yyuu@sjtu.edu.cn
[3] Feiyu Tech, Shanghai, China
mail@lijuanru.com

Abstract. Quantum computing threatens current public-key cryptosystems, driving the need for post-quantum cryptography (PQC). However, PQC implementations face additional risks. We find implementation issues in existing PQC libraries (e.g., pq-crystals and liboqs), while they also fail to fully leverage modern processors. To address these issues, we propose four optimization strategies: Branch Optimization, Register Allocation, Vectorized Execution, and Secure and Efficient Pipelining. These strategies minimize execution branches, instruction counts, and memory accesses while enhancing security, mitigating the implementation from side-channel attack risk. We implement these in PQMagic, a high-performance PQC library for ML-KEM and ML-DSA, and it significantly outperforms state-of-the-art libraries. For ML-KEM-1024, it achieves efficiency gains of up to 1.77x, 1.79x, and 1.52x for Keygen, Encaps, and Decaps, while reducing instruction counts and memory access overhead by up to 47.1% and 60.1%. For ML-DSA-87, it improves Keygen, Sign, and Verify by up to 2.24x, 1.89x, and 2.04x, with instruction counts and memory access reduced by up to 44.4% and 64.5%. Additionally, PQMagic eliminates up to 90.9% of branch operations in matrix expansion for ML-DSA. Besides, PQMagic also outperforms traditional cryptographic algorithm combinations (RSA-2048/ECDSA-256 + ECDH) selected from OpenSSL. It only has a slight gap at the highest level L5 compared to ECDSA-256 + ECDH combination. Our work shows that combining modern hardware capabilities with careful instruction scheduling enables secure and efficient PQC implementations, paving the way for post-quantum cryptographic migration.

Keywords: Post-Quantum Crypto · Crypto Engineering · PQMagic

1 Introduction

Quantum computing threatens to break many public-key cryptosystems currently in use. Consequently, research into post-quantum cryptography (PQC)

focuses on developing cryptosystems secure against both quantum and classical computers. Unfortunately, not all threats against PQC come from quantum computing. As an empirical study [6] demonstrated, systems-level bugs—rather than inherent flaws in the underlying ciphers—are a major contributor to security vulnerabilities in many widely used cryptographic libraries. This conclusion also applies to PQC libraries. Since PQC implementations must interoperate with existing processors, operating systems, and networks, software security issues like memory corruption remain a critical concern. Moreover, PQC libraries remain vulnerable to traditional side-channel attacks (e.g., those exploiting timing leaks to extract secret-key information). Thus, it is critical to develop secure, efficient PQC software components and integrate them into modern cryptographic infrastructure to mitigate both quantum and conventional threats.

PQC software especially PQC libraries, like other crypto software, need to apply a rigorous security standard for their implemented code. We argue that although many current crypto libraries have deployed various code securing strategies (e.g, using specific hardware ISA such as AES-NI and SHA-NI to secure the crypto operations, using formal verification to avoid non-constant time execution), PQC libraries are less concerned. Even though a variety of PQC crypto libraries have been developed, to the best of our knowledge, seldom work thoroughly examined to what extent these libraries adopted secure code implementation to protect them against common attacks. In this paper, we first demonstrate the implementation issues in current PQC libraries and discuss the risks. We found that our investigated PQC libraries (pq-crystals[7,10] and liboqs[19]) did not fully leverage the features of modern processors, hence led to a not-that-optimal code implementation that potentially increased the risks of timing side-channel or secret information disclosure.

In response, we propose a series of code securing strategies that mainly utilize four strategies to generate secure and efficient implementations of current PQC ciphers such as ML-KEM and ML-DSA. We have implemented PQMagic, a PQC library that followed the proposed code securing strategies and demonstrated that our implementation significantly reduced execution branches, instruction counts, and memory accesses. In comparison with state-of-the-art optimized PQC libraries, the optimal implementation of PQMagic achieves significant reductions in instruction counts and memory access overhead. For ML-KEM-1024, these reductions reach up to 47.1% and 60.1%, respectively. Similarly, for ML-DSA-87, PQMagic reduces instruction counts and memory access overhead by up to 44.4% and 64.5%, respectively. Additionally, PQMagic successfully eliminates branch operations in matrix expansion for ML-DSA by up to 90.9% compared to state-of-the-art libraries. In terms of performance, PQMagic significantly outperforms state-of-the-art optimized PQC libraries. For ML-KEM-1024, PQMagic achieves efficiency gains of up to 1.77x, 1.79x, and 1.52x for Keygen, Encaps, and Decaps operations, respectively. Similarly, for ML-DSA-87, PQMagic demonstrates improvements of 2.24x, 1.89x, and 2.04x for Keygen, Sign, and Verify operations, respectively. Notably, we evaluate PQMagic by combining ML-KEM and ML-DSA together against traditional cryp-

tographic algorithm combinations (RSA-2048/ECDSA-256 + ECDH) selected from OpenSSL. PQMagic outperforms the RSA-2048 + ECDH combination across all security levels, achieving efficiency improvements of 4.77x at L1, 3.25x at L3, and 2.67x at L5. When compared to ECDSA-256 + ECDH, PQMagic maintains superior performance at L1 and L3 (1.58x and 1.07x, respectively), with only a slight efficiency gap (0.88x) at the highest security level L5. These results showed that security of PQC software implementation could benefit from both new hardware ISAs and proper instruction scheduling.

2 Security Issues in Current PQC Implementations

The rapid evolution of modern CPU architectures has provided developers with substantial hardware resources [4,8,12,18], such as vector instruction sets (e.g., AVX512), multi-level cache hierarchies, out-of-order execution, and superscalar pipelines. These resources not only significantly improve the performance of code implementations but also help mitigate security risks.

However, through extensive analysis of existing works [7,10,14,19,20,21], we have found that many implementations of ML-KEM and ML-DSA often overlook security issues in the pursuit of performance optimization, fail to fully utilize the features and resources of modern CPUs. For instance, the pq-crystals library [7,10] optimizes ML-KEM and ML-DSA using only the AVX2 instruction set, while liboqs [19] provides further encapsulation based on this work. Although these open-source libraries have made some progress in performance optimization, they still fall short in mitigating the security risk of algorithm implementations. Its frequent memory accesses and branch-heavy code structures not only hinder further performance improvements but also introduce risks such as timing side-channel attacks and memory exposure. Moreover, as these libraries do not fully leverage the architectural advantages of modern CPUs, there remains significant room for performance optimization. As for some closed-source optimizations [20,21], while they claim to utilize the AVX512 instruction set for performance gains, they also lack a thorough consideration of algorithm security and fail to fully exploit instruction pipeline optimizations to enhance both security and computational efficiency.

To provide a more concrete illustration of the security risks in existing algorithm implementations, we analyzed the implementation details of the state-of-the-art open-source libraries, pq-crystals [7,10] and liboqs [19]. As shown in Figure 1, both pq-crystals and liboqs use switch-case structures in their source code. The switch-case structure generates multiple branch instructions after compilation. When branch prediction fails, the CPU must flush the pipeline and reload instructions, incurring performance penalties. Beyond these computational inefficiencies, the current implementation exhibits suboptimal design characteristics. This branch-heavy pattern represents poor programming practice, as its adoption in other code contexts could potentially introduce additional side-channel security risk.

```c
static inline void polyvec_matrix_expand_row (/* ... */) {
  switch(i) {
    case 0:
      polyvec_matrix_expand_row0 (buf, buf + 1, rho);
      *row = buf;
      break;
    case 1:
      polyvec_matrix_expand_row1 (buf + 1, buf, rho);
      *row = buf + 1;
      break;
    /* Many Other Cases */
    case 7:
      polyvec_matrix_expand_row7 (buf + 1, buf, rho);
      *row = buf + 1;
      break;
  }
}
```

Fig. 1: Risky Branch-Prone Control Flow Pattern in pq-crystal[10] and liboqs[19]

To further verify the potential security risks introduced by the branching structure in the source code, we conducted a comprehensive reverse engineering analysis of binaries compiled with *O3* optimization. We found that the compiler retained the switch-case branch structures in all cases except the lowest security level parameter set (i.e., ML-DSA-44), failing to optimize them. This indicates that switch-case structures will truly risk the algorithms in real-world deployments.

Additionally, Figure 2 shows that these implementations involve extensive memory read/write operations. Each macro function is called four times and contains multiple memory operations. This design introduces high memory access overhead, which also suffers a significant risk of memory exposure. The high frequency of memory accesses increases the risk of cache side-channel vulnerabilities, through which sensitive data could potentially be extracted by attackers.

3 Method

To mitigate security issues in the code optimization process while further improving algorithm execution efficiency, we propose the following four optimization strategies:

- **Branch Optimization**: Reducing complex branches to reduce the impact of misprediction while demonstrating a coding paradigm that inherently mitigates side-channel risks across all coding scenarios.
- **Register Allocation**: Maximizing data residency in registers to reduce memory accesses, thereby lowering the risk of data leakage through cache access patterns.

```
.macro levels2t7 off
/* level 2 */
vmovdqa /*offset*/(%rdi),%ymm4
/* 7 More vmovdqa */
vpbroadcastd
↪  /*offset*/(%rsi),%ymm1
vpbroadcastd
↪  /*offset*/(%rsi),%ymm2
/* Other Operations */

/* level 3 */
vmovdqa /*offset*/(%rsi),%ymm1
vmovdqa /*offset*/(%rsi),%ymm2
/* Other Operations */

/* level 4 */
vmovdqa /*offset*/(%rsi),%ymm1
vmovdqa /*offset*/(%rsi),%ymm2
/* Other Operations */

/* ... */

/* level 7 */
/* ... */

/* Store the Result Back */
vmovdqa %ymm9,/*offset*/(%rdi)
/* 7 More vmovdqa */
.endm
```

```
.macro levels0t1 off
/* level 0 */
vpbroadcastd
↪  /*offset*/(%rsi),%ymm1
vpbroadcastd
↪  /*offset*/(%rsi),%ymm2
vmovdqa /*offset*/(%rdi),%ymm4
/* 7 More vmovdqa */
/* Other Operations */

/* level 1 */
vpbroadcastd
↪  /*offset*/(%rsi),%ymm1
/* 3 More vpbroadcastd */
/* Other Operations */

vmovdqa %ymm4,/*offset*/(%rdi)
/* 7 More vmovdqa */
.endm

.text
.global cdecl(ntt_avx)
cdecl(ntt_avx):
vmovdqa _8XQ*4(%rsi),%ymm0
levels0t1        0
/* 3 more levels0t1 */
levels2t7        0
/* 3 more levels2t7 */
ret
```

Fig. 2: Risky Memory Access Patterns in NTT Implementations in pq-crystal and liboqs

- **Vectorized Parallel Execution**: Leveraging SIMD instructions to accelerate computations, reducing the total number of instructions required for the same operation as well as improving data parallelism, and thus shortening execution time. This helps mitigate certain security threats, such as timing side-channel attacks, by minimizing the time window available for adversarial observation.
- **Secure and Efficient Pipelining**: Mitigating the risk of the compiler introducing unsafe data dependencies or control dependencies that could inadvertently leak information [16] by directly writing assembly codes, while optimizing execution order to better align with CPU pipeline structures for both security and performance.

This section will detail how these optimization strategies work together to enhance algorithm security while fully leveraging the performance potential of modern hardware architectures.

3.1 Branch Optimization

Our design employs a branch optimization strategy that systematically eliminates unnecessary branching dependencies. This architectural decision stems from the performance challenges associated with conditional branching, where variable execution timing can disrupt pipeline efficiency and computational predictability.

By replacing conventional switch-case constructs with dedicated function implementations for each logical path, we achieve more deterministic execution behavior. The branch-free design not only ensures consistent processing timelines but also establishes a better coding paradigm.

The optimized architecture consolidates all polynomial sampling operations into a unified function implementation. By eliminating branching constraints, this approach achieves full utilization of AVX512 instruction sets, enabling simultaneous processing of eight polynomials with enhanced compiler optimization opportunities. Furthermore, the reduction in function call overhead also contributes to overall execution efficiency and system stability.

3.2 Register Allocation

Memory access latency is a critical bottleneck in secure and high-performance computing. To mitigate this issue, we prioritize storing critical data (e.g., polynomial coefficients) in registers, minimizing memory accesses and reducing the risk of data leakage through cache access patterns.

Modern AVX512 registers provide robust hardware support for secure and efficient computation. Each register can store 512 bits of data, and with AVX512 instruction set, the number of available registers is expanded to 32. This allows us to retain more data in registers, reducing reliance on memory and mitigating potential side-channel vulnerabilities.

For the ML-KEM algorithm, every polynomial contains 256 coefficients, with each coefficient being 12 bits long (stored as 16-bit values). This requires only 8 AVX512 registers to store all coefficients. Similarly, for the ML-DSA algorithm, every polynomial consists of 256 coefficients, with each coefficient being 23 bits long (stored as 32-bit values), requiring only 16 AVX512 registers. Even after allocating registers for coefficient storage, there remain 24/16 available 512-bit registers for ML-KEM/ML-DSA, which can be leveraged for intermediate computations. By preloading polynomial coefficients into registers, we minimize memory access patterns that could otherwise be exploited in cache-based side-channel attacks.

3.3 Vectorized Parallel Execution

Modern CPUs provide SIMD instruction sets such as AVX512, which allow us to process multiple data elements in configurations like 32×16-bit or 16×32-bit simultaneously. This capability is particularly beneficial for cryptographic algorithms such as Keccak-1600 and polynomial arithmetic in ML-KEM as well

as ML-DSA, where efficient parallelism not only accelerates computation but also enhances security by reducing exposure to timing-based side-channel attacks.

For Keccak-1600, AVX512 enables up to 8-way parallel processing of independent hash operations, allowing multiple 64-bit state computations to be performed simultaneously within a single instruction cycle. Similarly, ML-KEM and ML-DSA rely heavily on polynomial arithmetic, where coefficients are stored as 32-bit and 16-bit values, respectively. Since computations on different coefficients exhibit minimal data dependency, they are well-suited for vectorized execution. By utilizing AVX512, multiple coefficients can be processed in parallel within a single register, effectively reducing the total number of instructions required for the same operation. This not only improves execution efficiency but also narrows the attack window for adversaries, further mitigating the risk of side-channel attacks.

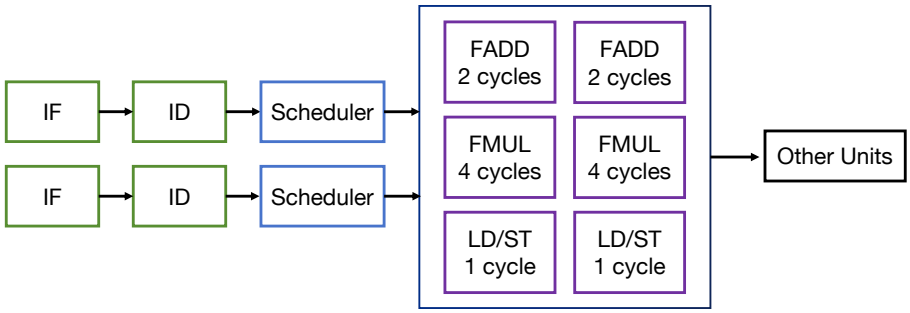


Fig. 3: Illustration of Modern CPU Pipeline

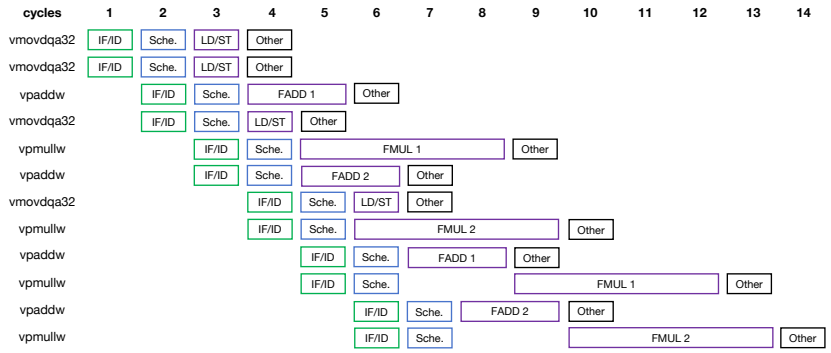
3.4 Secure and Efficient Pipelining

Instruction pipelining is a fundamental mechanism for achieving instruction-level parallelism (ILP) [1,2]. We directly use assembly code to implement algorithm components, controlling instruction scheduling order. Then we take advantage of multiple execution units in modern superscalar processors, optimally arranging similar types of assembly instructions together, rather than simply interleaving different types of instructions. Therefore, we reduce potential security risks introduced by compilers [16] while improving the ILP.

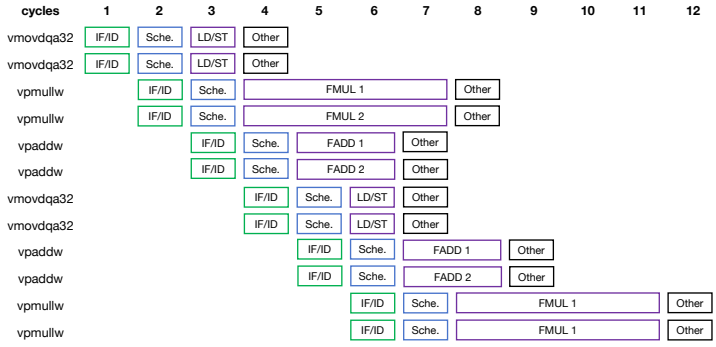
Specifically, modern processors (e.g., AMD and Intel CPUs) enhance parallelism through superscalar architecture [8,12,18], which provides the following key features:

– **Multiple Execution Units**: Independent functional units (e.g., integer ALUs, FPUs, load/store units) allow concurrent execution of heterogeneous operations, reducing control dependencies that could leak execution timing variations.

– **Multi-Issue Capability**: Modern CPUs can fetch, decode, and dispatch multiple instructions per cycle, enabling higher throughput while minimizing compiler-induced timing variations.



(a) Instruction Scheduling with Low Parallelism



(b) Optimized Instruction Scheduling with Improved Parallelism

Fig. 4: Comparison of Instruction-Level Parallelism Before and After Optimization on a Superscalar CPU

Figure 3 shows a simplified diagram of a modern superscalar CPU pipeline structure, including two instruction fetch (IF) units, two instruction decode (ID) units, two instruction scheduler units, and six execution units with three different types (i.e., FADD, FMUL, and LD/ST).

To illustrate the security and performance benefits of increasing ILP on superscalar CPUs, we designed a toy example consisting of 12 instructions, as shown in Figure 4. We aim to compare different instruction scheduling schemes on our simplified CPU architecture (Figure 3). We then ensure that the optimized

scheduling scheme achieves improved performance while mitigating compiler-introduced security risks at the same time.

In Figure 4a, although instructions are interleaved to facilitate parallel execution across different execution unit types, the lack of consideration for parallelism within the same type of execution units leads to 2 pipeline stalls, requiring 14 cycles to complete.

In contrast, the optimized secure instruction scheduling scheme in Figure 4b carefully arranges instructions to balance execution across different execution units. By overlapping instructions of the same type and filling idle cycles of high-latency instructions with other types, the schedule eliminates pipeline stalls successfully. This approach reduces the total cycle count to 12, improving performance by 14.3%. Besides that, manually scheduling the instruction pipeline using hand-writing assembly directly could naturally mitigate the risk of timing side-channel attacks introduced by compilers.

4 PQMagic

In this section, we present our PQC library, PQMagic[4]. It applies optimization strategies proposed in Section 3 and systematically reduces the risk of security issues while improving computational efficiency for ML-KEM/ML-DSA implementation. The implementation of PQMagic focuses on securing three critical operations: **polynomial arithmetic**, **hash components**, and **matrix expansion**.

For **polynomial arithmetic**, we employed the *Register Allocation* to maximize data residency, *Vectorized Parallel Execution* to accelerate computations, and *Secure and Efficient Pipelining* to optimize instruction scheduling, ensuring both efficiency and mitigation to side-channel attacks. For **hash components**, we leveraged *Vectorized Parallel Execution* to process multiple data elements simultaneously, reducing the total number of instructions and minimizing the time window for potential timing attacks. For **matrix expansion**, we applied *Branch Optimization* to eliminate complex conditional branches introduced by switch-case structure, which reduces the risk of timing side-channel attacks. Besides, it gives an opportunity for parallelism sampling in matrix expansion and avoids performance penalties caused by frequent branch mispredictions.

The remainder of this section provides a detailed explanation of how these strategies are applied to optimize the implementation of the three critical operations.

4.1 Polynomial Arithmetic

Polynomial arithmetic remains one of the most computationally intensive core operations in both ML-KEM and ML-DSA algorithms, and its performance and security significantly impact the overall efficiency of these cryptographic schemes.

[4] https://pqcrypto.dev/

Among the components of polynomial arithmetic, the Number Theoretic Transform (NTT) and its inverse operation (INTT) are particularly critical, as they not only dominate the computational cost but also present a potential attack surface. Therefore, to enhance both the security and efficiency of polynomial arithmetic, we focused on optimizing NTT operations, aiming to reduce their vulnerability to side-channel attacks while improving their execution efficiency.

Modern processors widely support the AVX512 instruction set, which provides 512-bit registers and expands the number of registers to 32. Leveraging this capability, we apply *Register Allocation* and *Vectorized Parallel Execution* strategies to PQMagic. First, we maximized the advantages of AVX512 by preloading all coefficients required for NTT operations into the registers, thereby reducing the risk of memory exposure and eliminating the performance overhead caused by repeated memory access. Specifically, in the ML-KEM algorithm, a polynomial consists of 256 coefficients of 12 bits each, which can be entirely stored in just 8 AVX512 registers. Similarly, in the ML-DSA algorithm, a polynomial contains 256 coefficients of 23 bits, requiring only 16 AVX512 registers for storage.

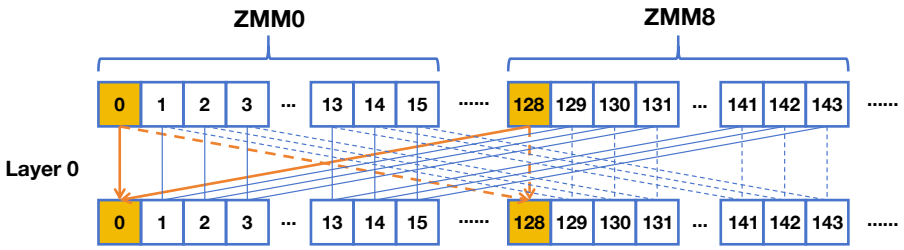


Fig. 5: NTT Layer 0 For ML-DSA

By consolidating all coefficient data into registers, this approach improves data locality and cache hit rates. As a result, by leveraging the data storage capabilities of AVX512 registers, the optimized NTT/INTT operations require only a single memory read for each coefficient and each root value, and a single memory write to store the results after computation. All other computations are performed entirely within the registers, eliminating additional memory access. Therefore PQMagic could minimize the risk of memory data exposure and improve the execution efficiency.

In addition to enhancing data parallelism, we further applied *Secure and Efficient Pipelining* strategy to directly use assembly code, optimizing instruction scheduling and improving ILP. As shown in Figure 5, the first layer of NTT operations in ML-DSA reveals that only two coefficient registers (e.g., ZMM0 and ZMM8) exhibit data dependencies during execution, while the remaining registers operate independently. Leveraging this feature, we could interleave operations of different coefficient registers, fully utilizing the parallel capabilities

of the instruction pipeline and the out-of-order execution capabilities of modern processors.

Instruction	CPI (Intel CPU)	CPI (AMD CPU)
vmovdqa32	0.5	0.5
vpaddw/vpsubw	0.5	0.25
vpaddd/vpsubd	0.5	0.25
vpmullw/vpmulhw	0.5/1.0	0.5
vpmuldq	0.5/1.0	0.5

Table 1: Instruction CPI on Different CPU platforms.

Furthermore, we carefully considered the features of modern superscalar processors, aiming to utilize multiple execution units. By referencing [3,13], we analyzed the number of parallel execution units for vector instructions in modern CPUs. Table 1 lists the CPI (Cycles Per Instruction) values for key instructions frequently used in NTT operations across common CPU platforms. CPI reflects the average number of cycles required to execute the instruction. It also indirectly indicates the number of parallel execution units. Based on this data, we further optimized the instruction scheduling strategy. When the CPI value is small, it indicates more parallel execution units. In this case, we carefully arranged identical instructions together to fully utilize hardware resources. When the CPI value is large, we filled gaps between such instructions with other operations to prevent pipeline stalls. This strategy maximizes ILP, while also reducing time-window for adversaries. Besides, using assembly code directly allows us to avoid compiler-induced execution variations that could lead to security issues [16] naturally.

4.2 Hash Components

The SHAKE hash function serves as a core cryptographic primitive in both ML-KEM and ML-DSA, and its computational efficiency and security directly impact the overall performance and robustness of these algorithms. We thus applied *Vectorized Parallel Execution* strategies for optimizing the Keccak-1600 structure in SHAKE, focusing on enhancing parallel computation while mitigating timing-based side-channel risks.

By leveraging the 512-bit wide vector registers provided by AVX512, we can enable parallel processing of up to 8 independent hash operations within a single instruction cycle. The degree of parallelism is doubled compared to the AVX2 version, increasing from 4-way to 8-way. This optimization not only increases throughput but also reduces the total number of instructions required for the same amount hash operation. Fewer instructions mean a shorter execution time, which naturally narrows the time window available for adversaries to observe and

exploit timing variations, thereby reducing the risk of timing-based side-channel attacks.

Additionally, our strategy helps to minimize data loading and storage latency by processing multiple data elements simultaneously within registers, reducing the frequency of memory accesses when doing hash operations, further mitigating the risk of cache-based side-channel attacks.

4.3 Matrix Expansion

In the implementation of matrix expansion, we applied *Branch Optimization* to eliminate complex conditional branches introduced by switch-case structures. The optimized implementation demonstrates superior programming practices by employing uniform execution paths across all operations. This design choice not only enhances code reliability but also eliminates performance fluctuations caused by conditional branch mispredictions.

By unrolling the switch-case operations, we achieve both more efficient compiler optimization potential and better parallel processing capabilities. Specifically, through branch optimization, we consolidate all polynomial sampling operations into a single unified function. This integrated approach processes the entire matrix by grouping polynomials in sets of eight, fully leveraging the parallel processing capabilities of AVX512 instructions. The implementation not only enhances execution consistency and stability, but also significantly reduces function call overhead. Therefore, this approach achieves improved performance for matrix expansion while showing a better coding paradigm that inherently reduces potential side-channel risks across all execution scenarios.

5 Evaluation

In this section, we conducted a comprehensive evaluation of the strategies implemented in PQMagic, focusing on security risk mitigation, performance improvements and practical applicability with a case study.

5.1 Experimental Setup

We evaluated PQMagic on an x64 platform featuring an AMD Ryzen 5 9600X processor and 256 GB RAM, with a Debian 12 OS. For our comparative evaluation, we conducted a comprehensive survey of existing implementations and identified: two state-of-the-art open-source libraries (pq-crystals [7,10] and liboqs [19]), as well as two recent works employing AVX-512 optimizations [20,21]. To ensure both reproducibility and fair comparison, we ultimately selected implementations with publicly available source code as our benchmark targets, i.e., the pq-crystals and liboqs libraries. From these libraries, we derived three datasets for evaluation:

- **pq-crystals-portable**: The baseline implementation from pq-crystals, representing an unoptimized version of ML-KEM and ML-DSA. Note that the commit versions we selected for these two algorithms are *4768bd37* and *444cdcc8* respectively.
- **pq-crystals-opt**: An optimized version from pq-crystals, leveraging AVX2 instructions to enhance performance. The commit versions for both ML-KEM and ML-DSA are the same as **pq-crystals-portable**.
- **liboqs**: A further refined version from liboqs, which integrates and encapsulates pq-crystals' implementations while providing additional component-level adjustments. The selected commit version is *5450d7c2*.

To maximize the performance of the comparison targets in our dataset, we compiled each project with the highest optimization level (*-O3*) and enabled architecture-specific optimizations by specifying *-march=native*. This compilation setting allows the compiler to fully utilize all available CPU instruction sets (including AVX-512) for optimal code generation.

	pq-crystals-portable	pq-crystals-opt	liboqs	PQMagic
ML-DSA-44	2	1	1	1
ML-DSA-65	2	9	9	1
ML-DSA-87	2	11	11	1

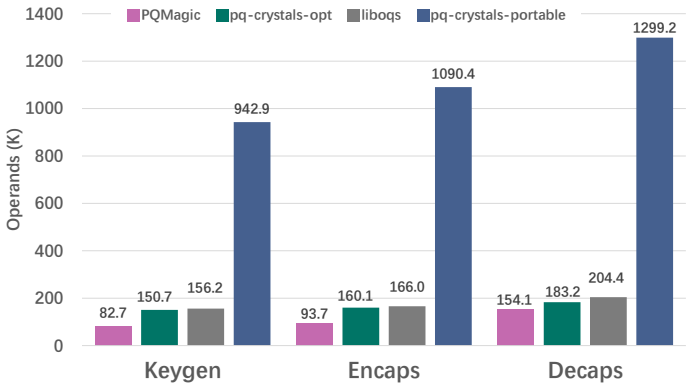
Table 2: Comparison of Branch Operation Numbers in Matrix Expansion Function Across Libraries.

5.2 Evaluation of Security Strategies in PQMagic

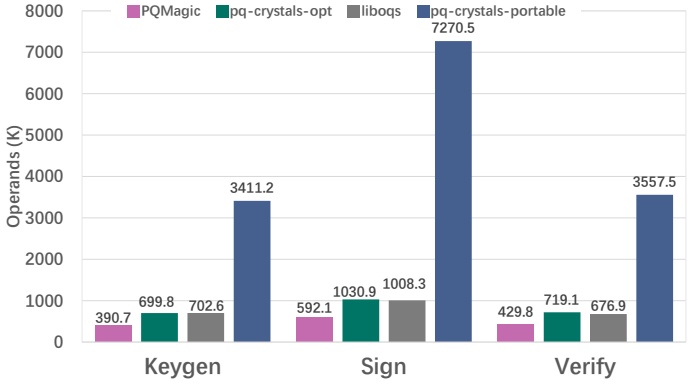
To validate the effectiveness of our optimization strategies, we designed to compare PQMagic's branch number in matrix expansion operation, total instruction number and memory operations against three implementations in our dataset.

Branch Reduction in Matrix Expansion First, we compared the number of branches in the matrix expansion operation across all implementations. All code was compiled with the highest optimization flags (*-O3, -march=native*), and the resulting binaries were comprehensively reverse-engineered to count the branches in the matrix expansion operation. The results are shown in Table 2. PQMagic achieved the fewest branches: for ML-DSA-87 parameter set, the highest security level (L5), the branch count was reduced from 11 in both pq-crystals-opt and liboqs to just 1 in PQMagic. This reduction is attributed to the elimination of nested loops in the sampling process, which typically generate two branches. For the ML-DSA-65 parameter set, PQMagic reduced the branch

count from 9 in pq-crystals-opt and liboqs to 1. For the ML-DSA-44 parameter set, compiler optimizations result in pq-crystals-opt and liboqs achieving a branch count of 1, matching PQMagic's performance in this regard. In contrast, the unoptimized pq-crystals-portable implementation retained 2 branches due to its nested loop structure. These results demonstrate that PQMagic's branch optimization strategy successfully minimizes branch counts, reducing the risk of timing side-channel attacks and improving execution stability by mitigating branch misprediction penalties.



(a) Instruction Count of ML-KEM-1024



(b) Instruction Count of ML-DSA-87

Fig. 6: Compare the number of instructions of PQMagic's implementation of ML-KEM and ML-DSA under the highest security parameters with other open source library implementations

Instruction Count and Memory Access Reduction Then, we compared the number of instructions and memory access operations in PQMagic against the other implementations. To ensure fairness in testing, we fixed identical input values for all test cases, including random numbers, signed messages, and signature contexts. This design guarantees that each implementation performs the same operation routines.

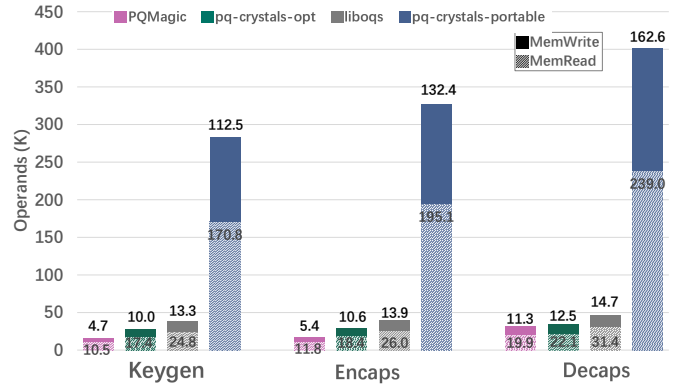
Figure 6 and Figure 7 shows the results for ML-KEM and ML-DSA at their highest security levels (L5). Compared to the performance-optimized pq-crystals-opt implementation, PQMagic reduced the instruction count for ML-KEM's Keygen, Encaps, and Decaps operations by 45.1%, 41.5%, and 15.9%, respectively, while memory access overhead was reduced by 44.5%, 40.7% and 9.8%. Compared to liboqs, PQMagic achieved reductions of 47.1%, 43.6%, and 24.6% in instruction count and 60.1%, 56.9%, and 32.3% in memory access overhead for the same operations. Against the unoptimized pq-crystals-portable implementation, PQMagic reduced instruction counts by 91.2%, 91.4%, and 88.1%, and memory access overhead by 94.6%, 94.7%, and 92.2% for Keygen, Encaps, and Decaps respectively.

For ML-DSA, PQMagic outperformed pq-crystals-opt by reducing instruction counts for Keygen, Sign, and Verify operations by 44.2%, 42.6%, and 40.2%, respectively, and memory access overhead by 61.6%, 51.2%, and 64.5%. Compared to liboqs, PQMagic achieved reductions of 44.4%, 41.3%, and 36.5% in instruction count and 50.8%, 38.2%, and 50.9% in memory access overhead for Keygen, Sign, and Verify respectively. Against the unoptimized pq-crystals-portable implementation, PQMagic reduced instruction counts by 88.5%, 91.9%, and 87.9%, and memory access overhead by 90.0%, 92.7%, and 91.3% for each operation.

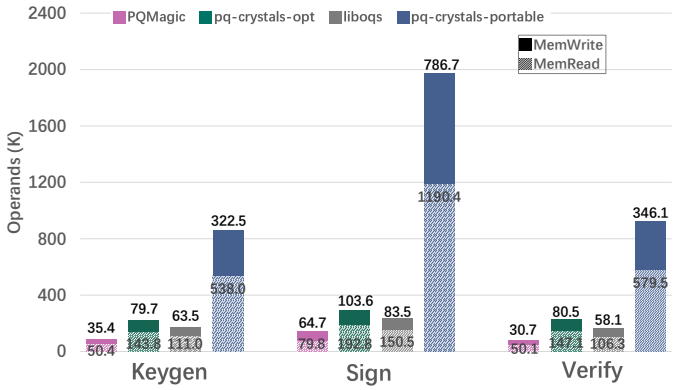
These results highlight that we lower the risk of data leakage through memory access patterns and minimize the time window available for adversarial observation. As a result, these strategies raise the bar for potential attackers, making it more challenging to exploit side-channel vulnerabilities.

5.3 Performance Analysis of ML-KEM Implementation

In the performance evaluation of the ML-KEM algorithm, we adopt the same testing methodology as described in Section 5.4. As illustrated in Figure 8, PQMagic achieves significant performance improvements compared to other implementations. Against pq-crystals, PQMagic demonstrates speedups of 1.27x, 1.24x, and 1.19x in Keygen, Encaps, and Decaps operations, respectively. Compared to liboqs, PQMagic increase the performance by 1.77x, 1.79x, and 1.52x for the same operations. When compared to the unoptimized pq-crystals-portable implementation, PQMagic delivers even more substantial gains, with performance improvements of 8.05x, 7.83x, and 8.53x in Keygen, Encaps, and Decaps, respectively.



(a) Memory Operands of ML-KEM-1024



(b) Memory Operands of ML-DSA-87

Fig. 7: Compare the number of memory operations of PQMagic's implementation of ML-KEM and ML-DSA under the highest security parameters with other open source library implementations

5.4 Performance Analysis of ML-DSA Implementation

In this section, we conduct a performance evaluation of the PQMagic ML-DSA implementation. We ran each benchmark binary within 3 seconds, recording the number of algorithm executions for each test. The results of the ML-DSA comparison with the highest level of security parameters [9] are shown in Figure 8. The results demonstrate that PQMagic significantly outperforms other implementations. Compared to pq-crystals, PQMagic achieves performance improvements of approximately 1.90x in Keygen, 1.74x in Sign, and 1.83x in Verify. When compared to liboqs, PQMagic shows even greater gains, with speedups of 2.24x, 1.89x, and 2.04x for Keygen, Sign, and Verify, respectively. Against the

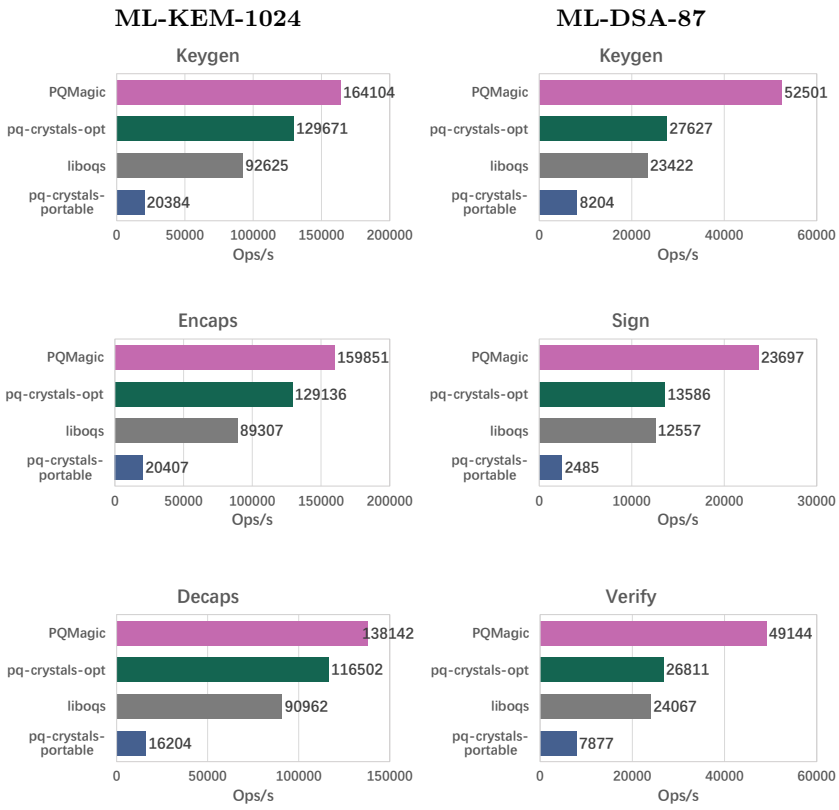


Fig. 8: Comparison of operations per second for PQMagic implementation of ML-KEM and ML-DSA at highest security parameters with other open source library implementations.

unoptimized pq-crystals-portable implementation, PQMagic delivers remarkable improvements of 6.40x, 9.54x, and 6.24x for the same operations.

5.5 Case Study

Post-quantum cryptographic algorithms are inherently more complex than traditional cryptographic algorithms, inevitably introducing higher computational and storage overhead, which poses significant challenges for post-quantum cryptographic migration. To evaluate the contributions of PQMagic to post-quantum cryptographic migration, we designed the following experiments to assess the performance of the migrated algorithms.

We selected the widely used TLS 1.3 protocol to evaluate the impact of cryptographic algorithms on handshake performance and designed a simplified model for establishing secure connections. Specifically, this model retains only

the core processes of key exchange and authentication while excluding external interference factors such as network latency and packet loss.

In our experiments, we simulated key exchange and authentication workflows using post-quantum cryptographic algorithms (ML-DSA + ML-KEM) and traditional algorithms (RSA-2048/ECDSA-256 + ECDH), conducting a comparative analysis between PQMagic implementations at L1/L3/L5 security levels and traditional algorithm implementations from the state-of-the-art library OpenSSL [15]. Note that, ML-DSA-44 is at L2 security level, while ML-KEM-512 only reaches L1. So, we treated ML-DSA-44 + ML-KEM-512 as the minimal security level of its components, i.e., L1. As shown in Figure 9, the results demonstrate that PQMagic outperforms the RSA-2048 + ECDH combination across all security levels. At the L1 security level, PQMagic achieves a 4.77x efficiency improvement over RSA-2048 + ECDH. At the L3 and L5 security levels, the improvements are 3.25x and 2.67x, respectively. When compared to the more efficient ECDSA-256+ECDH combination, PQMagic still shows superior performance at the L1 and L3 security levels, with efficiency improvements of 1.58x and 1.07x, respectively. At the highest L5 security level, PQMagic's efficiency is slightly lower, at 0.88x compared to ECDSA-256 + ECDH.

Given that the L3 security level already meets NIST's recommended requirements and the performance gap at L5 is only 9 microseconds, these results highlight PQMagic's strong practicality and its potential to facilitate the transition to post-quantum cryptography.

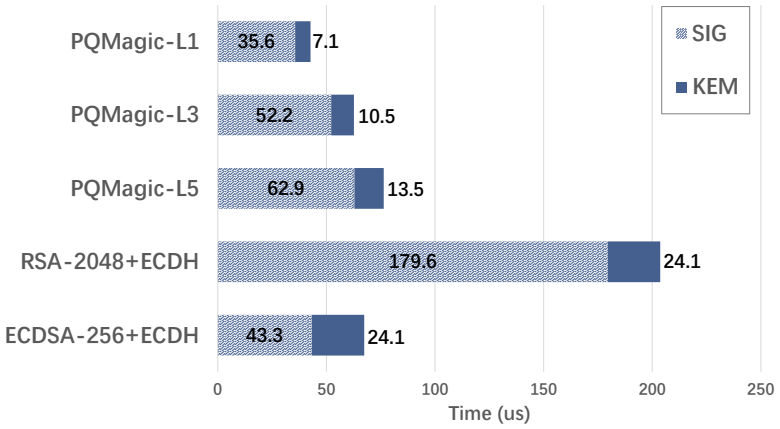


Fig. 9: The TLS 1.3 handshake latency of PQMagic post-quantum algorithm compared to traditional cryptographic algorithms.

6 Related Work

The optimization of post-quantum cryptographic algorithms has been widely studied. Gueron et al. [11] proposed a vectorized sampling step for accelerating lattice-based key exchange, laying the groundwork for efficient implementations of post-quantum primitives. Seiler [17] introduced an AVX2-optimized implementation of the Number Theoretic Transform (NTT) for ring-LWE-based cryptography, which became a cornerstone for subsequent optimizations. These techniques were further adopted and extended in the pq-crystals library, which implemented the Kyber [7] and Dilithium [10] algorithms (later standardized as ML-KEM [5] and ML-DSA [9] by NIST) using AVX2 instructions for core operations.

Recent advancements have focused on leveraging the AVX512 instruction set for further optimizations. Lei et al. [14] proposed parallel polynomial sampling and arithmetic operations using AVX512, improving the efficiency of ML-DSA. Zheng et al. [21] introduced PSPM-TEE, an optimized vectorization approach for polynomial multiplication for ML-DSA. Besides that, Zheng et al. [20] also optimize ML-KEM using AVX512, integrating it into the TLS 1.3 protocol. However, these works did not fully exploit the capabilities of modern superscalar CPU pipelines or discuss potential security risks in their implementations.

7 Conclusion

In this paper, we first identify several security issues in existing PQC implementations, which also fail to fully leverage the resource advantages of modern CPUs to enhance performance. To address these issues, we propose four optimization strategies that reduce branch counts, instruction counts, and memory access operations. Based on these optimization strategies, we implement PQMagic, demonstrating that it not only achieves lower security risks compared to state-of-the-art implementations but also delivers superior performance. Notably, PQMagic even surpasses the execution efficiency of traditional cryptographic algorithms, showcasing its strong practicality and providing robust support for the transition to post-quantum cryptography.

While our optimization approach delivers significant performance gains, it has inherent limitations regarding portability and maintainability. The AVX-512 dependency limits portability to other architectures, and assembly codes reduce code readability. These trade-offs make our approach best suited for performance-critical scenarios with fixed hardware targets.

Acknowledgments. The authors would like to thank the reviewers for their valuable feedback. This work was partially supported by Shanghai Science and Technology Innovation Action Plan Special Project (23511100900).

References

1. Abdulrahman, A., Becker, H., Kannwischer, M.J., Klein, F.: Fast and clean: Auditable high-performance assembly via constraint solving. IACR Transactions on Cryptographic Hardware and Embedded Systems **2024**(1), 87–132 (2024)
2. Abdulrahman, A., Kannwischer, M.J., Lim, T.H.: Enabling microarchitectural agility: Taking ml-kem & ml-dsa from cortex-m4 to m7 with slothy. Cryptology ePrint Archive (2025)
3. Abel, A., Reineke, J.: Uops info. `http://www.uops.info`, last accessed 2025/03/13
4. Abel, A., Reineke, J.: uops. info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 673–686 (2019)
5. Alagic, G., Dang, Q., Moody, D., Robinson, A., Silberg, H., Smith-Tone, D., et al.: Module-lattice-based key-encapsulation mechanism standard (2024)
6. Blessing, J., Specter, M.A., Weitzner, D.J.: Cryptography in the wild: An empirical analysis of vulnerabilities in cryptographic libraries. In: Proceedings of the 19th ACM Asia Conference on Computer and Communications Security. pp. 605–620 (2024)
7. Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-kyber: a cca-secure module-lattice-based kem. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 353–367. IEEE (2018)
8. Cohen, B., Subramony, M., Clark, M.: Next generation "zen 5" core. In: 2024 IEEE Hot Chips 36 Symposium (HCS). pp. 1–27. IEEE (2024)
9. Dang, T., Lichtinger, J., Liu, Y.K., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., et al.: Module-lattice-based digital signature standard (2024)
10. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-dilithium: A lattice-based digital signature scheme. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 238–268 (2018)
11. Gueron, S., Schlieker, F.: Speeding up r-lwe post-quantum key exchange. In: Nordic conference on secure IT systems. pp. 187–198. Springer (2016)
12. Intel: Intel 64 and ia-32 architectures optimization reference manual. `https://www.intel.com/content/www/us/en/content-details/821612/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html`, last accessed 2025/03/13
13. Intel: Intel intrinsics guide. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`, last accessed 2025/03/13
14. Lei, D., He, D., Peng, C., Luo, M., Liu, Z., Huang, X.: Faster implementation of ideal lattice-based cryptography using avx512. ACM Transactions on Embedded Computing Systems **22**(5), 1–18 (2023)
15. OpenSSL: Openssl. `https://github.com/openssl/openssl`, last accessed 2025/03/13
16. Schneider, M., Lain, D., Puddu, I., Dutly, N., Capkun, S.: Breaking bad: How compilers break constant-time˜ implementations. arXiv preprint arXiv:2410.13489 (2024)
17. Seiler, G.: Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography. Cryptology ePrint Archive (2018)

18. Singh, T., Oliver, S., Rangarajan, S., Southard, S., Henrion, C., Schaefer, A., Johnson, B., Tower, S.B., Hoover, K., John, D., et al.: "zen 5": The amd high-performance 4nm x86-64 microprocessor core. In: 2025 IEEE International Solid-State Circuits Conference (ISSCC). vol. 68, pp. 1–3. IEEE (2025)
19. Stebila, D., Mosca, M.: Post-quantum key exchange for the internet and the open quantum safe project. In: Avanzi, R., Heys, H. (eds.) Selected Areas in Cryptography (SAC) 2016. Lecture Notes in Computer Science, vol. 10532, pp. 1–24. Springer (October 2017), `https://openquantumsafe.org`
20. Zheng, J., Zhu, H., Dong, Y., Song, Z., Zhang, Z., Yang, Y., Zhao, Y.: Faster post-quantum tls 1.3 based on ml-kem: Implementation and assessment. In: European Symposium on Research in Computer Security. pp. 123–143. Springer (2024)
21. Zheng, J., Zhu, H., Song, Z., Wang, Z., Zhao, Y.: Optimized vectorization implementation of crystals-dilithium. arXiv preprint arXiv:2306.01989 (2023)