KpqBench: Performance and Implementation Security Analysis of KpqC Competition Round 1 Candidates

YongRyeol Choi[1], MinGi Kim[2], YoungBeom Kim[3], JinGyo Song[4], JaeHwan Jin[5], HeeSeok Kim*[6], and Seog Chung Seo*[7]

[1] Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul, South Korea mike0726@kookmin.ac.kr
[2] Department of Information Security, Cryptology, and Mathematics, Kookmin University, Seoul, South Korea mingi1684@kookmin.ac.kr
[3] Department of Financial Information Security, Kookmin University, Seoul, South Korea darania@kookmin.ac.kr
[4] Wired Network Development Team, LG U+, Seoul 07795, South Korea jingyosong@lguplus.co.kr
[5] Wired Network Development Team, LG U+, Seoul 07795, South Korea daenamu1@lguplus.co.kr
[6] Department of Cyber Security, Korea University, Sejong 30019, South Korea 80khs@korea.co.kr
[7] Department of Financial Information Security, Kookmin University, Seoul, South Korea scseo@kookmin.ac.kr

Abstract. As the global migration to post-quantum cryptography (PQC) continues to progress actively, in Korea, the Post-Quantum Cryptography Research Center has been established to acquire PQC technology, leading the KpqC Competition. In February 2022, the KpqC Competition issued a call for proposals for PQC algorithms. By November 2022, 16 candidates were selected for the first round (7 KEMs and 9 DSAs). Currently, Round 1 submissions are being evaluated with respect to security, efficiency, and scalability in various environments. At the current stage, evaluating the software through an analysis to improve the software quality of the first-round submissions is judged appropriately. In this paper, we present analysis results regarding performance and implementation security on based dependency-free approach of external libraries. Namely, we configure extensive tests for an analysis with no dependencies by replacing external libraries that can complicate the build process with hard coding. From the performance perspective, we provide analysis results of performance profiling, execution time, and memory usage for each of the KpqC candidates. From the implementation security perspective, we examine bugs and errors in the actual implementations using Valgrind software, a metamorphic testing methodology that can include wide test coverage and constant-time implementation against the timing attack. Until the KpqC standard algorithm is announced, we argue that continuous integration of extensive tests will lead to higher-level software quality of KpqC candidates.

Keywords: Post-Quantum Cryptosystems · Benchmarking· Constant-time· Cryptographic Library· KpqC Competition· Metamorphic Testing· Software Validation

1 Introduction

With the development of quantum computer technology, the importance of post-quantum cryptography (PQC), which can replace public key cryptography in a quantum environment, is being stressed. In 2016, the National Institute of Standards and Technology (NIST) announced the PQC Competition through an open competition such as AES and SHA3. Currently, four standardization candidates for Key Encapsulation Mechanism (KEM) and Digital Signature Algorithm (DSA) have been selected, and NIST is conducting a competition for additional standardization proposals for digital signature algorithms. Unlike AES and SHA-3, the mathematical complexity of PQC algorithms has increased, and they support a wide range of key lengths and signature lengths, ranging from bytes to megabytes. In addition, the performance of each algorithm varies, and some algorithms are too slow or faster than traditional public-key cryptography. The various features of the PQC algorithms have made the selection of candidates at NIST more difficult and task burdens the community, research institutes, and industry in reviewing and evaluating candidates. In other words, this meant that candidates had to be integrated and tested on various platforms and situations, and the importance of software was emphasized here. In these difficulties, various community and industry projects contributed to selecting NIST PQC Competition candidates. These include PQClean[1], which integrates reference C implementations in Intel, pqm4[2] for clean implementations on ARM Cortex-M4 platforms; OQS OpenSSL[3], StrongSwan[4] integrating reference implementations into security protocols and applications. Based on this research, the implementation suitability of the submitted software was evaluated on various platforms.

To enhance domestic PQC capabilities, a Korea Post-Quantum Cryptography (KpqC) Competition was held. The KpqC Competition requested to submit a development roadmap until February 2022 Considering domestic technology, and based on this, 18 candidates for Round 0 candidates were selected. Through the round 1 of public evaluation, 16 candidates have been selected, and the selection of algorithms for standardization will be completed by September 2024. Afterward, it is expected that the selected algorithm will be standardized for PQC and included in the algorithms subject to KCMVP verification. The most important thing aspect of this is the software implementation. Some candidates generate errors and warnings when building, and it takes time and effort repeatedly to get them to compile and run cleanly. Unlike the NIST PQC Competition, where the suitability and security of software are evaluated by participating worldwide, there are limitations in software evaluation for KpqC Competition candidates. Thus, to strengthen the software quality of domestic candidates, it is necessary to verify candidates with an extensive test framework through continuous integration such as PQClean[1] and pqm4[2].

As with NIST PQC Competition candidates, the KpqC Competition required candidates to be developed by an ANSI C software implementation, which also had to be accompanied by a KAT (Known Answer Test) value to verify implementation correctness. However, these candidates are limited in verifying the correctness of the implementation of the software. First, the KAT test checks the output value for a given input value, and it is challenging to find bugs in the software implementation. For this reason, a new software verification method called metamorphic testing, which can verify wide test coverage, has recently been proposed [5,6]. In IEEE/ACM MET WorkShop'19 [5], several bugs in implementation were found by applying metamorphic testing to the initial NIST PQC Competition candidates, and in IEEE'18 [6], vulnerabilities in the hash function that passed CAVP verification were found by applying metamorphic testing. Thus, the application of metamorphic testing enables the implementation vulnerabilities of in a wide range of software. In addition, the second limitation is that there is no verification of the constant-time implementation of the candidates. The most important aspect of the NIST PQC Competition from the second round is the countermeasure against side-channel attacks. Although PQC may be mathematically secure in a quantum computing environment, it is still vulnerable to side-channel attacks depending on how it is implemented. Furthermore, there are many studies to achieve constant-time implementation of PQC [7,8]. For this reason, constant-time implementation is important for resisting side-channel attacks of PQC.

In this paper, we present the evaluation for KpqC round 1 candidates considering performance and implementation security on an integration basis through extensive tests. From a performance perspective, we examine the execution time, memory usage, and profiling results for KpqC competition candidates. From an implementation security perspective, we validate KpqC competition candidates extensively, from Valgrind validation to detect basic implementation errors to metamorphic testing with wide test coverage. Furthermore, side-channel vulnerabilities are analyzed by checking whether they implement constant time through Valgrind. Finally, we have removed dependencies on external libraries that can complicate builds. We have put the extensive test of the KpqC Competition for continuous integration into the public domain and made it available at **https://github.com/kpqclib/kpqclib**. The goal of this paper is to develop a continuous integration test by collecting independent C implementations of KpqC competition candidates in various environments, such as PQClean[1] and pqm4[2]. We expect to contribute to improving the software quality of the KpqC Competition.

1.1 Contributions

- **Design of an extensive test to improve software quality of the KpqC Competition candidates**
  Recently, four algorithms have been selected as candidates for standardization in the NIST PQC Competition. Various community and industry

projects contributed to the evaluation of NIST by reviewing the software verification and applicability in various environments for submitted software. In Korea, there is an ongoing KpqC competition led by the Post Quantum Cryptography Research Center. However, verifying the software of the KpqC competition candidates with the same level of rigor as the NIST PQC Competition is challenging. The key factor for migrating the public key cryptography to PQC is the extensive verification of software. Thus, we present the continuous integration test for the KpqC Competition round 1 candidate considering performance and implementation security. The source code for our extensive test is publicly available on **https://github.com/Unlimitosu/KPQClib**. We aspire that through verification of our continuous integration test, the quality of the software of KpqC Competition candidates will be improved.

- **Providing the analyses of three performance of KpqC competition candidates: benchmark, memory consumption, and profiling**
  We present the results of a detailed performance analysis of Round 1 KpqC submissions from three perspectives: benchmark, memory consumption, and profiling. Benchmarks submitted by round 1 candidates perform reliably. However, it is difficult to expect fair results because the benchmarking environment is not the same. For this reason, we provide benchmarking results for KpqC round 1 candidates in a typical CPU environment (Intel) for fair comparison. In addition, we evaluate the applicability of KpqC competition candidates in resource-constrained devices by closely comparing memory usage. Some algorithms may be difficult to apply in resource-constrained devices due to excessive key and signature lengths, so their applicability in resource-constrained devices should be carefully analyzed. Finally, we present profiling results for KpqC round 1 candidates. Profiling may provide a bottleneck point for each KpqC submission, suggesting the direction of optimization research.

- **Verifying implementation security of KpqC Competition candidates through extensive tests**
  We evaluate the implementation security of the KpqC round 1 candidates through extensive tests. For basically extensive test design, we have replaced working dependencies in external libraries that can complicate builds with hard coding. Implementation security can be categorized into implementation errors and side-channel vulnerabilities. In implementation errors, verifying existing candidates was generally performed with the KAT test, but it is limited to finding bugs and errors. Thus, we verify them through metamorphic testing, which can cover a wide range of basic verification through the Valgrind tool to find bugs and errors. As a result, simple errors were found in two candidates, and metamorphic error was found in one candidate. In addition, we present an implementation method to solve this error because a metamorphic error means that the same master key value can be generated for forged inputs. In side-channel vulnerability analysis, we identify the constant-time implementation, which is side-channel resistant, for

each KpqC candidate via the Valgrind tool. Most of the KpqC Competition candidates achieved constant-time implementation, and only one candidate failed to satisfy it.

1.2 Differences from the existing KPQClean project

In Cryptology ePrint Archive'23[9], a KPQClean, which integrates KpqC Competition candidates and provides benchmarking results, has been proposed. They presented benchmarking results in two general CPU environments (Intel and Ryzen) for KpqC round 1 candidates. The -O2 and -O3 options showed detailed performance comparison results. In addition, accessibility was enhanced by removing external library elements used in many of the KpqC Competition candidates and replacing them with hard coding. In this paper, we provide benchmarking results in only one environment (Intel) but suggest the direction of applicability of the KpqC algorithms in resource-constrained devices by comparing the memory usage. Furthermore, extending the applicability by removing the working dependency (external library) is the same as the paper above[9]. Additionally, we verify KpqC Competition candidates by applying metamorphic testing, and a wide range of software verification techniques. Finally, side-channel resistance is evaluated by checking whether constant-time is implemented through Valgrind. The detailed differences between KPQClean[9] and our test framework are shown in Table 1.

Test	**KPQClean[9]**	**KPQCLib(Our Work)**
Benchmark Results	Ryzen 7 4800, Intel i5-8259U → 2 Environment	Intel i7-13700K → 1 Environment
Working Dependencies	Eliminating the external library → OpenSSL	Eliminating the external library → OpenSSL
Memory Usage	-	Providing Stack Usage for KpqC round 1 candidates
Algorithm Profiling	-	Finding performance bottleneck of KpqC round 1 candidates
Software Engineering	-	Providing testing framework through Valgrind test and metamorphic testing
Side Channel Resistance	-	Providing results on whether it is a constant-time implementation

Table 1: Difference between testing framework of KPQClean[9] and Our Work

The rest of the paper is summarized as follows. Section 2 provides a detailed description of the KpqC Competition, metamorphic test, memory leakage, and

Constant-time implementation verification through Valgrind, and the definition of bugs and errors found. Section 3 presents related work for software verification and side-channel vulnerabilities in PQC. Section 4.1 presents benchmarking results, Valgrind testing results, and metamorphic testing results for KpqC round 1 candidates. Finally, Section 5 gives the conclusions of this paper and future work.

2 Preliminaries

2.1 KpqC Competition Overview

Globally, there is active research and development of PQC to achieve security in the era of quantum computing. In Korea, efforts are being made to secure domestic PQC technologies and promote policies applicable to the domestic landscape. To pursue this goal, the KpqC (Korea Post-Quantum Cryptography) Research Group has been organizing the KpqC Competition since 2021. Unlike the NIST PQC Competition, the KpqC Competition started from Round 0, considering domestic PQC-related capabilities.

In Round 0, participants were required to submit development proposals, including algorithm functionality, underlying issues, design approach, and development plans. Afterward, they were granted approximately 9 months for the development phase. The submissions were evaluated based on criteria such as algorithm excellence and originality, soundness of security rationale, specificity, and feasibility of development objectives, leading to the selection of Round 0 candidates. In total, 18 algorithms were chosen, comprising 8 KEMs and 10 DSAs. Round 1 in KpqC Competition is currently in progress, and KpqC Competition plans to select Round 2 candidates in December 2023. Round 2 will be conducted from February to September 2024. The final algorithm selection is scheduled for September 2024. KpqC Competition is considering conducting Round 3 to prepare the necessary elements for implementing the algorithms selected in Round 2. The classification of Round 1 candidates is shown in Table 2 and 3.

A summary of each algorithm is following:

PKE/KEM Schemes

- **IPCC** IPCC (Improved Perfect Code Cryptosystem) is a graph-based PQC that relies on the security of the Perfect Code Cryptosystem (PCC) problem, which determines the existence of Perfect Domination Sets (PDS) in a graph. The PCC problem has been proven to be NP-Complete, and generally, PCC-based encryption schemes struggle to provide both security and efficiency. However, IPCC is designed to operate efficiently for the same k by utilizing multiple graphs. The key generation and decryption processes follow the design principles of traditional PCC-based encryption schemes. IPCC is built upon the problem of determining PDS in 3-regular graphs, where a 3-regular graph refers to a graph in which all vertices have a degree

Based Problem	Algorithm	Total
Lattice	NTRU+ SMAUG TiGER	3
Code	Layered ROLLO-I PALOMA REDOG	3
Graph	IPCC	1

Table 2: PKE/KEM Algorithm Classification of KpqC Round 1 Candidates

Based Problem	Algorithm	Total
Lattice	GCKSign HAETAE NCCSign Peregrine SOLMAE	5
Code	Enhanced pqsigRM	1
Polynomial	MQSign	1
Isogeny	FIBS	1
Zero-knowledge Proof	AIMer	1

Table 3: DSA Algorithm Classification of KpqC Round 1 Candidates

of 3. PDS represents a subset D of the set of all vertices V, where for each vertex, the set of connected vertices is a subset of D. The key generation function in IPCC takes the cardinality of the sets of vertices in each graph as input and generates the public and private keys accordingly. The private key in IPCC is represented by the Perfect Domination Function (PDF), and the public key consists of sets of graphs. PDF is a function that maps sets of vertices in a graph to 0 or 1, under the condition that the sum of connected vertices for each vertex is 1. The encryption in IPCC involves generating an invariant polynomial f_G^k corresponding to the vertex set G and the maximum degree k. This polynomial is then used for encryption. Decryption is performed by applying the PDF function to each element of the ciphertext. The sizes of IPCC's public key, private key, and ciphertext are 4,800 bytes, 400 bytes, and 92,000 bytes, respectively, for a security strength of 80 bits.

– **Layered ROLLO-I** Layered ROLLO-I is a code-based PQC scheme based on the Rank Syndrome Decode problem which is NP-complete. ROLLO, which is built upon the Low-Rank Parity Check (LRPC) code, was submitted to the NIST PQC Competition. However, it exhibited lower decryption performance compared to lattice-based PQC. To alleviate these performance limitations, it was designed based on the ideal LRPC (BII-LRPC) code with a modified structure. By multiplying low-rank vectors into two polynomials, Layered ROLLO-I made it challenging for attackers to exploit the cryptosystem's structural properties, thereby enhancing the security strength. Layered ROLLO-I provides security levels of 128 (resp. 192 and 256) bits with 1,240 (resp. 1,699 and 2,571) bytes of the public key, 120 (resp. 120 and 120) bytes of the private key, and 620 (resp. 850 and 1,286) bytes of ciphertext.

– **NTRU+** NTRU+ is a scheme that shares a similar design rationale with NTRU, which was a Round 3 candidate in the NIST PQC Competition. The security of NTRU+ is based on the RLWE problem in the NTRU lattice. One of the distinguishing features of NTRU+ is the proposal of the Semi-generalized One Time Pad (SOTP), an encoding method that ensures the upper bound of the decryption error rate becomes negligible when an attacker randomly selects the random values used in encryption. SOTP operates on a message $x \in \{0,1\}^n$ and another input value $u = (u_1, u_2) \in \{0,1\}^{2n}$, where $SOTP(x,u) = (x \oplus u_1) - u_2$. Additionally, NTRU+ uses the NTT-friendly ring $\mathbb{Z}_q[X]/(X^n - X^{n/2} + 1)$, where $n = 2^i 3^j$ and $q = 3457$, to perform all polynomial multiplications using NTT. The advantage of using NTT is that it reduces the time complexity of polynomial multiplication into $O(n \log n)$. Furthermore, unlike some other schemes, NTRU+ does not employ the Fujisaki-Okamoto transform[10] in the IND-CCA. Instead, it adopts a method of recovering and comparing the random values used in encryption during the decryption process. NTRU+ provides security levels of 576 (resp. 768, 864, and 1,152) bits with 864 (resp. 1,699, 1,296, and 1,728) bytes of the public key, 1728 (resp. 2,304, 2,592, and 3,456) bytes of the private key, and 864 (resp. 1,152, 1,296, and 1,728) bytes of ciphertext.

– **PALOMA** PALOMA is a PQC scheme based on Goppa codes, which are used in classic McEliece-like cryptosystems. It relies on the NP-hard Syndrome Decoding Problem and involves shuffling the parity-check matrix similar to the Niederreiter cryptosystem. However, unlike the Niederreiter cryptosystem, PALOMA skips the process of converting the message into a specific Hamming weight for decryption. This omission reduces the computational complexity of the encryption and decryption processes, resulting in improved performance. PALOMA used a binary separable Goppa code $C = [n, k, \geq 2t+1]_2$, defined by the support set L and the separable Goppa polynomial $g(X)$. The support set is derived from $[\alpha_0, \alpha_1, \cdots, \alpha 2^m - 1] \leftarrow$ SUFFLE (F_{2^m}), where $L \leftarrow [\alpha_0, \alpha_1, \cdots, \alpha_{n-1}]$, and $g(X) \leftarrow \prod_{j=n}^{n+t-1}(X - \alpha_j)$. Goppa code can be generated in constant time due to the reducibility and separability of $g(X)$ over $\mathbb{F}_{2^{13}}[X]$. PALOMA provides different sizes of public and private keys and ciphertexts based on the security level: 128 (resp. 192, and 256) bits. It has 319,448 (resp. 812,032, and 1,025,024) bytes of public keys, 94,496 (resp. 355,400, and 257,064) bytes of private keys, and 136 (resp. 240, and 240) bytes of ciphertexts.

– **SMAUG** SMAUG is a scheme based on Module-LWE/Module-LWR and shares a similar design rationale with RLizard and Lizard, which were submitted to the NIST PQC Competition. SMAUG employs a sparse secret key and predefines the Hamming weight of the secret key for Module-LWE/Module-LWR, allowing the extraction of a secret key that meets the specified conditions. Additionally, SMAUG sets the values of p and q as powers of 2 to replace rounding operations with bitwise shifts. It also avoids using NTT during the multiplication process. SMAUG provides different sizes of public and private keys and ciphertexts based on the security level: 1 (resp. 3, 5) bits. For each security level, it has 672 (resp. 992, and 1,632) bytes of public keys, 174(846) (resp. 185(1,177), and 182(1,814)) bytes of private keys, and 768 (resp. 1,024, and 1,536) bytes of ciphertexts.

– **TiGER** TiGER is a scheme based on the RLWE/RLWR in the spirit of RLizard and Lizard, which were submitted to the NIST PQC Competition. TiGER utilizes a sparse secret key and predefines the Hamming weight of the secret key to generate a secret key that meets certain conditions. Additionally, TiGER employs Error Correcting Codes during message encoding to recover from errors. This ensures that the decryption failure probability becomes negligible. The Error Correcting Code options used are D2[11] or XEf[12]. Another characteristic of TiGER is the small size of ciphertexts and public keys, which is due to the usage of a 1-byte value for $q = 256$ in TiGER. In practice, TiGER has ciphertexts and public keys of size 768 bytes and 480 bytes, respectively, for security level 1. Even at the highest security level of 5, the sizes are 1,152 bytes and 928 bytes, respectively.

DSA Schemes

– **AIMer** AIMer is a zero-knowledge proof-based DSA that utilizes the AIM (Arithmetic Inverse Mask) function, which is a one-way function from the MPCinH[13] zero-knowledge proof framework. MPCinH allows for the efficient design of virtual zero-knowledge proof protocols through Multi-Party Computation (MPC). AIM function is a tweakable one-way function that provides multi-target one-wayness. For an$(l+1)$-tuple $(e_1, e_2, \cdots, e_l, e_*)$ and input and output size n, AIM function AIM: $\mathbb{F}_{2^n} \times \mathbb{F}_{2^n} \to \mathbb{F}_{2^n}$ is defined as follows: $AIM(iv, pt) = Mer[e_*] \circ Lin[iv] \circ Mer[e_1, \cdots, e_l](pt) \oplus pt$. Here, Mer is defined as $Mer[e](x) = x^{2^e - 1}$ for $x \in \mathbb{F}_{2^n}$, providing non-linearity. Lin is a linear function composed of matrix multiplication over $n \times ln$ size matrix $A_{iv} = [A_{iv,1}| \cdots |A_{iv,l}] \in (\mathbb{F}_2^{n\times n})^l$ and vector addition with $b_{iv} \in \mathbb{F}_{2^n}$. AIMer uses the Fiat-Shamir Transformation[14] to convert the interactive zero-knowledge proof system based on the BN++ proof system[15]. Using the transformed system, digital signatures are generated based on the proof produced, relying on the one-wayness property of the AIM function. AIMer-I (resp. III, V) provides NIST security level 1 (resp. 3, 5) with 32 (resp. 48, 64) byte of public key, 16 (resp. 24, 32) byte of secret key, and 5,904 (resp. 13080, 25152) byte of signature. The signature size of AIMer is flexible according to the value of N and τ.

- **Enhanced pqsigRM** Enhanced pqsigRM is a code-based DSA that replaces Goppa code used in the original CFS (Courtois, Finiasz, and Sendrier) algorithm with a modified Reed-Muller code[16]. CFS algorithm suffers from inflexibility in parameter extension and lacks security against Existential Unforgeability under Chosen Message Attack (EUF-CMA). Additionally, the signature generation time in CFS relies on the factorial of the error correction range of the Goppa code, denoted as $t!$. To achieve faster signature generation, the value of t needs to be small, which, unfortunately, compromises security and requires increasing the key size for higher security strength. To mitigate these drawbacks, Enhanced pqsigRM adopts the modified Reed-Muller code. The Reed-Muller code offers fast computations and simple decoding, but it is susceptible to attackers leveraging the algorithm's structure to gain secret information. To address this issue, the generator matrix $G_{(r,m)}$ is modified by replacing, appending, and padding certain parts, resulting in the generation of the modified Reed-Muller code. The recursive definition of Reed-Muller code is given as follows: $RM_{(r,m)} = \{(u|u+v)|u \in RM_{(r,m-1)}, v \in RM_{(r-1,m-1)}\}, RM_{(0,m)} = \{(0,\cdots,0),(1,\cdots,1)\}$. Enhanced pqsigRM provides NIST security level 1 (resp. 5). For each security level, it has 474,445 (resp. 2,000,000) bytes of public key and 512 (resp. 1,024) bytes of signature.

- **FIBS** FIBS is a DSA based on isogenies, and it utilizes CGL hash function[17] from the hash-based digital signature schemes XMSS (eXtended Merkle Signature Scheme)[18] and WOTS+ (Winternitz One-Time Signature+)[19]. CGL hash function is specifically designed based on isogenies, enabling it to provide security even in quantum computing environments. Both XMSS and WOTS+ heavily rely on the security of the hash function they use, and consequently, FIBS also ensures security in quantum computing environments by using CGL. In the FIBS scheme, a Pseudo-Random Number Generator (PRNG) is employed to generate random numbers, which are then used along with the public key to produce WOTS+ signatures through the CGL function. By repeating this process multiple times, FORS signatures are generated. One notable characteristic of FIBS is that it only offers a security strength of 128-bit, which corresponds to NIST security level 1. In contrast, other algorithms typically provide at least two or more security levels. FIBS provides 32 bytes of the public key, 64 bytes of the secret key, and 17,088 bytes of signature.

- **GCKSign** GCKSign is a DSA that bears a resemblance to Lyubashevsky's lattice-based DSA scheme[20]. GCKSign has a simpler structure compared to CRYSTALS-Dilithium, one of the final algorithms in the NIST PQC Competition, which allows it to mitigate the risks of side-channel attacks and maintain a smaller signature size. To address the existing challenge of GCK-OW (One-Wayness), GCK-TMO (Target-Modified One-wayness) is introduced and applied to enhance security. The security of GCKSign relies on this new notion. For the signatures, s and s' of the GCK (Generalized Compact Knapsack) function to be different with high probability, the secret key should satisfy $2^{128}q^n < (2\eta + 1)^{mn}$. Here, η represents the range of coefficients in the secret key and should be chosen to be large, which leads to an increase in the size of the signature. The probability of having $s \neq s'$ when the above condition is met is $1 - 2^{-128}$. For NIST security level 2 (resp. 3, 5), GCKSign requires 1,760 (resp. 1,952 and 3,040) bytes for the public key, 288 (resp. 288 and 544) bytes for the secret key, and 1,952 (resp. 2,080 and 3,104) bytes for the signature size.

– **HAETAE** HAETAE is a DSA that utilizes the Fiat-Shamir with aborts scheme. To reduce the sizes of the public key and signature, it introduces hyperball bimodal rejection sampling. HAETAE selects a signature by sampling a random variable y from a hyperball, which follows a bimodal Gaussian distribution. A bimodal Gaussian distribution means a Gaussian distribution with two distinct modes. Unlike the BLISS algorithm[21], which faces implementation challenges with rejection sampling, HAETAE adopts a simplified rejection technique that offers implementation advantages. This choice of rejection sampling allows it to maintain a smaller signature size. HAETAE-II (resp. III, V) provides security levels of 120 (resp. 180, 260)bits. The sizes for HAETAE-II (resp. III, V) are as follows: a public key of 2,529 (resp. 3,836 and 4,817) bytes, a secret key of 1,056 (resp. 1,568 and 2,080) bytes, and a signature size of 3,040 (resp. 4,064 and 5,792) bytes.

– **MQSign** MQSign is a multivariate quadratic-based DSA that relies on the security of the Multivariate Quadratic Problem and the Extended Isomorphism of Polynomials Problem. It utilizes the Unbalanced Oil and Vinegar (UOV) structure[22]. The UOV structure in MQSign consists of two maps, F and T. Among them, F is defined as $F^{(k)}(X) = \sum\limits_{i\in O, j\in V} \alpha_{i,j}^{(k)} x_i x_j + \sum\limits_{i,j\in V, i\leq j} \beta_{i,j}^{(k)} x_i x_j + \sum\limits_{i\in O\cup V} \gamma_i^{(k)} x_i + \eta^{(k)}$, where $F^{(k)}$ is a quadratic polynomial composed of three components, $F_V^{(k)}$, $F_{OV}^{(k)}$, and $F_{LC}^{(k)}$. $F_V^{(k)}$ and $F_{OV}^{(k)}$ are quadratic polynomials consisting of products of Vinegar$\times$Vinegar and Vinegar$\times$Oil variables, respectively, while $F_{LC}^{(k)}$ represents the linear and constant terms. On the other hand, T is a random linear map $T : \mathbb{F}_q^n \to \mathbb{F}_q^n$ chosen over the finite field $\mathbb{F}_q$, and the public key in UOV is obtained by computing $F \circ T$. A notable feature of MQSign is that the secret key can be selected based on four combinations of $F_V^{(k)}$, $F_{OV}^{(k)}$, and $F_{LC}^{(k)}$. The $F_{LC}^{(k)}$ component remains fixed, while the choice of $F_V^{(k)}$ and $F_{OV}^{(k)}$ determines the specific combination used. MQSign provides NIST security levels 1, 3, and 5. The key size and signature size depend on the combination of polynominals.

- **NCCSign** NCCSign is a DSA that applies a non-cyclotomic ring and is similar to the selected algorithm CRYSTALS-Dilithium in the NIST PQC Competition. Traditional lattice-based algorithms use a polynomial ring denoted as $R_q = \mathbb{Z}_q[x]/ < x^n + 1 >$. In this case, using a cyclotomic polynomial allows the use of NTT, which offers advantages in computations. However, NCCSign employs a non-cyclotomic polynomial $\phi(x) = x^p - x - 1$ for a prime number p. This allows for flexible parameter selection by not requiring $n = 2^k$ as in the case of cyclotomic polynomials. However, the drawback is that NTT cannot be used, which leads to a limitation in the polynomial multiplication process, necessitating the use of the Toom-Cook and Karatsuba methods. Apart from the transformation of the module structure in Dilithium to a ring structure, the algorithm of NCCSign remains the same as in Dilithium. NCCSign provides security strengths at NIST security level 1 (resp. 3, 5). The concrete parameters for NCCSign are as follows: a public key size of 1,564 (resp. 1,997 and 2,663) bytes, a secret key size of 2,266 (resp. 3,312 and 4,402) bytes, and a signature size of 2,458 (resp. 3,605 and 5,055) bytes.

- **Peregrine** Peregrine is a DSA based on the NTRU lattice, similar to the selected algorithm FALCON in the NIST PQC Competition. While FALCON requires floating-point operations and utilizes Fast Fourier Transform (FFT), Peregrine uses NTT in $\mathbb{Z}$. Peregrine employs a round-off algorithm, which offers ease of implementation and faster speed but has the drawback of longer signature length and relatively lower security. Furthermore, in contrast to FALCON, which uses precisely computed Gaussian distributions for random variable distributions for each basis, Peregrine uses binomial distributions that do not require floating-point operations. Interestingly, Peregrine's signature verification algorithm is the same as FALCON's, which makes Peregrine and FALCON share many similarities. The Peregrine family consists of Peregrine-512 and Peregrine-1024, providing security strengths at NIST security levels 1 and 5, respectively. The public key, secret key, and signature lengths for Peregrine-512 (resp. 1,024) are 897 (resp. 1,793) bytes for the public key, 1,281 (resp. 2,305) bytes for the secret key, and 666 (resp. 1,280) bytes for the signature.

- **SOLMAE** SOLMAE is a DSA based on the NTRU lattice, similar to FALCON. In SOLMAE, a hybrid sampler was used for sampling, which is the same sampler employed in MITAKA[23]. By using the hybrid sampler, SOLMAE gains advantages such as faster speed, potential for parallel implementation, and ease of implementation. However, SOLMAE-1024 has a longer signature length and lower security compared to FALCON-1024, which is a drawback. One of the distinguishing features of SOLMAE is that it uses the Gaussian distribution for the integer operations in the Z sampler and the elliptical Gaussian distribution for operations in the Fourier domain in the N sampler. This choice was made because the compression technique intended to be applied to SOLMAE works well with elliptical Gaussian distributions.

The SOLMAE family consists of SOLMAE-512 and SOLMAE-1024, providing security strengths at NIST security levels 1 and 5, respectively. The public key and signature lengths for SOLMAE-512 (resp. 1,024) are 896 (resp. 1,792) bytes for the public key and 666 (resp. 1,375) bytes for the signature.

2.2 Valgrind

Valgrind is an instrumentation framework for building dynamic analysis tools. Valgrind provides several tools for memory debugging, memory leaks, and program profiling. Representative tools include Memcheck, Cachegrind, Callgrind, Massif, and Helgrind.

Profile Memory Usage It was tested using Massif, one of Valgrind's tools. Massif is essentially a heap profiler and performs detailed heap profiling by taking periodic snapshots of a program's heap. This show which part allocates the most memory by time and how much is used.

Suitability for Resource-constrained Implementations NIST has mandated the evaluation of candidates in the "microcontroller" environment as part of the NIST PQC process. This has spurred research into optimized implementations of PQC algorithm candidates within constrained environments. Through this, the importance of extending cryptographic algorithms to embedded environments becomes evident. NIST has selected the ARM Cortex-M4 as the standard embedded platform. The Cortex-M4, renowned for its cost and performance efficiency, finds widespread use in various embedded contexts.

Constant-time Implementation Test Timing attack, first proposed by Paul C. Kocher in 1996[24], is an attack that finds secret information by analyzing execution time. Variable timing means that the operation time of the algorithm is different depending on the input value. A timing attack using variable timing allows an attacker to extract secret information such as a secret key or plain text. There are various causes of variable timing. Typically, it is caused by a difference in branching statement and memory access time. If a specific condition of a branch statement is related to secret information, secret information can be extracted through various input values. In addition, if secret information is stored in the cache and executed while minimizing memory access, related information may be leaked due to the difference in access speed. To avoid such attacks, programs must be implemented in constant time. A constant-time implementation means an implementation in which variable timing does not occur for secret information. Modern cryptography presents various methods for constant-time implementation. Examples include replacing branches with bitwise operations or not accessing array indices via secret values. In this paper, we check the KpqC submissions whether variable timing occurs in the content related to secret information.

Memcheck in Valgrind Memcheck of Valgrind is a tool that detects memory management problems mainly for C and C++ programs. Memcheck can detect the following targets: Memory leaks, use of uninitialized values, access to disallowed addresses, free for dynamic allocation, etc. Memcheck provides information about the exact location in the code where these errors occurred and the stack trace leading up to that point, as soon as such errors occur.

2.3 Metamorphic Testing

Metamorphic Testing (MT) is a proposed method for evaluating the implementation correctness of a newly developed program. MT efficiently detects bugs or unexpected behaviors that may not have been found or are difficult to discover within the program. Mouha et al.[25] applied MT to candidates of the SHA-3 Competition and discovered implementation defects in 41 cases, which proved to be challenging for programmers to detect.

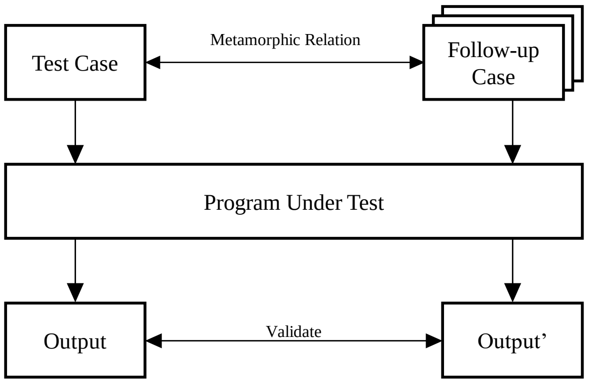

Fig. 1: Overview of Metamorphic Testing

The process of MT is as follows: given a Program Under Test (PUT) P and an input I, a follow-up case I' is generated. The follow-up case is created based on a concept called Metamorphic Relation (MR), which is different for each program. The verifier can define appropriate relations to conduct Metamorphic Testing on PUT. Subsequently, for each generated follow-up case, the program's original output $O = P(I)$ and the output of the follow-up case $O' = P(I')$ are compared to check if they violate the Metamorphic Relation. If a follow-up case violates the relation, it indicates the presence of a bug in the program. The schematic overview of this MT process is depicted in Figure 1.

Mouha et al. proposed four tests of MT: Bit Contribution Test, Bit Exclusion Test, Update Test, and Combinatorial Update Test. In this paper, the Bit Contribution Test and Bit Exclusion Test are applied, along with two additional tests: the Encrypt-Decrypt Test for KEM, and the Bit Verify Test for DSA. Both are designed according to the characteristics of KEM and DSA schemes. The Update Test is not used because we used the same codes for hash functions that passed the Update Test already.

3 Related Works

As the NIST PQC Competition progresses, there has been a surge in active research and projects focused on NIST PQC Competition candidates, including software implementation verification and the application of various security protocols. In EuroS&P'22 [1], an extensive testing framework (continuous integration) called PQClean was introduced to improve the software of NIST PQC candidates. PQClean successfully integrated more than 230 implementations of 17 different parameter sets of NIST PQC candidates. The testing framework conducted verification of NIST PQC candidates based on various implementation criteria and found a number of defects in candidates in most tests, which significantly contributed greatly to improving the quality of NIST PQC candidates. In eprint'19[2], a C reference implementation of NIST PQC candidates for ARM-Cortex-M4 was integrated, and the results provided the feasibility and performance of PQC solutions in embedded environments. In NDSS'22[26], real-time results for the TLS 1.3 handshake under actual network conditions for NIST PQC candidates were presented. The adoption of at least two types of PQ signature algorithms was shown to be feasible without additional overheads compared to existing legacy algorithms. Additionally, the proposal of PQ signature algorithm certificate chains minimized the size of certificates, thereby reducing TLS 1.3 handshake times. In ICISC'22[27], an in-depth performance analysis of NIST PQC candidates applied to IPSec VPN was conducted. The evaluation was based on StrongSwan, and the performance of IPSec VPN was measured by applying NIST PQC KEMs to IKEv2. Some NIST PQC candidates showed performance similar to existing legacy algorithms. Until now, the NIST PQC Competition has garnered significant attention worldwide, and numerous verifications of the candidates have been performed in various verification techniques and security protocols. However, in Korea, S/W verification of KpqC round 1 candidate and verification in various environments is challenging. Thus, in this paper, we present an extensive testing framework such as PQClean and pqm4 to strengthen the S/W quality of the KpqC round 1 candidates.

4 Proposed Extensive Testing Framework Results for KpqC Competition Round 1 Submissions

4.1 Benchmarking Results

Benchmarking Strategy and Results In this section, we describe the approach for the clean library and performance evaluation results for the candidates

Table 4: Benchmarking Result of KEM (clock cycles)

Algorithm	`keygen`	`encap`	`decap`
Layered ROLLO-I-128	566,521	138,693	1,105,803
Layered ROLLO-I-192	923,064	291,653	1,607,498
Layered ROLLO-I-256	967,131	235,696	1,986,714
NTRU+-576	379,297	164,943	173,173
NTRU+-768	418,814	205,980	223,949
NTRU+-864	434,559	231,656	258,189
NTRU+-1152	961,254	305,845	342,358
PALOMA-128	811,949,376	1,045,648	80,723,208
PALOMA-192	937,006,400	922,370	160,819,184
PALOMA-256	1,107,190,400	960,079	164,153,648
SMAUG-128	132,042	84,747	53,387
SMAUG-192	197,674	74,256	105,143
SMAUG-256	288,981	166,963	140,702
TiGER-128	151,587	108,754	94,782
TiGER-192	213,487	137,515	96,453
TiGER-256	245,740	328,930	294,580

of the KpqC Competition including memory usage measurement. Firstly, we removed the entire external dependencies in each algorithm such as OpenSSL. A number of reference codes utilize AES API in OpenSSL for DRBG, which requires that OpenSSL needs to be pre-installed. Also, some algorithms use a hash function in OpenSSL. Thus, we removed all of the dependencies with OpenSSL for clean code and easy building.

The evaluation was conducted using an Intel Core i7-13700K processor on Ubuntu 20.04 LTS with compile option -O3. We measured the clock cycles for each security level. However, the benchmarking for REDOG was excluded from the results. It is because the submitted reference code for REDOG was implemented in Python, making it impractical to compare its benchmarking results with other algorithms that were implemented in C/C++. Benchmarking for REDOG will be included in the future when a reference code implemented in C is submitted. Also, the benchmarking result of IPCC is not included because the reference code of IPCC does not work in Ubuntu. We checked that it works well in macOS with M1, but our benchmarking environment is Ubuntu. Thus, we omitted the result of IPCC because of the unbalanced environment: we will benchmark IPCC with future work when the reference code runs well in Ubuntu. All measurements were based on the average results of 100 runs. All of the benchmarking results are shown in Table 4 and 5. Also, the graphs of the benchmarking results are shown in Figure 10-12 in Appendix. In addition, the benchmarking performance of FIBS was measured, but the results were not shown. The reason was the long signing time, which made it challenging to display the graphs of other algorithms even on a log scale.

The benchmarking graph for `keygen`, `encap`, and `decap` of KEM is presented in Table 4. In `keygen`, PALOMA showed the highest computational overheads, while SMAUG and TiGER showed relatively lower clock cycles overall. In addition, Layered ROLLO-I and NTRU+ showed similar performance at security levels 1 and 3, but at security level 5, NTRU+ required more clock cycles. In `encap`, PALOMA exhibited the highest number of clock cycles, and TiGER showed better performance than NTRU+ except at security level 5, where they performed similarly. Layered ROLLO-I and SMAUG achieved the lowest performance, with similar results at security levels 1 and 3, but SMAUG outperformed Layered ROLLO I at security level 5. In `decap`, similar to `keygen` and `encap`, PALOMA exhibited the highest number of clock cycles. It was followed by Layered ROLLO-I and NTRU+, while SMAUG and TiGER showed lower clock cycles in `decap`. Overall, the benchmarking results showed that PALOMA had

Table 5: Benchmarking Result of DSA (clock cycles)

Algorithm	`keygen`	`sign`	`verify`
AIMer-I	343,199	6,826,911	6,181,673
AIMer-III	710,477	12,677,123	11,858,871
AIMer-V	1,786,713	27,092,783	25,736,881
Enhanced pqsigRM-612	929,021,499	878,157	229,538
Enhanced pqsigRM-613	234,120,808	3,912,608	1,636,105
FIBS	311,493,411,686	7,228,086,586,377	423,968,581,653
GCKSign-II	226,499	1,007,346	176,441
GCKSign-III	235,363	895,923	235,314
GCKSign-V	305,983	1,081,682	299,714
HAETAE-II	1,210,536	3,946,594	1,818,119
HAETAE-III	2,731,698	9,666,993	3,544,363
HAETAE-V	4,200,368	17,259,545	5,399,993
MQSign-72/46	43,538,176	429,170	606,638
MQSign-112/72	160,988,541	695,351	1,546,297
MQSign-148/96	412,059,415	1,526,165	3,267,239
NCCSign-II	1,654,211	23,575,814	2,850,279
NCCSign-II aes	1,618,854	23,330,995	2,846,476
NCCSign-III	2,974,413	43,124,795	5,671,150
NCCSign-III aes	2,921,031	40,787,350	5,506,699
NCCSign-V	5,576,377	82,360,508	10,617,240
NCCSign-V aes	5,534,544	80,069,836	10,623,338
Peregrine-512	10,716,809	353,576	10,245
Peregrine-1024	36,850,723	614,001	85,073
SOLMAE-512	24,933,588	313,957	33,225
SOLMAE-1024	63,966,071	601,942	142,281

Table 6: Benchmarking of Stack Size of KEM(bytes)

Algorithm	Heap/Stack Usage (B)
Layered ROLLO-I-128	17,144 / 7,832
Layered ROLLO-I-192	19,752 / 9,520
Layered ROLLO-I-256	24,232 / 13,176
NTRU+-576	2,000 / 17,808
NTRU+-768	2,000 / 22,928
NTRU+-864	2,000 / 25,488
NTRU+-1152	2,000 / 33,136
PALOMA-128	0 / 16,641,680
PALOMA-192	0 / 16,641,792
PALOMA-256	0 / 16,641,792
SMAUG-128	2,320 / 9,584
SMAUG-192	3,920 / 14,560
SMAUG-256	9,040 / 27,472
TiGER-128	1,248 / 11,944
TiGER-192	1,248 / 18,000
TiGER-256	1,248 / 31,208

the slowest performance, and SMAUG had the fastest performance among the KEM algorithms.

The benchmarking result for `keygen`, `sign`, and `verify` of DSA is presented in Table 5. In `keygen`, similar to KEM, it shows an increasing trend in required clock cycles as the security level rises. However, Enhanced pqsigRM exhibited a decrease in clock cycles even with increasing security levels. Overall, Enhanced pqsigRM and MQSign required the highest clock cycles, while GCKSign and AIMer needed relatively lower clock cycles. In `sign`, MQSign took the longest to generate signatures, followed by AIMer and HAETAE. Peregrine and SOLMAE showed similar performance and at the same time generated the fastest signatures. In particular, GCKSign showed intermediate performance in security level 1 but showed the third fastest performance in level 5, the highest security level. In `verify`, all algorithms tended to increase clock cycles as the security level increased, and AIMer showed the slowest verification speed. Peregrine and SOLMAE showed the fastest verification speed in sign, but Peregrine was slightly faster in sign but showed a significant performance gap with SOLMAE in `verify`. GCKSign was measured with a small gap in clock cycles required even though the security level increased.

To gain insights into the memory usage patterns of the algorithms at different security levels, we harnessed Massif. This tool enabled us to determine the peak memory consumption of each algorithm. To utilize Massif when running Valgrind, it is necessary to specify it as the tool to be used by adding `--tool=massif` as an option. Memory usage during execution can be measured using the Massif

Table 7: Benchmarking of Stack Size of DSA(bytes)

Algorithm	Max Memory Usage(Heap/Stack)
AIMer-I	215,656 / 6,432
AIMer-III	461,000 / 6,624
AIMer-V	913,432 / 5,264
Enhanced pqsigRM-612	4,848,448 / 1,620,416
Enhanced pqsigRM-613	25,179,560 / 4,248,240
GCKSign-II	14,896 / 51,560
GCKSign-III	15,152 / 48,808
GCKSign-V	17,200 / 80,136
HAETAE-II	17,072 / 106,080
HAETAE-III	20,080 / 156,152
HAETAE-V	22,576 / 196,264
MQSign-72/46	345,320 / 904,488
MQSign-112/72	1,277,976 / 3,400,664
MQSign-148/96	2,960,968 / 7,949,128
NCCSign-I	15,888 / 187,688
NCCSign-III	18,192 / 261,736
NCCSign-V	21,104 / 347,272
Peregrine-512	119,104 / 7,120
Peregrine-1024	230,208 / 5,376
SOLMAE-512	19,264 / 120,144
SOLMAE-1024	27,408 / 237,904

tool with `--stacks=yes` option. The results are stored in `massif.out`, and we can view them using `ms_print` command.

We have presented the memory usage of KEM algorithms for Round 1 candidates in Table 6. The maximum usage for both heap and stack is indicated in bytes. The algorithm that exhibited the highest memory consumption is PALOMA, showing the highest memory usage across all security levels. It is noteworthy that PALOMA did not use heap memory, attributed to its implementation avoiding memory allocation. Furthermore, among all algorithms, SMAUG and TiGER demonstrated the least memory usage overall. Except for PALOMA, the remaining algorithms utilized both stack and heap memory according to their respective reference codes.

We have presented the memory usage of DSA algorithms for Round 1 candidates in Table 7. The algorithm that consumed the most memory was Enhanced pqsigRM, particularly Enhanced pqsigRM-613, which exhibited significant heap usage. Additionally, the algorithms with the least memory usage were Peregrine, HAETAE, and SOLMAE. It was observed that all reference codes utilized both stack and heap memory. While Table 6 and 7 represent the maximum usage of heap and stack memory individually, these values may vary depending on the implementation approach. For instance, implementations that avoid dynamic allocation, like PALOMA, may reduce heap usage, whereas heavy dynamic memory allocation could increase heap consumption while reducing stack usage. Therefore, the provided tables aim to divide the total memory usage into stack and heap components, highlighting that it is challenging to definitively determine whether the given implementations will use precisely the specified sizes of heap or stack. In summary, it should be emphasized that the tables are intended to illustrate the total memory usage, divided into stack and heap while acknowledging that the exact heap or stack sizes used by these implementations may not be conclusively determined.

Suitability Assessment for Resource-constrained Implementation We explore the feasibility of implementing KpqC algorithm candidates in resource-constrained environments, taking into consideration the results of Massif's memory profiling. In line with the findings in [2], we specifically target the STM32F4 Discovery board, which features an ARM Cortex-M4 CPU clocked at 168MHz, 1MB of flash memory, and 192KB of RAM. By evaluating the memory usage of KpqC algorithm candidates and comparing it to the available RAM on the board, we assess both implementation feasibility and scalability.

According to [2], the 192KiB memory of the STM32F4DISCOVERY board is segmented into multiple memory regions. However, this paper evaluated algorithm candidates based on the 192KiB threshold. For KEM, the PALOMA algorithm, and for DSA, the AIMer, Enhanced pqsigRM, MQsign, and NCC-sign algorithms were determined to be unsuitable for implementation in constrained environments. The total memory usage of PALOMA varied with security strength but was measured at 16,251KiB, representing the highest among KEM. Additionally, the memory usage of Enhanced pqsigRM recorded approximately 28,738KiB, marking the highest among DSA. Apart from the mentioned algorithms, HAETAE, Peregrine, and SOLMAE also exhibited cases where memory usage exceeded 192KiB depending on the parameter set. While using higher-tier boards could increase RAM capacity, it demands a proportional increase in cost, making it unsuitable. Thus, research on memory optimization techniques is necessary for the embedded implementation of these algorithms.

Profiling Result In this section, we provide profiling results for certain components of the Round 1 DSA candidates. The profiling results present the top three most frequently called functions in each algorithm's `keygen`, `sign`, and `verify`. The profiling was conducted using Visual Studio 2019. To enable execution in MSVC for functions and GCC extensions that are specific to Ubuntu, minor code modifications were made to ensure minimal impact on overall performance. Profiling results for KEM and FIBS are not included in this section and are left as future work. The profiling outcomes are presented in Table 12 and 13 in the Appendix.

In most algorithms, bottlenecks primarily occurred in polynomial multiplication, division, reduction, matrix multiplication, and Keccack. Particularly, cases like HAETAE employing schoolbook algorithms and using the C language's modulo operation for remainder calculations resulted in this operation accounting for approximately 90% of the total computations, causing significant performance degradation. Furthermore, algorithms based on NTRU problems like Peregrine experienced bottlenecks due to operations related to NTRU problems. Algorithms utilizing finite field operations, such as AIMer, MQSign, and NCCSign, also faced bottlenecks related to finite field computations. In AIMer, Keccak operations emerged as a significant bottleneck across the entire algorithm.

Consequently, optimizing the parts responsible for bottlenecks in each algorithm could lead to substantial performance improvements in their optimal implementations. For example, the naive implementation of polynomial multiplication in HAETAE has a time complexity of $O(n^2)$. Replacing such an implementation with a multiplication algorithm like Toom-Cook or Karatsuba, which has a lower time complexity, is expected to directly impact the implementation's performance. In cases like SOLMAE, which uses FFT with a time complexity of $O(n \log n)$, it might be challenging to replace it with a lower-complexity multiplication algorithm. Thus, utilizing the optimized code for the respective algorithm can be expected to enhance performance. In order to achieve post-quantum security in resource-constrained environments, there is a need for optimal implementation research on the candidates of the KpqC Competition. We hope that the profiling results presented in this paper will help identify bottlenecks in each algorithm, stimulating active research toward optimal implementations tailored to each algorithm's characteristics.

4.2 Security Analysis Strategy and Results

In this section, we provide security analysis results for 1 Round KpqC submissions by classifying them into basic S/W verification and Constant-time verification using the Valgrind tool, and metamorphic testing. Before providing security analysis, we first define the risk level of found bugs and errors. Afterwards, we provide detail results of security analysis for 1 Round KpqC submissions through the basic S/W verification, cost-time implementation verification, and metamorphic testing.

Defining bugs and errors In this section, we define the bugs and errors found in our test framework, including a design rationale for proposed Metamorphic Testings. We designed our test framework with basic compile warning, memory error, and constant-time implementation verification through the Valgrind tool, and verification through metamorphic testing. Errors generated in this verification can be largely classified into four categories, and the meanings of each error are as follows.

– **Compile Warnings** Warnings generated during compilation are intended to alert the developer to potential problems with code and do not have a fatal effect when the developer runs the compiled file. As the compiler analyzes the source code, it points out non-errors but potential bugs or informs the developer of areas where the quality of the code can be improved. By processing these warnings without ignoring them, the quality of the program can be increased. Common compiler warnings include: There are `Unused Variable`, `Uninitialized Variable`, `Type Mismatch`, and `Type Redefinition`, `etc.` `Unused Variable` and v are warnings that occur when an unused variable exists and when a variable is used without initialization. `Type Mismatch` and `Type Redefinition` occur when the data type of a variable is different from what was expected and when multiple variables with the same name are defined. In addition to these warnings, various warnings can occur depending on the compiler version and compilation options used. These warnings should be handled carefully to improve code quality and prevent bugs.

- **Memory Error** There are various causes of memory errors in algorithm behavior. First, we have the issue of `Memory Leak`. This occurs when a program dynamically allocates memory space but fails to deallocate it. In C/C++ language, where there is no garbage collector, it's the programmer's responsibility to release memory. While memory is automatically reclaimed when the process terminates, it is essential to free unused memory during runtime to avoid memory waste. Next, we encounter `Invalid Read` errors, which involve reading from or writing to memory locations that have not been initialized or allocated. This can lead to unintended data access, potentially resulting in incorrect encryption/decryption outcomes. Errors may also arise from referencing invalid memory addresses. For instance, if a function uses a memory address declared as a local variable, the validity of that memory address may differ depending on the function's termination status. While `Memory Leak` may be considered minor errors, `Invalid Read`, which involve using invalid data, can lead to critical issues that affect the behavior or results of the program.
- **Metamorphic Testing Error** Errors in metamorphic testing mean that forged inputs produce results for original inputs. In other words, forgery attacks are possible due to errors in the implementation, which can have a significant impact on authentication services. Metamorphic testing performs wide-coverage verification of implementations by designing metamorphic relations based on algorithmic relationships. Representative metamorphic testing includes Bit Contribution, Bit Exclusion, Encrypt-Decrypt, and Bit Verify verification. These verifications are performed by comparing the result with the original value and the result with the forgery value. Thus, the metamorphic testing error is a very fatal error because a forgery attack is possible.
- **Non-Constant-time Implementation** Code that handles sensitive information, such as private keys, must adhere to the principle of being "constant-time" to prevent the leakage of secret values through side channels, including timing attacks. This entails avoiding any conditional branches that depend on secret values and refraining from memory accesses that are determined by secret information. Algorithms that fail to meet these constant-time conditions are classified as `Constant-test Fail`. This vulnerability to side-

channel attacks, particularly timing attacks, presents a critical issue, as it can lead to severe security risks. In 2.2, a detailed explanation of constant-time implementation is provided.

Design of Valgrind Tests We provide an overview of the Valgrind tests conducted on the candidates for the KpqC Competition. The focus of these tests was on Constant-time Tests and Memory Error checks for the submitted algorithms. To facilitate these tests, a dedicated test code which is named `const_test` was developed, specifically designed for evaluating KEM (Key Encapsulation Mechanism) and DSA (Digital Signature Algorithm) operations. Notably, this test code excluded file I/O functionalities inherited from the NIST_PQCgenKAT code. We use Memcheck to test the constant-time implementation and detect Memory Errors of the algorithms proposed in the KpqC competition. The tests we conducted are described in detail below.

- **Constant-time Implementation Test** In this paper, a test tool was generated using Memcheck's built-in functions `VALGRIND_MAKE_MEM_UNDEFINED()`, `VALGRIND_MAKE_MEM_DEFINED()`, and `VALGRIND_CHECK_MEM_IS_DEFINED()`. Referring to [28], `poison()`, we defined `unpoison()`, and `is_poisoned()` in the poison.h file. In Valgrind, there is a function to trace undefined data. This function reports when data derived from undefined data is used for branching or memory access. We can track the possible occurrence of variable timing by identifying the flag as undefined data using a secret. At this time, we can set the flag using `poison()`. The input value is the address and size of the data to set the flag. The flag can be released with `unpoison()`. In addition, a warning about an uninitialized variable occurs, and to distinguish it from the data we want, we can check which value the flag is for through `is_poisoned()`. we incorporated flags into the `const_test` code, with a specific focus on crucial points such as the `Crypto_Encryption` function for KEM algorithms and the `Crypto_sign` function for DSA algorithms. These flags were strategically placed to highlight secret information being processed. During the Valgrind Memcheck tests, our analysis revolved around interpreting warnings. This process involved careful scrutiny to distinguish between uninitialized variables, which triggered warnings, and the actual flagged data that was a subject of interest. Each algorithm was tested with the lowest security level standard. When executing Valgrind, the movement path of the flag was tracked using the `--track-origins=yes` option. Since Memcheck is already included in the file when Valgrind is executed, it is not necessary to specify it separately.
- **Memory Error Test** Memcheck is capable of inspecting Memory Errors. By using `-leak-check=full` and `-show-leak-kinds=all` options, it becomes possible to check for memory leaks and other issues within the code. We identified the precise error occurrence locations by using the `-g` compile option.

Design Rationale of Metamorphic Testing In the case of Metamorphic Testing, we applied a total of 4 tests: Bit Contribution Test, Bit Exclusion Test, Encrypt-Decrypt Test, and Bit Verify Test. Each test is designed considering the characteristics of the algorithms, or basic properties of cryptographic algorithms. We designed the Metamorphic Relations and implementation approach of each test. The encrypt-Decrypt Test is designed for KEM/PKE, and the Bit Verify Test is designed for DSA. The tests applied for each algorithm are presented in Table. The overall design rationale is similar to Mouha et al.[25], but it has some differences. Also, we did not apply the Update Test, which verifies `Update()` of a hash function, providing the reasons in the following subsections.

– **Bit Contribution Test**

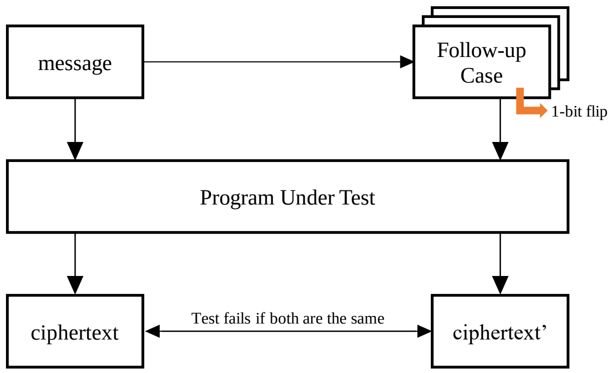

Fig. 2: Overview of Bit Contribution Test for KEM/DSA

The Bit Contribution Test is designed based on the fundamental characteristic of cryptographic algorithms where different plaintexts result in different ciphertexts. In this scenario, a fixed-size message m of length n is chosen, and its ciphertext c is computed. Then, a follow-up case is created by changing one bit of m, resulting in a new message m', and its corresponding ciphertext c' is computed. This process is repeated for all bits of m, and it checks whether two different ciphertext pairs exist with the same plaintexts. The Bit Contribution Test succeeds if no identical ciphertext pair is found.
This scenario relies on the 'avalanche effect' of cryptographic hash functions. The avalanche effect is the property of a hash function where a small change in the input value should cause a completely different hash value. Similarly, cryptographic algorithms must produce completely different ciphertexts for different inputs. Therefore, the same design can be applied to cryptographic algorithms. In the Bit Contribution Test, the MR states that "if the keys are the same, different plaintexts must result in different ciphertexts." The overview of the Bit Contribution Test is shown in Figure 2.

– **Bit Exclusion Test**

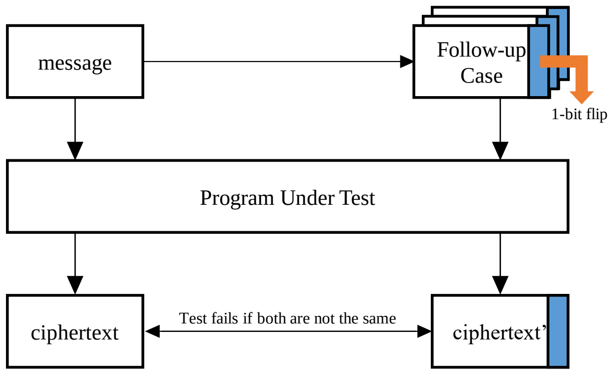

Fig. 3: Overview of Bit Exclusion Test for KEM/DSA

The Bit Exclusion Test is not designed based on the characteristics of the algorithm itself but rather on the features of the implementation environment. In the C language, it allows programmers to read beyond the allocated size of an array. As a result, it can lead to unintended consequences, such as mistakenly reading the last bit of a message or reading beyond the message's length. If the implementation reads beyond the message's length and uses that value in the encryption, the final ciphertext may differ from the expected result. The Bit Exclusion Test is designed to detect such implementation 'mistakes' or bugs. In this scenario, a fixed-length message m of size n is taken, and follow-up cases m' are generated by changing bits beyond the message's length, i.e., bits $n + 1$, $n + 2$, and so on. Then, the original message's ciphertext c and the follow-up case's ciphertext c' are compared. If they are different, it indicates that the encryption used data beyond the allocated memory, resulting in a failure of the Bit Exclusion Test. For example, let's consider a scenario where the message length is 128 bits, and the length information is passed to the encryption function. The message is stored in an array of 16 elements of each byte since the message is stored as a byte array. If we set the array size to 20 and change the 128-th bit of the message (assuming 0-indexing), the message length remains 128 bits when passed to the encryption. Therefore, the encryption should not use the 128-th bit. If the ciphertext changes after modifying the 128-th bit, it indicates that data beyond the given message length was read, which may indicate a bug in the implementation. In the Bit Exclusion Test, the MR states that "if the length information is the same, changing data beyond the length does not affect the ciphertext." The overview of the Bit Exclusion Test is shown in Figure 3. The blue area means the extended array of the original message.

– **Update Test**

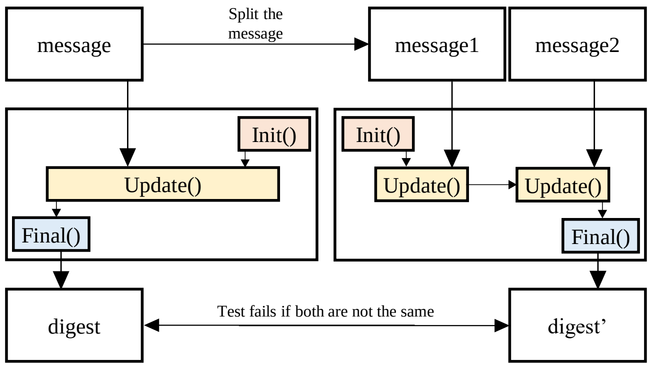

Fig. 4: Overview of Update Test

The Update Test is designed based on the characteristics of the `Update()` process in a hash function. According to the SHA-3 Competition API Specification [29], an implementation must provide four different functions: `Init()`, `Update()`, `Final()`, and `Hash()`. Among these, `Update()` is called after `Init()` to process the message and compute the digest. Finally, calling `Final()` results in obtaining the hash value y. Assume that a message m is divided into multiple message blocks $m_1, m_2, ...$ on a block-by-block basis (where the length of each m_i is assumed to be a multiple of 8 bits). In this case, after calling `Init()`, the implementation performs `Update (`m_1`)`, `Update (`m_2`)`, and so on, and finally executes `Final()` to obtain the hash value y'. The generated y and y' must be the same, which is a requirement specified in the API's incremental processing requirement. This requirement is important when hashing very long messages since waiting for the entire message to be received in situations like packet transmission over a network can result in delays. Therefore, the Update Test compares the hash value of the original message m with the hash values of its partitions $m_1, m_2, ...$ in consecutive order. If the two hash values differ, the implementation fails the Update Test as it does not meet the API's requirements. In this paper, the Update Test was not applied because the hash function used in the paper is based on the code from PQClean[30], and this code has been verified to pass the Update Test. Since the Update Test is entirely dependent on the implementation of the hash function in algorithms that include a hash function, if the used hash function implementation passes the Update Test, then the algorithm implementation will also pass the Update Test. The overview of the Update Test is shown in Figure 4.

– **Encrypt-Decrypt Test**

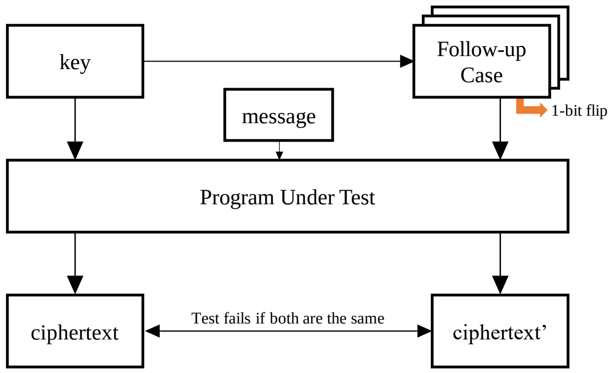

Fig. 5: Overview of Encrypt-Decrypt Test for KEM

The Encrypt-Decrypt Test is a test similar to the Bit Contribution Test and is designed based on the characteristics of the KEM scheme where different keys result in different ciphertexts. In this scenario, a fixed key k and a fixed message m are chosen, and the ciphertext c is computed. Then, follow-up cases are generated by changing one bit of the key k, resulting in new keys k', and their corresponding ciphertexts c' are computed. This process is repeated for all bits of k, and if no two identical ciphertexts are found among these ciphertexts, the Encrypt-Decrypt Test is successful. The MR in the Encrypt-Decrypt Test states that "if the message is the same, different keys must result in different ciphertexts." This test ensures that the encryption process produces different ciphertexts when different keys are used, thereby confirming the security property of the algorithm that ciphertexts are dependent on the choice of keys. The overview of the Encrypt-Decrypt Test is shown in Figure 5.

– **Bit Verify Test**
The Bit Verify Test is a test similar to the Bit Contribution Test and is designed based on the signature property of the DSA. DSA signatures are dependent on the message being signed. Therefore, if a signature is tampered with for a certain message, the signature would be recognized as invalid by the `Verify()` function in DSA. Inspired by this idea, the following scenario is conducted: First, a signature s (or sm where m is the message) is generated for a given message m. Then, a follow-up case is created by changing one bit of the signature s (or sm), resulting in s' (or sm'). Next, the altered signature is verified. If the modified signature is validated, the Bit Verify Test fails.
The MR in the Bit Verify Test states that "for the same message, the signature remains the same." This test ensures that any modifications to the

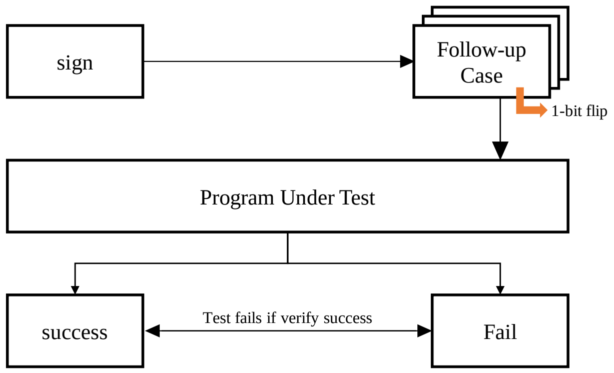

Fig. 6: Overview of Bit Verify Test for DSA

signature or the signature concatenated with the message result in an invalid signature, as expected by the security property of DSA. The overview of the Bit Verify Test is shown in Figure 6. Also, the original signature should be verified.

When performing verification for the aforementioned tests, all random values are fixed during the execution. All algorithms submitted to KpqC Competition generate random numbers using the `randombytes()` function as a Deterministic Random Bit Generator (DRBG). Due to this, even for the same input message, there exists a probabilistic possibility that the ciphertext and signature may differ. If random values are not fixed, it can lead to incorrect test results. For instance, consider the scenario of the Bit Exclusion Test.

In reality, if the implementation does not read beyond the array boundary and successfully passes the Bit Exclusion Test, it is still possible that the ciphertext might probabilistically differ due to random variations. Consequently, it may fail the Bit Exclusion Test. Thus, in order to obtain accurate Metamorphic Testing results, it is essential to fix all random variables to constant values.

Valgrind Test Results By conducting Valgrind tests, we were able to comprehensively evaluate the candidates' performance in terms of memory usage, constant-time behavior, and memory management. This thorough analysis assisted in selecting algorithms that are not only robust and secure but also feasible for deployment in embedded environments. The test results were categorized for KEM and DSA algorithms and are presented in Table 8 and 9, respectively. We were unable to perform Valgrind tests for some algorithms due to the following reasons.

– **IPCC** The IPCC algorithm builds successfully in the Ubuntu environment; however, it fails to generate the correct keys during decapsulation, making valid testing impossible. On macOS, it runs correctly, but Valgrind, being

Table 8: Results of Valgrind Testing on KEM

Algorithm	Const-time Test	Memory Error
Layered ROLLO-I-128	PASS	Leak
Layered ROLLO-I-192		
Layered ROLLO-I-256		
NTRU+-576	PASS	None
NTRU+-768		
NTRU+-864		
NTRU+-1152		
PALOMA-128	PASS	None
PALOMA-192		
PALOMA-256		
SMAUG-128	PASS	None
SMAUG-192		
SMAUG-256		
TiGER-128	PASS	None
TiGER-192		
TiGER-256		

a tool designed for Linux environments, prevents us from conducting tests on that platform. However, we plan to conduct the tests in the future if the algorithm can be executed correctly on Linux.

– **REDOG** The REDOG algorithm's implementation package was written in Python. Since Valgrind is effective for programs written in C/C++, we did not perform the test for REDOG. However, if submissions in other languages are presented in the competition, we plan to conduct tests accordingly.

Below is a comprehensive overview of the Valgrind test results. The identified errors have been classified in accordance with the definitions outlined in Section 4.2. The Layered ROLLO-I and Enhanced pqsigRM algorithms have been found to exhibit Memory Errors such as `Memory Leak`, `Invalid Read`. In this section, we also explain the feasibility of implementing the proposed algorithm candidates within constrained environments.

Memory Error

– **Layered ROLLO-I** In the Layered ROLLO-I, `Memory Leak` have been detected. Testing was conducted on the biix algorithm. The majority of memory leaks occurred in the `init` function, where memory for data used in computations was allocated. A notable example is within the `biix_sk_generate()` function, where memory allocated for `invX1` was not released until the algorithm's termination. Additionally, memory allocation for `pkTmp1` occurred twice. To address this, appropriate memory free functions were introduced, and memory allocation was modified to occur only once. Another instance

Table 9: Results of Valgrind Testing on DSA

Algorithm	Const-time Test	Memory Error
AIMer-I	PASS	None
AIMer-III		
AIMer-V		
Enhanced pqsigRM-612	PASS	Leak/Invalid Read
Enhanced pqsigRM-613		
GCKSign-II	PASS	None
GCKSign-III		
GCKSign-V		
HAETAE-II	PASS	None
HAETAE-III		
HAETAE-V		
MQSign-72/46	PASS	None
MQSign-112/72		
MQSign-148/96		
NCCSign-I	PASS	None
NCCSign-III		
NCCSign-V		
Peregrine-512	PASS	None
Peregrine-1024		
SOLMAE-512	PASS	None
SOLMAE-1024		

involved allocating memory for a specific structure's size during initialization and releasing memory using a function that only freed some members of the structure, leading to memory leaks. Similar errors stemming from similar causes were also identified. In the case of `biix_128`, approximately 4,852 bytes of unreleased memory were detected. The pseudo-code for modifying `Memory Leak` error example is shown in Algorithm 1.

– **Enhanced pqsigRM** The Enhanced pqsigRM algorithm encountered `Memory Leak` and `Invalid Read` errors during the test. `Memory Leaks` were identified within the `crypto_sign` and `crypto_sign_keypair()`. For example, the `sign` variable in `crypto_sign` was allocated memory using the `newmatrix()` function but was not freed. Therefore, the memory pointed to by `sign` must be deallocated before the termination of the function. The same type of memory error was also found in the `crypto_sign_keypair()` function. In another instance, a pointer pointing to the memory allocated in `init_decoding` remained until the algorithm's termination. Similarly, it is necessary to include a section for deallocating the memory. For the Enhanced pqsigRM-612, 332,558 bytes of unreleased memory and 16,384 bytes pointed by the remaining pointer were recorded. The `Invalid Read` occurred within the `export_sk()` function. During the process of copying the contents of the

Algorithm 1 Code Snippet of Modified L-ROLLO sk_gen Function

```
1: biix_secretKey skTmp1
2: biix_publicKey pkTmp1
   ...
   // deleted code
3: // rbc_qre_init(&(pkTmp1.h))
   ...
4: rbc_qre_init(&(pkTmp1.h))
5: rbc_qre_mul(pkTmp1.h, invX1, skTmp1.y)
   ...
   // added code
6: rbc_qre_clear(invX1)
7: rbc_qre_clear_modulus()
8: return
```

Algorithm 2 Code Snippet of Modified Enh. pqsigRM crypto_sign Function

```
1: uint16_t *Q, *part_perm1, *part_perm2, *s_lead;
   ...
   // deleted code
2: matrix *sign = newMatrix(1, CODE_N)
   ...
   // added code
3: deleteMatrix(sign)
   ...
4: return
```

Algorithm 3 Code Snippet of Modified Enh. pqsigRM export_sk Function

```
1: exportMatrix(sk, Sinv)
2: memcpy(sk + Sinv->alloc_size, Q, sizeof(uint16_t) * CODE_N)
3: memcpy(sk + Sinv->alloc_size + sizeof(uint16_t) * CODE_N,
   part_perm1, sizeof(uint16_t) * CODE_N / 4)
   // added code
4: memcpy(sk + Sinv->alloc_size + sizeof(uint16_t) * CODE_N + sizeof(uint16_t)
   * CODE_N/4,
   part_perm2, sizeof(uint16_t) * CODE_N / 4)
```

`part_perm2` variable using the `memcpy()` function, an incorrect range size was passed. Modification of the parameter indicating the size to be read is necessary. For the correction of `Memory Leak` and `Invalid Read` errors, we have the following sections of pseudo-code for each Algorithm 2, 3.

Metamorphic Testing Results In this section, we present the results of the Metamorphic Tests conducted on the candidates of the KpqC Competition. For KEMs, we conducted the Bit Contribution Test and the Bit Exclusion Test. For

Table 10: Results of Metamorphic Testing on KEM

Algorithm	Bit Contribution Test	Bit Exclusion Test	En-Decrypt Test
Layered ROLLO-I	Pass	Pass	Pass
PALOMA-128	Pass	Pass	Pass
PALOMA-192	Pass	Pass	Pass
PALOMA-256	Pass	Pass	Pass
SMAUG-128	Pass	Pass	Pass
SMAUG-192	Pass	Pass	Pass
SMAUG-256	Pass	Pass	Pass
TiGER-128	Pass	Pass	Pass
TiGER-192	Pass	Pass	Pass
TiGER-256	Pass	Pass	Pass
NTRU+-576	Pass	Pass	Pass
NTRU+-768	Pass	Pass	Pass
NTRU+-864	Pass	Pass	Pass
NTRU+-1152	Pass	Pass	Pass

DSAs, we performed the Bit Contribution Test, the Bit Exclusion Test, and the Bit Verify Test. All tests were conducted based on the reference code provided on the KpqC website. However, we were unable to provide the Metamorphic Testing results for all algorithms, and the reasons are as follows:

- **IPCC** IPCC implementation is built successfully in the Ubuntu environment; however, during the decapsulation process, it fails to generate the correct keys.The author of IPCC confirmed that it builds correctly in the macOS environment, but in our development environment, Ubuntu, the same code does not produce the expected test vectors, making it difficult to expect the entire code to function correctly.
- **Peregrine** Peregrine runs successfully; however, in the reference code, the `keypair()`, `sign`, and `verify` functions are not separated. All the steps are implemented within a single `main` function, which makes it challenging to conduct Metamorphic Testing. For future research, we plan to refactor the code and separate each step into its own function to facilitate Metamorphic Testing.

Table 11: Results of Metamorphic Testing on DSA

Algorithm	Bit Contribution Test	Bit Exclusion Test	Bit Verify Test
AIMer-I	Pass	Pass	Pass
AIMer-III	Pass	Pass	Pass
AIMer-V	Pass	Pass	Pass
GCKSign-II	Pass	Pass	Pass
GCKSign-III	Pass	Pass	Pass
GCKSign-V	Pass	Pass	Pass
HAETAE-II	Pass	Pass	Pass
HAETAE-III	Pass	Pass	Pass
HAETAE-V	Pass	Pass	Pass
MQSign-72/46	Pass	Pass	Pass
MQSign-112/72	Pass	Pass	Pass
MQSign-148/96	Pass	Pass	Pass
NCCSign-I	Pass	Pass	Pass
NCCSign-III	Pass	Pass	Pass
NCCSign-V	Pass	Pass	Pass
Peregrine-512	Pass	Pass	Pass
Peregrine-1024	Pass	Pass	Pass
SOLMAE-512	Pass	Pass	Pass
SOLMAE-1024	Pass	Pass	Pass

The results of the Metamorphic Testing conducted in this paper are shown in Table 10 and 11. All algorithms passed the Bit Exclusion Test, Bit Verify Test, and Encrypt-Decrypt Test for all security levels.

5 Concluding Remarks

In this paper, we conducted extensive tests to verify the round 1 candidates of the KpqC Competition considering the performance and implementation security. From a performance point of view, we presented benchmarking results in a general CPU environment for fair comparison and evaluated applicability in resource-constrained devices with respect to memory usage. Lattice-based algorithms such as SMAUG, TIGER, and NTRU+ in KEM and HAETAE, GCKSign, and SOLMAE, etc. in DSA were the most competitive. Furthermore, we presented bottlenecks through the profiling of some round 1 candidates and suggested optimization points. Most of the bottlenecks have been Keccak and polynomial multiplication. From implementation security, we performed Valgrind tests to find errors and bugs, Metamorphic Testing with extensive test coverage, and evaluation of the side channel resistance. We found memory errors in the Layered ROLLO-I and Enhanced pqsigRM through Valgrind tests. This was because the allocated memory was not freed or an overflow accessing an area other than the allocated memory. Through our extensive test, we expect to improve the software quality of the KpqC first-round candidates. We plan to integrate and verify the round 1 candidates of the KpqC Competition in various environments such as ARM Cortex-M4, ARMv8, and GPU in the future.

References

1. Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *IEEE European Symposium on Security and Privacy, EuroS&amp;P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society.
2. Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking nist pqc on arm cortex-m4. Cryptology ePrint Archive, Paper 2019/844, 2019. https://eprint.iacr.org/2019/844.
3. Christian Paquin, Douglas Stebila, and Goutam Tamvada. Benchmarking post-quantum cryptography in TLS. *IACR Cryptol. ePrint Arch.*, page 1447, 2019.
4. Seungyeon Bae, Yousung Chang, Hyeongjin Park, Minseo Kim, and Youngjoo Shin. A performance evaluation of ipsec with post-quantum cryptography. In Seung-Hyun Seo and Hwajeong Seo, editors, *Information Security and Cryptology - ICISC 2022 - 25th International Conference, ICISC 2022, Seoul, South Korea, November 30 - December 2, 2022, Revised Selected Papers*, volume 13849 of *Lecture Notes in Computer Science*, pages 249–266. Springer, 2022.
5. Sydney Pugh, Mohammad S. Raunak, D. Richard Kuhn, and Raghu Kacker. Systematic testing of post-quantum cryptographic implementations using metamorphic testing. In Xiaoyuan Xie, Pak-Lok Poon, and Laura L. Pullum, editors, *Proceedings of the 4th International Workshop on Metamorphic Testing, MET@ICSE 2019, Montreal, QC, Canada, May 26, 2019*, pages 2–8. IEEE / ACM, 2019.
6. Nicky Mouha, Mohammad S. Raunak, D. Richard Kuhn, and Raghu Kacker. Finding bugs in cryptographic hash function implementations. *IACR Cryptol. ePrint Arch.*, page 891, 2017.
7. Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking dilithium: Efficient implementation and side-channel evaluation. *IACR Cryptol. ePrint Arch.*, page 394, 2019.
8. Thomas Pornin. New efficient, constant-time implementations of falcon. *IACR Cryptol. ePrint Arch.*, page 893, 2019.
9. Hyeokdong Kwon, Minjoo Sim, Gyeongju Song, Minwoo Lee, and Hwajeong Seo. Evaluating kpqc algorithm submissions: Balanced and clean benchmarking approach. Cryptology ePrint Archive, Paper 2023/1163, 2023. https://eprint.iacr.org/2023/1163.
10. Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Annual international cryptology conference*, pages 537–554. Springer, 1999.
11. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key {Exchange—A} new hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343, 2016.
12. Hayo Baan, Sauvik Bhattacharya, Scott Fluhrer, Oscar Garcia-Morchon, Thijs Laarhoven, Ronald Rietman, Markku-Juhani O Saarinen, Ludo Tolhuizen, and Zhenfei Zhang. Round5: Compact and fast post-quantum public-key encryption. In *Post-Quantum Cryptography: 10th International Conference, PQCrypto 2019,*

Chongqing, China, May 8–10, 2019 Revised Selected Papers 10, pages 83–102. Springer, 2019.

13. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 21–30, 2007.
14. Jelle Don, Serge Fehr, and Christian Majenz. The measure-and-reprogram technique 2.0: Multi-round fiat-shamir and more. In *Annual International Cryptology Conference*, pages 602–631. Springer, 2020.
15. Daniel Kales and Greg Zaverucha. Efficient lifting for shorter zero-knowledge proofs and post-quantum signatures. *Cryptology ePrint Archive*, 2022.
16. David E Muller. Application of boolean algebra to switching circuit design and to error detection. *Transactions of the IRE professional group on electronic computers*, (3):6–12, 1954.
17. Denis X Charles, Kristin E Lauter, and Eyal Z Goren. Cryptographic hash functions from expander graphs. *Journal of CRYPTOLOGY*, 22(1):93–113, 2009.
18. Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. Xmss-a practical forward secure signature scheme based on minimal security assumptions. In *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29–December 2, 2011. Proceedings 4*, pages 117–129. Springer, 2011.
19. Andreas Hülsing. W-ots+–shorter signatures for hash-based signature schemes. In *Progress in Cryptology–AFRICACRYPT 2013: 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings 6*, pages 173–188. Springer, 2013.
20. Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 598–616. Springer, 2009.
21. Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *Annual Cryptology Conference*, pages 40–56. Springer, 2013.
22. Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 206–222. Springer, 1999.
23. Thomas Espitau, Pierre-Alain Fouque, François Gérard, Mélissa Rossi, Akira Takahashi, Mehdi Tibouchi, Alexandre Wallet, and Yang Yu. Mitaka: a simpler, parallelizable, maskable variant of falcon. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 222–253. Springer, 2022.
24. Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996.
25. Nicky Mouha, Mohammad S Raunak, D Richard Kuhn, and Raghu Kacker. Finding bugs in cryptographic hash function implementations. *IEEE transactions on reliability*, 67(3):870–884, 2018.
26. Dimitrios Sikeridis, Panos Kampanakis, and Michael Devetsikiotis. Post-quantum authentication in TLS 1.3: A performance study. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020.* The Internet Society, 2020.

27. Seungyeon Bae, Yousung Chang, Hyeongjin Park, Minseo Kim, and Youngjoo Shin. A performance evaluation of ipsec with post-quantum cryptography. In Seung-Hyun Seo and Hwajeong Seo, editors, *Information Security and Cryptology - ICISC 2022 - 25th International Conference, ICISC 2022, Seoul, South Korea, November 30 - December 2, 2022, Revised Selected Papers*, volume 13849 of *Lecture Notes in Computer Science*, pages 249–266. Springer, 2022.
28. M. Neikes. "TIMECOP: Automated dynamic analysis for timing side-channels", 2020. https://www.post-apocalyptic-crypto.org/timecop/. [Online]. Available.
29. ANSI NIST. C cryptographic api profile for sha-3 candidate algorithm submissions, revision 5, february 2008.
30. Matthias J. Kannwischer, Peter Schwabe, Douglas Stebila, and Thom Wiggers. Improving software quality in cryptography standardization projects. In *IEEE European Symposium on Security and Privacy, EuroS&amp;P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*, pages 19–30, Los Alamitos, CA, USA, 2022. IEEE Computer Society.

Appendix A Benchmarking Result of KEM

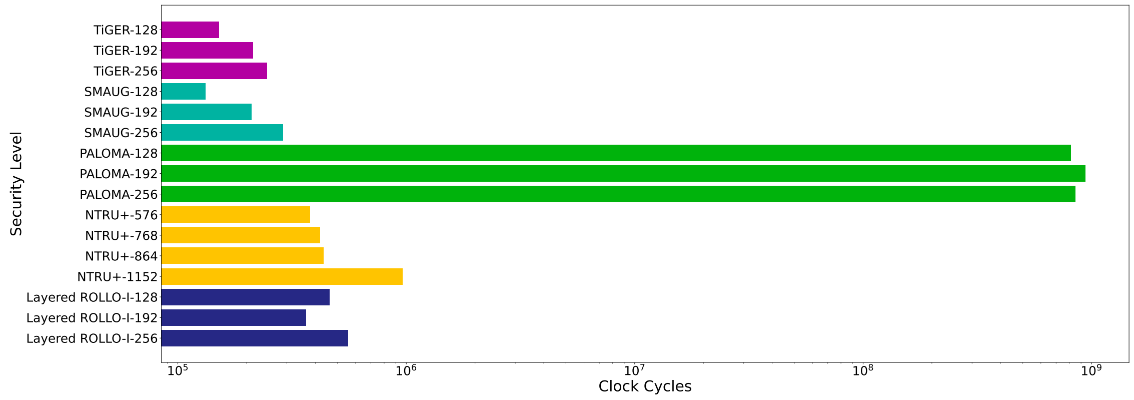

Fig. 7: Benchmarking Results on `keygen()` of KEM

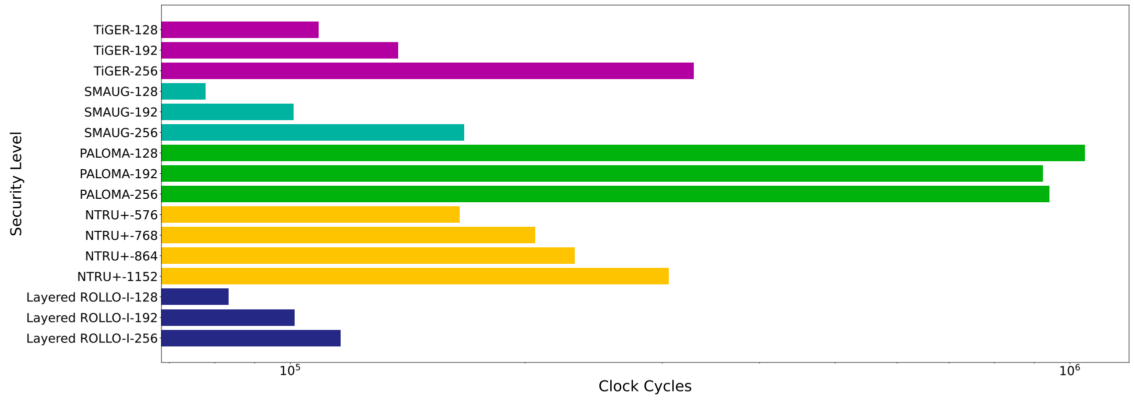

Fig. 8: Benchmarking Results on `encap()` of KEM

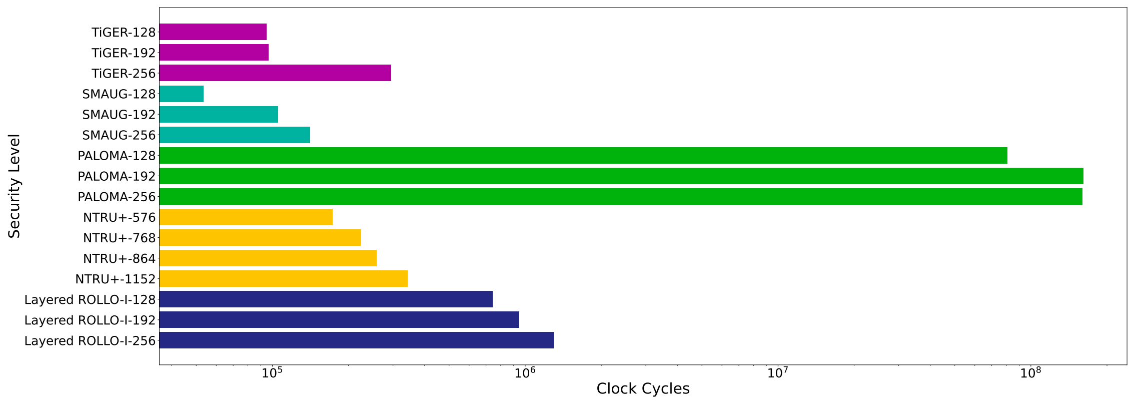

Fig. 9: Benchmarking Results on `decap()` of KEM

Appendix B Benchmarking Result of DSA

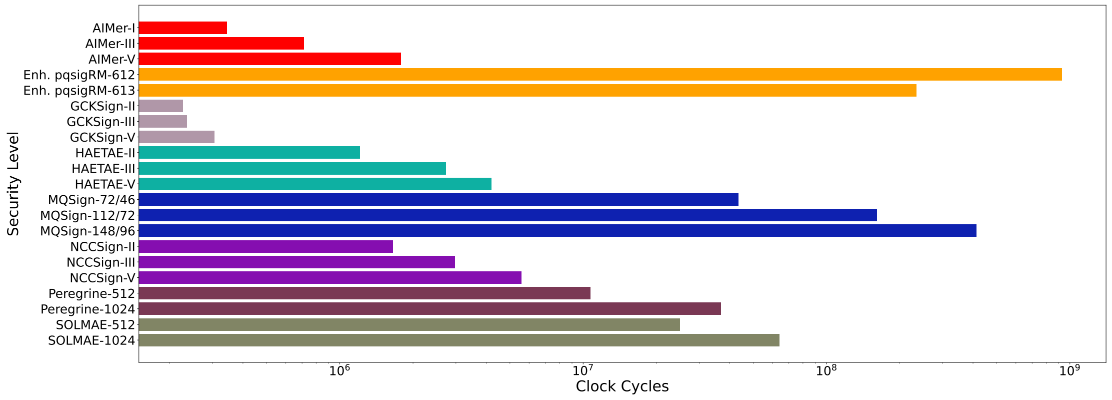

Fig. 10: Benchmarking Results on `keygen()` of DSA

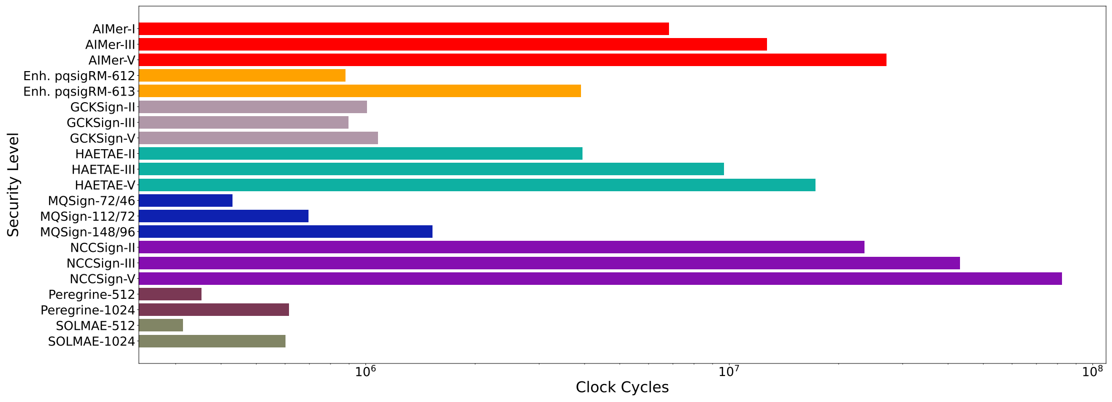

Fig. 11: Benchmarking Results on `sign()` of DSA

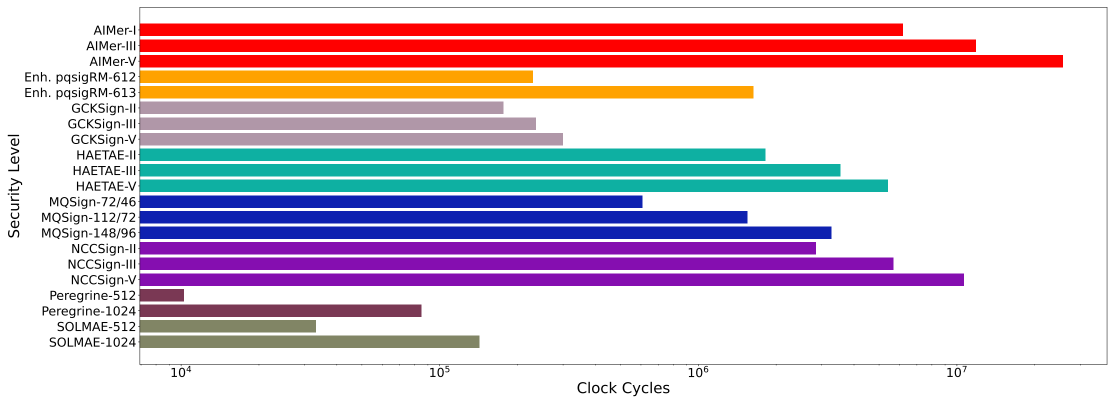

Fig. 12: Benchmarking Results on `verify()` of DSA

Appendix C Profiling Result of DSA

Table 12: Profiling Result of DSA Schemes(1)

Algorithm	Function	Bottlenecks	
AIMer-I	`keygen()`	`keccack`	21.46%
		`transpose_upper_to_lower`	21.05%
		`transpose_lower_to_upper`	18.62%
	`sign()`	`keccack`	56.11%
		`aim128_mpc`	28.51%
		`poly_mul`	6.90%
	`verify()`	`keccack`	38.16%
		`aim128_mpc`	30.21%
		`GF2_128_matmul_vec`	10.53%
Enhanced pqsigRM-612	`keygen()`	`isNonsingular`	4.98%
		`rref`	4.84%
		`inverse`	3.74%
	`sign()`	`vector_mtx_product`	48.58%
		`recursive_decoding_mod`	40.94%
		`randombytes`	4.73%
	`verify()`	`vector_mtx_prodt`	88.03%
		`free`	8.01%
		`hashMsg`	1.85%
GCKSign-II	`keygen()`	`KeccackF1600_StatePermute`	70.96%
		`randombytes`	11.77%
		`rej_uniform`	7.80%
	`sign()`	`KeccackF1600_StatePermute`	72.08%
		`randombytes`	5.52%
		`rej_uniform`	4.91%
	`verify()`	`KeccackF1600_StatePermute`	81.01%
		`rej_uniform`	5.86%
		`ntt`	2.78%

Table 13: Profiling Result of DSA Schemes(2)

Algorithm	Function	Bottlenecks	
HAETAE-II	keygen()	_allrem	89.96%
		poly_naivemul	7.37%
		keccack_squeezeblocks	1.40%
	sign()	_allrem	90.35%
		poly_naivemul	6.87%
		KeccakF1600_StatePermute	1.08%
	verify()	_allrem	90.74%
		poly_naivemul	6.77%
		KeccakF1600_StatePermute	1.15%
MQSign-72/46	keygen()	gf4v_mul_2_u32	24.45%
		gf4v_mul_3_u32	5.06%
		memcpy	4.72%
	sign()	gf4v_mul_2_u32	16.46%
		gf256v_mul_u32	9.07%
		_gf256v_conditional_add_u32	5.96%
	verify()	gf4v_mul_u32	26.64%
		gf4v_mul_2_u32	24.76%
		gf4v_mul_3_u32	4.69%
NCCSign-I	keygen()	mod_add	45.39%
		montgomery_reduce	6.44%
		randombytes	5.20%
	sign()	mod_add	49.37%
		montgomery_reduce	7.12%
		mod_sub	2.95%
	verify()	mod_add	48.17%
		montgomery_reduce	6.99%
		mod_sub	3.09%
Peregrine-512	keygen()	solve_NTRU_intermediate	69.33%
		zint_bezout	8.34%
		make_fg	5.69%
	sign()	mq_NTT	33.30%
		mq_iNTT	18.18%
		modp_NTT2_ext	13.88%
	verify()	mq_NTT	27.13%
		mq_iNTT	28.11%
		mq_poly_tomontymul_ntt	2.86%
SOLMAE-512	keygen()	falcon_inner_FFT	24.28%
		falcon_inner_iFFT	24.81%
		solve_NTRU_intermediate	15.98%
	sign()	falcon_inner_FFT	30.47%
		falcon_inner_iFFT	32.78%
		samplerZ	12.56%
	verify()	falcon_inner_FFT	43.41%
		falcon_inner_iFFT	47.00%
		falcon_inner_poly_mul_fft	6.01%