SaberX4: High-throughput Software Implementation of Saber Key Encapsulation Mechanism

Sujoy Sinha Roy
School of Computer Science, University of Birmingham, United Kingdom
email: *s.sinharoy@cs.bham.ac.uk*

***Abstract*—Saber is a module lattice-based CCA-secure key encapsulation mechanism (KEM) which has been shortlisted for the second round of NIST's Post Quantum Cryptography Standardization project. To attain simplicity and efficiency on constrained devices, the Saber algorithm is serial by construction. However, on high-end platforms, such as modern Intel processors with AVX2 instructions, Saber achieves limited speedup using vector processing instructions due to its serial nature.**

In this paper we overcome the above-mentioned algorithmic bottleneck and propose a high-throughput software implementation of Saber, which we call 'SaberX4', targeting modern Intel processors with AVX2 vector processing support. We apply the batching technique at the highest level of the implementation hierarchy and process *four* Saber KEM operations simultaneously in parallel using the AVX2 vector processing instructions. Our proof-of-concept software implementation of SaberX4 achieves nearly 1.5 times higher throughput at the cost of latency degradation within acceptable margins, compared to the AVX2-optimized non-batched implementation of Saber by its authors.

We anticipate that both latency and throughput of SaberX4 will improve in the future with improved computer architectures and more optimization efforts.

***Index Terms*—public-key cryptography, post-quantum cryptography, lattice-based cryptography, key encapsulation scheme, Saber**

I. Introduction

Public-key cryptography is used in key exchange, authentication and digital signature applications. The most popular public-key algorithms, namely the RSA and ECC cryptosystems, rely on the hardness of integer factorization and elliptic-curve discrete logarithm problems from the Number Theory. These problems cannot be solved using our present-day computers. Quantum computers are a new class of powerful computers which rely on quantum mechanical phenomenon to perform 'certain' tasks significantly faster than our present-day computers. These 'certain' tasks include drug discovery, DNA sequencing, machine learning etc., and also solving the number theoretic problems that are the foundations of RSA and ECC cryptosystems.

Post-quantum cryptography is a new branch of cryptography which develops and studies cryptographic algorithms that are secure against both quantum and classical computers, and can interoperate with existing communications protocols and networks [1]. The mathematical problems behind post-quantum cryptographic algorithms are computationally infeasible for present-day as well as powerful quantum computers. Existing post-quantum cryptographic algorithms rely on code-based, hash-based, lattice-based, multivariate quadratic equation-based and super-singular isogeny-based computation problems. Research in the past 10 years, have resulted in a rich collection of post-quantum cryptographic schemes. In 2017, the American National Institute of Standards and Technology (NIST) initiated 'Post-quantum Cryptography Standardization,' a project to solicit, evaluate, and standardize one or more post-quantum public-key cryptographic algorithms [1]. Of the 69 candidate schemes which were submitted to the first round of the standardization project, 26 have advanced to the second round. The NIST has anticipated that the second round will conclude by June 2020 and the third round will begin. In these two final rounds, the algorithms will be scrutinized for their security and implementation aspects.

Saber [3] is a CCA-secure Key Encapsulation Mechanism (KEM) whose security relies on the hardness of the Module Learning With Rounding problem (MLWR) which is presumed to be a computationally infeasible problem on module lattices. Saber has been designed considering high security, efficiency, flexibility and at the same time simplicity. Saber uses power-of-two modulus which makes modular reduction free of cost and makes the protocol free from the requirement of rejection sampling. However, the use of non-prime modulus precludes the use of asymptotically fastest Number Theoretic Transform (NTT)-based polynomial multiplier which is found to be not a disadvantage as small-degree polynomials can be multiplied efficiently using generic algorithms such as Karatsuba and ToomCook [3]. Software implementations on both high-end [3] and low-end processors [6] have demonstrated efficiency, flexibility and scalability of the Saber algorithm.

Pseudo-random number generation using SHAKE-128 extendable output function [2] occupies a significant portion of Saber's execution time. On platforms with vector processing instructions, pseudo-random number generation using SHAKE-128 can be performed in parallel. For example, Kyber KEM [7] uses the 4x vectorized SHAKE with four seeds to generate pseudo-random strings in parallel. On the contrary, the Saber [4] chooses to use the serial SHAKE-128 to make the implementation of Saber simple and efficient on resource-constrained platforms, e.g., billions of microcontrollers deployed in IoT applications. However, the application of serial SHAKE in Saber, prevents any parallel processing on vector-processors during the generation of pseudo-random strings. For example, the AVX2-optimized implementation of Saber [4] by its authors uses AVX2 instructions to speedup polynomial multiplications only. Whereas, the AVX2-

optimized implementation of Kyber [7] uses AVX2 instructions extensively to speedup both polynomial multiplication and pseudo-random number generation.

Contributions: In this paper we overcome the above-mentioned serial-execution bottleneck during the computation of Saber KEM by applying parallel processing at the highest level of the implementation hierarchy, i.e., the application-level, where the Saber KEM protocol is executed. We 'batch' four Saber KEM operations into a single process and execute them in parallel using 256-bit AVX2 vector processing instructions on modern Intel processor. We use the AVX2 implementation of SHAKE-128, known as SHAKE-128x4 [2], to compute four SHAKE-128 operations in parallel, thus producing four pseudo-random strings for the four KEM operations simultaneously. In addition to that, our high-level parallel processing approach is able to compute the SHA3 hash functions for the four KEM operations in parallel using AVX2 instructions. On an Intel Core i5 7th generation processor, our batched implementation achieves nearly 1.5 times higher throughput compared the implementation of Saber by its authors. The source codes of SaberX4 are available at https://github.com/sujoyetc/SaberX4.

II. Preliminaries

A. Notation

The ring of integers modulo q is denoted as $\mathbb{Z}_q$ where the integers are in $[0, q)$. The modular reduction operation $z \bmod q$ reduces an integer z in $[0, q)$. The quotient polynomial ring is denoted as $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ where n is a power-of-two. Thus polynomials in R_q are of n coefficients where each coefficient is in $[0, q)$. The notations R^l and $R^{l\times k}$ denote the ring of l-vectors and $l \times k$-matrices over R respectively. Polynomials in R_q are denoted in lower case letters and matrices/vectors are denoted in bold lower/upper case letters.

For a probability distribution χ over a set S, the notation $z \leftarrow \chi$ is used to denote sampling $z \in S$ according to the distribution χ. Similarly, $\boldsymbol{Z} \leftarrow \chi(R_q^{l\times k})$ denotes sampling the matrix $\boldsymbol{Z} \in R_q^{l\times k}$, where all coefficients have been sampled from χ. To indicate a deterministic sampling of matrix $\boldsymbol{Z}$ starting from a random seed r, the notation $\boldsymbol{Z} \leftarrow \chi(R_q^{l\times k}; r)$ is used. The uniform distribution is denoted as $\mathcal{U}$ and a centered binomial distribution with the parameter μ, which is an even number, is denoted as β_μ.

The bitwise left and right shift operations are denoted using the operators $\ll$ and $\gg$ respectively. When these operators are applied on polynomials or matrices, they are actually applied on the coefficients.

Saber uses a hierarchy of algorithms. First, a Chosen Plaintext Attack (CPA) secure public-key encryption scheme is realized whose security is based on the Module Learning with Rounding (MLWR) problem. Next, a post-quantum variant of the Fujisaki-Okamoto transform [5] is applied to implement Chosen Ciphertext Attack (CCA) secure Key Encapsulation Mechanism (KEM) scheme. In the following, we describe the algorithms used in CPA-secure 'Saber Public Key Encryption' and CCA-secure 'Saber Key Encapsulation'.

B. Saber Public Key Encryption

The CPA-secure key generation, encryption and decryption algorithms used in Saber [3] are shown in Alg. 1, 2 and 3 respectively. Saber uses two moduli $q = 2^{13}$ and $p = 2^{10}$ with bit-lengths $\epsilon_q = 13$ and $\epsilon_p = 10$ respectively. Key generation uses a 32 byte random seed $seed_{\boldsymbol{A}}$ and then expands it using the extendable output function SHAKE-128 inside $\texttt{gen}()$ in Alg. 1 to construct the pseudo-random matrix $\boldsymbol{A}$. The secret vector $\boldsymbol{s}$ is sampled from a binomial distribution β_μ with parameter μ. It consists of l polynomials. The key generation operation generates the public-key *pk* and secret-key *sk*.

The encryption operation in Alg. 2 uses matrix generation, binomial sampling, matrix-vector multiplication. The decryption operation computes vector-vector multiplication.

Algorithm 1 Saber.PKE.KeyGen() [4]

$seed_{\boldsymbol{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{A} = \texttt{gen}(\text{seed}_{\boldsymbol{A}}) \in R_q^{l\times l}$
$r = \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{s} = \beta_\mu(R_q^{l\times 1}; r)$
$\boldsymbol{b} = ((\boldsymbol{A}^T\boldsymbol{s} + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l\times 1}$
Return $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), sk := (\boldsymbol{s}))$

Algorithm 2 Saber.PKE.Enc$(pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), m \in R_2; r)$ [4]

$\boldsymbol{A} = \texttt{gen}(\text{seed}_{\boldsymbol{A}}) \in R_q^{l\times l}$
If (r is not specified)
  $r = \mathcal{U}(\{0,1\}^{256})$
$\boldsymbol{s'} = \beta_\mu(R_q^{l\times 1}; r)$
$\boldsymbol{b'} = ((\boldsymbol{A}\boldsymbol{s'} + \boldsymbol{h}) \bmod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l\times 1}$
$v' = \boldsymbol{b}^T(\boldsymbol{s'} \bmod p) \in R_p$
$c_m = (v' + h_1 - 2^{\epsilon_p - 1}m \bmod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
Return $c := (c_m, \boldsymbol{b'})$

Algorithm 3 Saber.PKE.Dec$(sk = \boldsymbol{s}, c = (c_m, \boldsymbol{b'}))$ [4]

$v = \boldsymbol{b'}^T(\boldsymbol{s} \bmod p) \in R_p$
$m' = ((v - 2^{\epsilon_p - \epsilon_T}c_m + h_2) \bmod p) \gg (\epsilon_p - 1) \in R_2$
Return m'

C. Saber Key-Encapsulation Mechanism

Saber KEM is the key-encapsulation mechanism consisting of three algorithms as described in Algorithms 4, 5 and 6 respectively. The readers may follow the original Round2 documentation of Saber [4] for detailed description and provable security of these algorithms.

Algorithm 4 Saber.KEM.KeyGen() [4]

$(seed_{\boldsymbol{A}}, \boldsymbol{b}, \boldsymbol{s}) = \texttt{Saber.PKE.KeyGen}()$
$pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$
$pkh = \mathcal{F}(pk)$
$z = \mathcal{U}(\{0,1\}^{256})$
Return $(pk := (seed_{\boldsymbol{A}}, \boldsymbol{b}), sk := (\boldsymbol{s}, z, pkh))$

Algorithm 5 Saber.KEM.Encaps($pk = (seed_{\boldsymbol{A}}, \boldsymbol{b})$) [4]

$m \leftarrow \mathcal{U}(\{0,1\}^{256})$
$(\hat{K}, r) = \mathcal{G}(\mathcal{F}(pk), m)$
$c = \texttt{Saber.PKE.Enc}(pk, m; r)$
$K = \mathcal{H}(\hat{K}, c)$
Return (c, K)

Algorithm 6 Saber.KEM.Decaps($sk = (\boldsymbol{s}, z, pkh), pk = (seed_{\boldsymbol{A}}, \boldsymbol{b}), c$) [4]

$m' = \texttt{Saber.PKE.Dec}(\boldsymbol{s}, c)$
$(\hat{K}', r') = \mathcal{G}(pkh, m')$
$c' = \texttt{Saber.PKE.Enc}(pk, m'; r')$
If $(c = c')$
  Return $K = \mathcal{H}(\hat{K}', c)$
Else
  Return $K = \mathcal{H}(z, c)$

III. AVX2 Implementation of Saber

A. Symmetric-key building blocks

In the algorithms of Saber, symmetric-key cryptographic primitives such as hash functions $\mathcal{G}$ and $\mathcal{H}$ and an extendable output function inside gen() are used. Saber uses SHA3-512 and SHA3-256, which were standardized in FPIS202 [2], to implement the hash functions $\mathcal{G}$ and $\mathcal{H}$ respectively. For generating pseudo-random numbers, Saber uses SHAKE-128 [2] as the extendable output function. Note that SHA3-512, SHA3-256 and SHAKE-128 are wrappers around the Keccak core [2]. On vector processing platforms, such as Intel AVX2, four Keccak function calls can be computed in parallel using the four 64-bit slots of the 256-bit vector processor. However, Saber uses the serial SHAKE-128 and generates the pseudo-random strings sequentially, thus making no use of the vector processor in this step. The authors of Saber chose the serial SHAKE-128 to make its implementation easier and efficient on constrained platforms [3].

B. Polynomial multiplication

Saber performs matrix-vector and vector-vector multiplications where the elements are polynomials in R_q. Hence, a significant portion of the overall computation time is spent in performing polynomial multiplications in R_q. Saber relies on generic polynomial multiplication algorithms such as Karatsuba or ToomCook. In [4] the authors of Saber proposed an AVX2-optimized software implementation of ToomCook polynomial multiplication and implemented a high-speed software implementation of Saber.

IV. SaberX4: High-Throughput AVX2 Implementation of Saber

In this section we propose our SaberX4 software which aims at increasing the number of Saber KEM operations per second, i.e., the throughput at the cost of minimum latency degradation. Let us consider a server machine which is computing numerous key exchange operations. For such a busy server, the speed of post-quantum key exchange is an extremely important issue as the server must be able to process several thousands of key exchange operations every second. When optimizing throughput of KEM, care must be taken so that the latency of individual KEM must not degrade too much to make the implementation impractical.

A. Pseudo-random number generation using SHAKE-128

SHAKE-128 uses Keccak which is a strong one-way function. Hence, it is not possible to parallelize the serial pseudo-random number generation in Saber by following standard low-level parallel processing tricks.

We find that the simple-and-serial structure of the Saber KEM algorithm actually opens the avenues for applying parallel processing at the highest layer of the implementation hierarchy, i.e., the application layer where KEM protocols are executed. We propose to execute *multiple* Saber KEMs in a batch using vector processing instructions. On AVX2 platforms (vectors of 256-bit width) we batch *four* Saber KEM operations together as we use the AVX2-optimized SHAKE-128x4 [2], which is the batch of *four* SHAKE-128 calls.

B. Hash computations using SHA3

During the CCA transform steps in Saber KEM, the SHA3-256 and SHA3-512 hash functions are computed several times. Similar to SHAKE-128, these Keccak-based hash functions are also computationally expensive. By batching four Saber KEM operations together, we are able to batch these hash function calls too: four hash values are computed in parallel for the four Saber KEM operations.

C. Polynomial multiplication

To compute the polynomial multiplications for the four batched Saber KEM operations, we consider two approaches that could utilize the vector processing instructions. The first approach is to compute the polynomial multiplications in parallel, each using a 64-bit slot. The second approach is to compute the polynomial multiplications for the batched KEMs serially: 1) one KEM from the batch occupies all the slots of the vector processor to compute its polynomial multiplication and then 2) the next KEM from the batch occupies the vector processor and computes its polynomial multiplication.

In this work, we choose the second approach as we could use the existing AVX2-optimized implementation of polynomial multiplication [4] by the authors of Saber. Their implementation packs 16 coefficients of a polynomial in the 16 slots (each 16-bit) of a AVX2 vector. Packing from integers to AVX2 vectors and the vice-versa are performed only at the beginning or at the end of a polynomial multiplication. Thus, excluding the pacing/unpacking operations, the polynomial multiplications are computed using AVX2 instructions.

D. Other building blocks

The saber algorithm uses several other building blocks namely, message encoder, message decoder, byte-string-to-polynomial and polynomial-to-byte-string conversion, binomial sampling, etc. These building blocks are relatively easier

to compute compared to SHAKE-128, SHA3 hash functions and polynomial multiplication. The implementation of Saber [4] by its authors describes these building blocks in standard C language and does not attempt to use vector processing instructions. In our implementation, we borrow these building blocks from the implementation of Saber. These small operations are thus executed serially in our implementation.

V. Results

We implemented SaberX4 using both C language and Intel AVX2 intrinsics. No assembly optimizations were explored in any of the public-key primitives. We compiled the software implementation using `gcc-5.5` with optimization flags `-O3 -fomit-frame-pointer -march=native -std=c99`. The computation times were measured on a Dell laptop with Ubuntu 16.04 operating system and Intel(R) Core(TM) i5-7200U CPU running at 2.5GHz. All the measurements were taken after disabling hyper-threading and Turbo-Boost. In Table I we present the performance results for our implementation and compare our results with the AVX2-optimized implementations of Saber [4] and Kyber [7].

In the table, the batched implementation of Saber is represented as 'SaberX4' to indicate the batching of four KEM operations. With respect to the non-batched implementation of Saber [4], our implementation achieves around 38%, 45% and 35% higher throughput for key generation, encapsulation and decapsulation operations respectively. However, latencies of the individual KEM operations increase roughly by a factor of 3 with respect to the non-batched implementation of Saber [4]. The latency degradation is well within the acceptable limits as individual KEM operations require below 0.15 ms.

The Kyber KEM algorithm [7] has been designed to leverage from the vector processing instructions on modern high-end platforms. The AVX2 and assembly-optimized implementation of Kyber KEM uses vector processing most of the time to achieve fast computation time. Thus Kyber has lower latency compared to Saber on Intel AVX2 platforms. In Table I we see that the encapsulation operation in SaberX4 achieves slightly higher throughput compared to the same of Kyber; whereas the key generation and decapsulation operations in SaberX4 are slightly slower than those of Kyber.

We would like to remark that there is not much scope for increasing the throughput of Kyber by applying similar batching technique. The computationally-expensive building blocks in Kyber are already parallel in nature and are computed using AVX2 instructions.

Theoretically, SaberX4 is expected to achieve nearly four times higher throughput compared to Saber. In practice, we obtain less than 1.5 times higher throughput. One reason behind this is that, vector processing has its own overhead. With improved computer architecture, this overhead is likely get smaller in the future. Additionally, in this work, our goal was to see if the idea of batching works in practice by having a *proof-of-concept* implementation. In the implementation phase, we tried to maximize code reuse from the existing library of Saber [4]. Some of the routines, such as message encoder, message decoder, byte-string-to-polynomial and polynomial-to-byte-string conversions, binomial sampling, etc. are still executed serially, although they can be batched to achieve higher throughput.

TABLE I
Performances on Intel AVX2 Platform

Scheme	Operation	Latency (cycles)	Throughput (ops/sec)
SaberX4	Key Generation	296,582	33,717
`AVX2 optimized`	Encapsulation	334,769	29,871
`Batched`	Decapsulation	354,417	28,215
Saber [4]	Key Generation	102,870	24,302
`AVX2 optimized`	Encapsulation	121,113	20,641
	Decapsulation	119,986	20,835
Kyber768 [7]	Key Generation	69,599	35,920
`AVX2 and ASM`	Encapsulation	89,723	27,863
`optimized`	Decapsulation	75,495	33,114

VI. Conclusions

In this paper we proposed a batching technique to realize a high-throughput software implementation of Saber KEM, which we call 'SaberX4', targeting high-end processors with AVX2 vector processing instructions. With a proof-of-concept implementation, we showed that the application-level batching technique maximizes the utilization of vector processor thus offering nearly 1.5 times higher throughput for the Saber KEM algorithm. As a future work, we would consider optimizing the current proof-of-concept implementation of SaberX4 to improve its throughput as well as latency.

References

[1] "NIST Post-Quantum cryptography Round 2 submissions," 2019. [Online]. Available: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions

[2] National Institute of Standards and Technology. 2015., "SHA-3 standard: Permutation-Based Hash and Extendable-Output Functions," FIPS PUB 202, 2015.

[3] J.-P. DÁnvers, A. Karmakar, S. S. Roy, and F. Vercauteren, "Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM," Cryptology ePrint Archive, Report 2018/230, 2018, https://eprint.iacr.org/2018/230, Africacrypt 2018.

[4] J.-P. DÁnvers, A. Karmakar, S. S. Roy, and F. Vercauteren, "Saber," Proposal to NIST PQC Standardization, Round2, 2019, https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/round-2-submissions.

[5] D. Hofheinz, K. Hvelmanns, and E. Kiltz, "A Modular Analysis of the Fujisaki-Okamoto Transformation," Cryptology ePrint Archive, Report 2017/604, 2017, http://eprint.iacr.org/2017/604.

[6] A. Karmakar, J. M. B. Mera, S. S. Roy, and I. Verbauwhede, "Saber on ARM CCA-secure module lattice-based key encapsulation on ARM," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 3, pp. 243–266, 2018. [Online]. Available: https://doi.org/10.13154/tches.v2018.i3.243-266

[7] P. Schwabe, R. Avanzi, J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, G. Seiler, and D. Stehle, "Crystals-kyber," Proposal to NIST PQC Standardization, Round2, 2019, https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/round-2-submissions.