Saber Post-Quantum Key Encapsulation Mechanism (KEM): Evaluating Performance in Mobile Devices and Suggesting Some Improvements

Leonardo A. D. S. Ribeiro[1], José Paulo da Silva Lima[1], Ruy J. G. B. de Queiroz[1], Amirton Bezerra Chagas[1], Jonysberg P. Quintino[1], Fabio Q. B. da Silva[1], Andre L. M. Santos[1] and José Roberto Ribeiro Junior [2]

[1]*Centro de Informática, Universidade Federal de Pernambuco, Av. Jornalista Anibal Fernandes, s/n, Cidade Universitária (Campus Recife), CEP: 50.740-560 - Recife - PE - Brasil*

[2]*Samsung Eletrônica da Amazônia LTDA, Rua Thomas Nielsen Júnior n. 150, Parque Imperador, CEP 13097-105, Campinas -SP-Brasil*

ladsr@cin.ufpe.br, jpsl3@cin.ufpe.br, ruy@cin.ufpe.br, abc@cin.ufpe.br, jpq@cin.ufpe.br, fabio@cin.ufpe.br, alms@cin.ufpe.br, jose.junior@samsung.com

Keywords: Post-Quantum Cryptography. Lattice Based Cryptography. Saber Code Evaluation and Improvements. x64 and ARM architectures

Abstract: Saber is one of the four finalists in the ongoing NIST Post-Quantum Cryptography Standardization Process. It is one of the three finalists that are based on lattice problems. This article intends to show the results of an analysis of Saber (Vercauteren, 2018) performance in x64 and ARM architectures. Saber was tested following a scenario where first a pair of public and private keys were generated. Then, a data representing a session key is encrypted using the generated public key and decrypted using the generated private key. Algorithm performance was evaluated running these steps in each architecture proposed. Based on the data collected, it is checked if Saber is suitable to mobile devices or not. Bottlenecks were found while executing Saber code. Also some improvements were proposed to its code.

1 INTRODUCTION

The goal of the research was to evaluate the reference implementation of one of finalists of NIST Post-Quantum Cryptography Standardization Process and find out if it is suitable to be used by an application running in an Android mobile environment based on ARMv8 architecture. Saber was the chosen algorithm in the list of finalists to be analyzed, because it has shown good performance in mobile devices according to some recent evaluations:(Saarinen, 2019) and (Xu et al., 2018). Saber evaluations were executed in ARM and x64 architectures and it was chosen security level Saber to make these analysis. Usually an x64 architecture has better performance compared to an ARM architecture in usage of cryptography algorithms, so comparing them may point out good ARM performance. The Saber code was evaluated using the versions submitted to round two and round three of the NIST standarditazion process.

2 EVALUATING REAL-WORLD PERFORMANCE DATA

The idea is to perform an analysis of NIST PQC Round 3 candidate Saber performance in x64 and ARM architectures. The reference implementation of this KEM candidate was tested following a scenario where first a pair of public and private keys was generated and then data representing a session key was encrypted using the public key generated and decrypted using the corresponding private key. Algorithm performance were evaluated running these steps in each target architecture. Based on the collected data, the evaluation sought to check whether Saber was suitable to be deployed in mobile devices. Bottlenecks were found in code according to performance analysis, and, as a consequence, some improvements were proposed to reference implementation code.

A recent survey of design and implementation aspects of post-quantum cryptographic algorithms (Howe et al., 2019) draws the attention to the fact that every "real-world efforts to deploy post-quantum cryptography will have to contend with new, unique

problems”. By experimenting with the reference implementation of Saber, we have tried to embark on a cryptographic engineering task which may bring insight into the real world deployment of such a candidate for standardization. Such an enterprise is likely to face peculiar circumstances whose evaluation may prove useful in the real world. Again, Howe et al. point out that specific cases “may require a diverse combination of computational assumptions woven together into a single hybrid scheme”, as well as “special attention to physical management of sensitive state”, not to mention “very unbalanced performance profiles, requiring distinct solutions for different application scenarios”.

In this spirit, the intention here is to gather concrete performance data for real implementations in resource-constrained (memory, processing speed, energy consumption, etc.) devices, in particular, embedded devices and mobile application running in platforms such as iOS and Android (scope of cell phones and tablets, with resource restrictions, battery, memory, etc.), and perform code improvements wherever appropriate.

3 ROUND TWO CODE EVALUATION

SABER was one of the seventeen public-key encryption and key-establishment algorithms chosen by NIST on round two of standardization process. The Saber’s algorithm code submitted in this round was evaluated according to its performance in two architectures: x64 and ARM. To evaluate SABER, a Java code was built to generate a pair of public and private keys and to use this pair of keys to encrypt and decrypt a specific data using this keys pair. This Java code was tested first in the x64 architecture to establish a performance baseline. To test in ARM environment, it was necessary to build an auxiliary Android application to call the developed Java code ported to this mobile architecture. The performance evaluation was done through execution of thirty rounds of a sequence of key generation, encryption and decryption of a specific data. The data used in tests represents a possible 228 bytes session key to be used in a secure communication. To enable Java code to communicate with SABER code built in C language, it was used Java Native Interface (JNI).

3.1 x64 Architecture

The tests were performed in a virtualized Kali Linux version 2019.1 with 2.3 GB of RAM memory, 20 GBytes of hard disk memory and Intel processor Core I3-6006U 2GHz. The code was profiled using print messages in the execution sequence of key generation, encryption and decryption. The profiling messages were stored in a file, thus allowing evaluation of SABER upon profile information with the statistical analysis of this stored data.The consumption times of functions related to key generation(indcpa-kem-keypair), encryption(indcpa-kem-enc) and decryption(indcpa-kem-dec) were plotted in graphs showing the minimum and maximum values (Figures 1, 2 and 3). The Key generation, encryption and decryption had average times respectively: 1458.00, 1584.04 and 382.43 microseconds. In addition, the time consumption was calculated for each function called in key generation, encryption and decryption (Figures 4, 5 and 6). In the key generation, the function ”MatrixVectorMul” consumed 81% of the process time. In encryption, the function ”MatrixVectorMul” consumed 59% of the process time, and in decryption, ”InnerProd” consumed 95% of the process time.

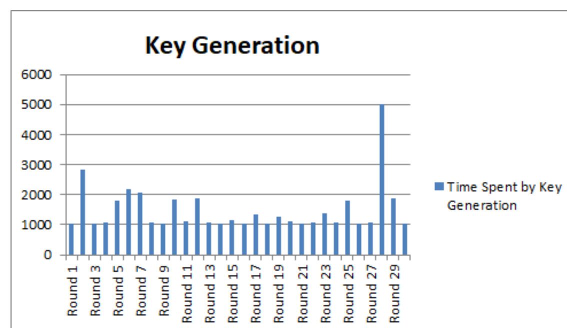

Figure 1: Key Generation Performance for x64

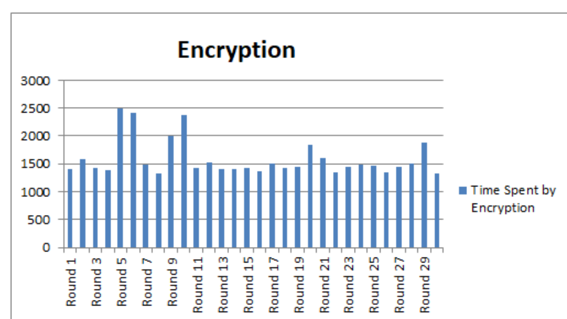

Figure 2: Encryption Performance for x64

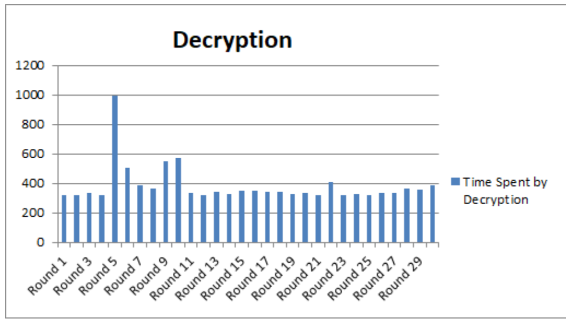

Figure 3: Decryption Performance for x64

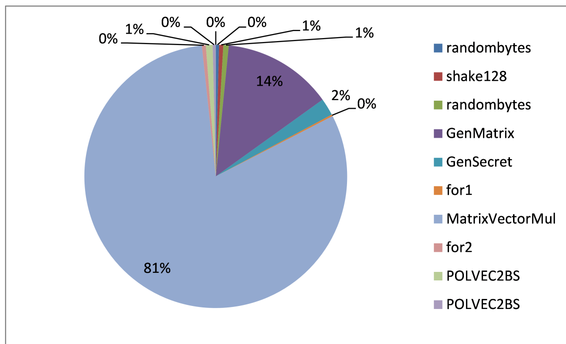

Figure 4: Functions Usage in Key Generation for x64

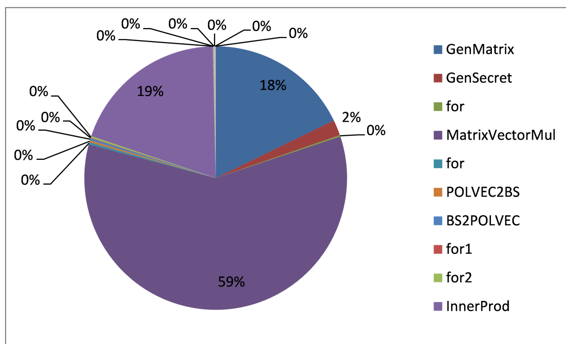

Figure 5: Functions Usage in Encryption for x64

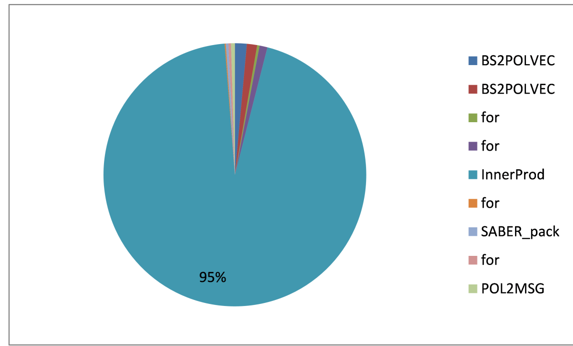

Figure 6: Functions Usage in Encryption for x64

3.2 ARM Architecture

The tests were performed in a ARM device, specifically in a Samsung Galaxy S20 device. It was necessary to build a specific Android application to enable calling the Java code used to evaluate Saber code in x64 architecture. To enable running of Saber code in a ARM environment, a compiled version of OpenSSL [1] for ARM architecture was downloaded and added to the Android application. Application code was profiled and executed thirty times. In each time, the Java profiling code was called and data was collected providing key generation, encryption and decryption consumption times. They were plotted in graphs for analysis(Figures 7, 8 and 9) and had average times respectively of: 894.20, 753.70 and 211.09 microseconds. Time consumption was calculated for each function called in key generation, encryption and decryption(Figures 10, 11 and 12). In the key generation, the function "MatrixVectorMul" consumed 67% of the process time. In encryption, the function "MatrixVectorMul" consumed 40% of the process time. And in the decryption, the function "InnerProd" consumed 88% of the process time.

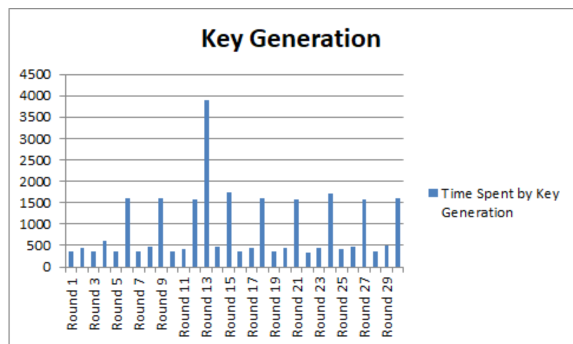

Figure 7: Key Generation Performance for ARM

[1] https://github.com/PurpleI2P/OpenSSL-for-Android-Prebuilt

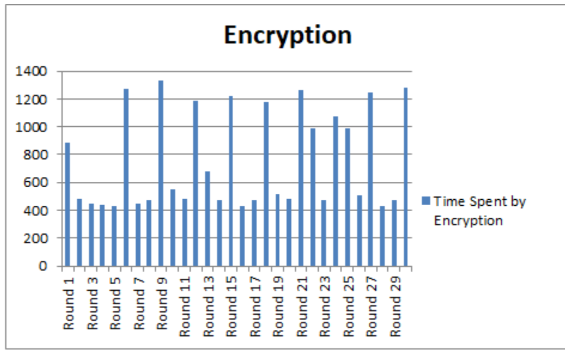

Figure 8: Encryption Performance for ARM

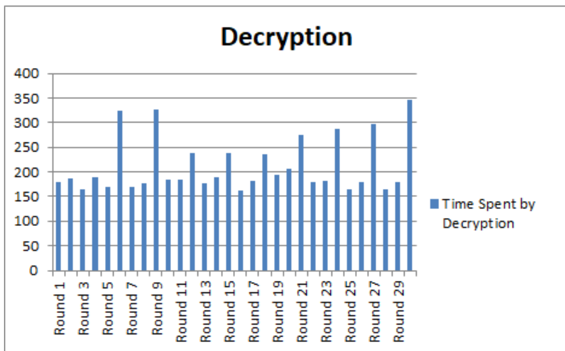

Figure 9: Decryption Performance for ARM

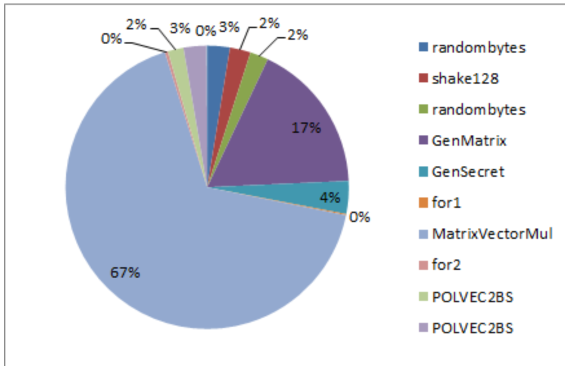

Figure 10: Functions Usage in Key Generation for ARM

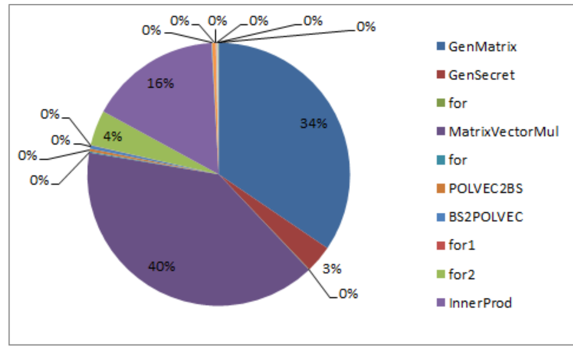

Figure 11: Functions Usage in Encryption for ARM

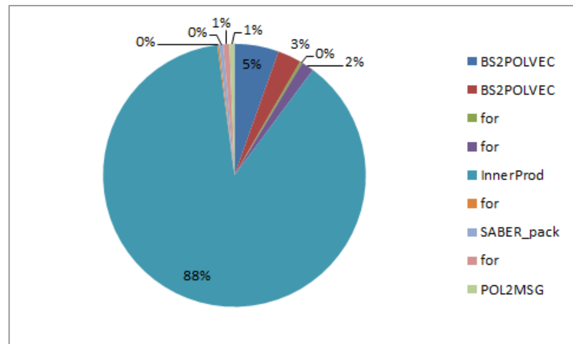

Figure 12: Functions Usage in Encryption for ARM

3.3 X64 versus ARM Architectures

The consumption times of x64 and ARM architectures were plotted in a graph to compare them (Figure 13). From the analysis of the graphs, it can be concluded that round two Saber standard code is more suitable to ARM than x64 architectures. In addition, it was shown that the key generation and encryption have higher average performance time. It was also found that the function "MatrixVectorMul" has the highest cost in key generation and encryption steps and the function "InnerProd" has higher cost in decryption step.

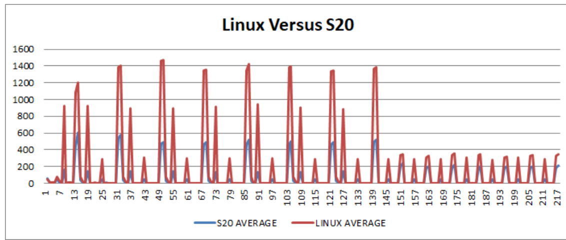

Figure 13: Linux versus S20 Performance in Round Two

4 ROUND THREE CODE EVALUATION

As a result of the round two, SABER was one of the three public-key encryption and key-establishment algorithms based on lattices chosen by NIST to be part on the round three of standardization contest. The algorithm owner submitted code improvements in this round and it was available for download to anyone who wanted to evaluate algorithm. Upon this new code, SABER was still evaluated according to its performance in x64 and ARM architectures. The same Java and Android applications used to evaluate round two's code were used to evaluate the Saber round three code. The performance evaluation was done in a more robust way by execution of thirty rounds with each one executing one thousand times the sequence of key generation, encryption and decryption of the same data used for testing the round's two tests.

4.1 x64 Architecture

The tests were performed in a computer with Ubuntu 20.04 LTS version with 8 GB of memory and processor Intel(R) Core(TM) i7-6700-3.4GHz. The sequence of key generation, encryption and decryption of a session key were profiled in a specific file. Upon analysis of this file, performance times calculated for key generation, encryption and decryption were plotted in a set of graphs (Figures 14, 15 and 16) and their averages times were respectively: 1970.18, 2435.74 and 574.68 microseconds. In addition, the percentage of functions usage was calculated for the key generation, encryption and decryption (Figures 17, 18 and 19). In key generation, the function "MatrixVectorMul" consumed 86 % of the process time, in encryption, the function "MatrixVectorMul" consumed 68% of the process time and in decryption, the function "InnerProd" consumed 96% of the process time.

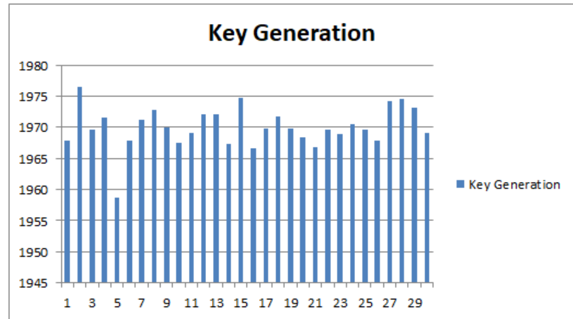

Figure 14: Key Generation Performance for x64

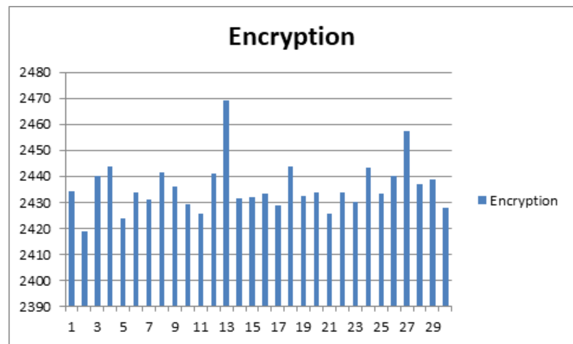

Figure 15: Encryption Performance for x64

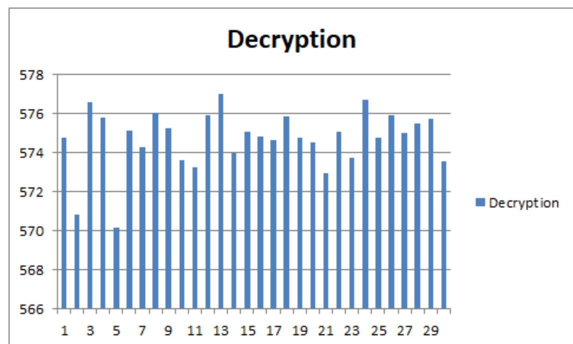

Figure 16: Decryption Performance for x64

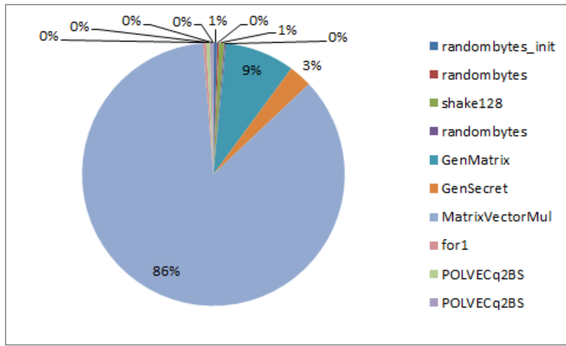

Figure 17: Functions Usage in Key Generation for x64

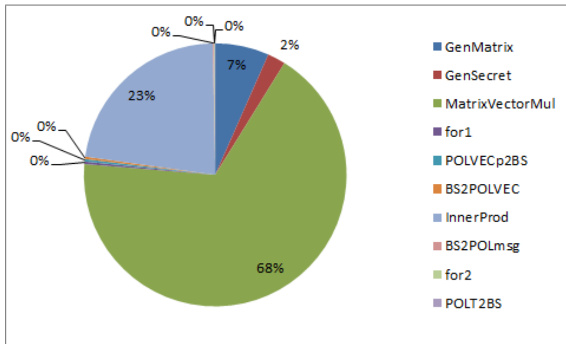

Figure 18: Functions Usage in Encryption for x64

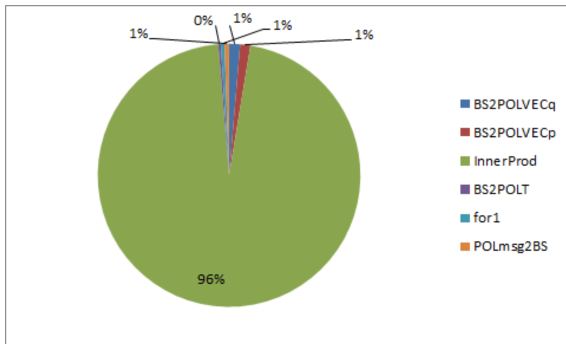

Figure 19: Functions Usage in Encryption for x64

4.2 ARM Architecture

The tests were performed in a smartphone Samsung Galaxy S20. The sequence of key generation, encryption and decryption of a session key were profiled in a specific file. Upon analysis of this file, performance times calculated for key generation, encryption and decryption were plotted in a set of graphs (figures 20, 21 and 22). Key generation, encryption and decryption had averages respectively: 333.96, 355.25 and 128.25 microseconds. It was also calculated percentage of functions usage for key generation, encryption and decryption (Figures 23, 24 and 25). In key generation, the function "MatrixVectorMul" consumed 64 % of the process time, in encryption, the function "MatrixVectorMul" consumed 55% of the process time and in decryption, the function "InnerProd" consumed 89% of the process time.

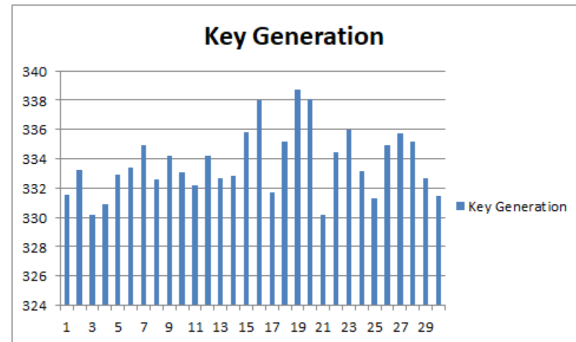

Figure 20: Key Generation Performance for ARM

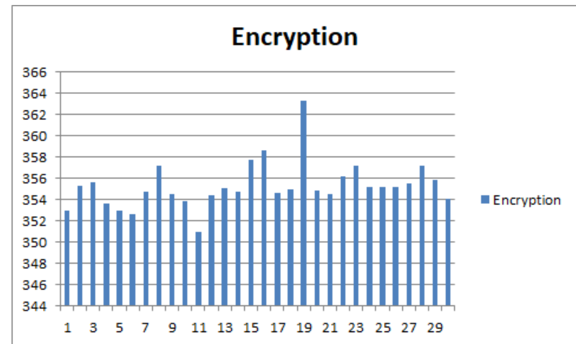

Figure 21: Encryption Performance for ARM

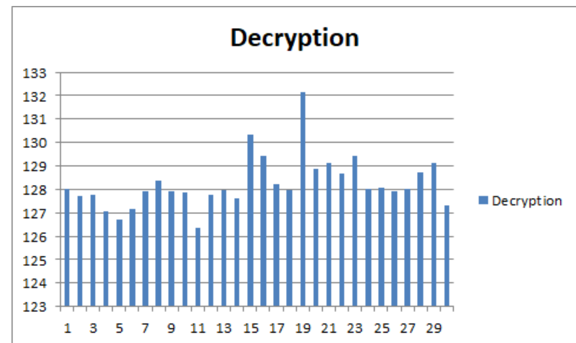

Figure 22: Decryption Performance for ARM

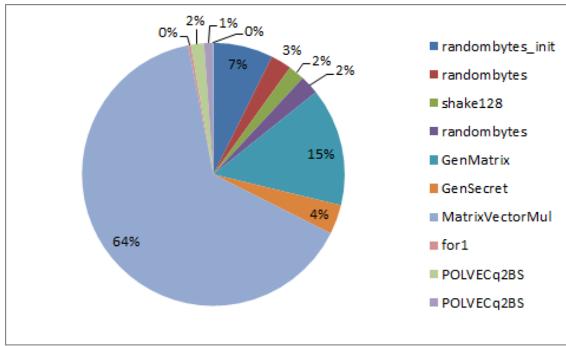

Figure 23: Functions Usage in Key Generation for ARM

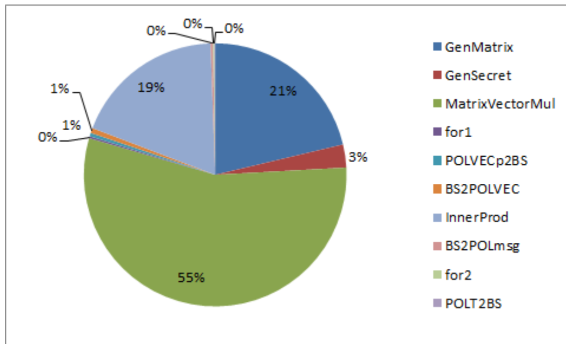

Figure 24: Functions Usage in Encryption for ARM

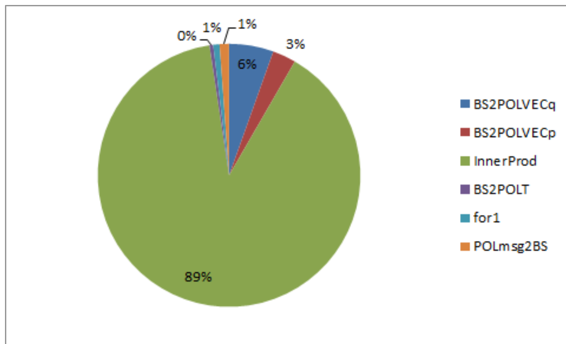

Figure 25: Functions Usage in Encryption for ARM

4.3 x64 versus ARM Architectures

The consumption times of x64 and ARM architectures were plotted in a graph to compare them (Figure 26). It was also created a table comparing performance of functions called by Saber code (table 1). Analysing performance of round three code it can be checked that x64 architecture had worst results compared to round two code. It can also be checked that ARM architecture had better results in round three code compared to round two code. Analysing the Figure 26 and Table 1, it was also concluded that Saber is more suitable to ARM architectures for almost all steps of key generation, encryption and decryption, with exception to random number generation that in x64 architectures had better performance (x64 was better four times and ARM twenty six times in the total of thirty interactions).

Table 1: x64 versus ARM Averages Comparison

Function	ARM	x64	ARM - x64	Faster
randombytes_init	16.31	9.44	6.86	X64
randombytes	5.64	4.22	1.42	X64
shake128	4.45	12.72	-8.27	ARM
randombytes	5.13	3.85	1.28	X64
GenMatrix	32.14	167.61	-135.47	ARM
GenSecret	8.01	12.72	-47.84	ARM
MatrixVectorMul	142.60	1678.03	-1535.43	ARM
for1	0.772	7.88	-7.12	ARM
POLVECq2BS	3.55	8.98	-5.42	ARM
POLVECq2BS	2.37	7.63	-5.26	ARM
memcpy	0.15	0.11	0.04	X64
indcpa_kem_keypair	334.199	1969.69	-1635.5	ARM
GenMatrix	31.4445	160.21	-128.77	ARM
GenSecret	7.06	53.50	-46.44	ARM
MatrixVectorMul	137.12	1631.43	-1494.31	ARM
for1	0.654	12.72	-6.97	ARM
POLVECp2BS	1.13	7.07	-5.94	ARM
BS2POLVEC	3.05	7.14	-4.09	ARM
InnerProd	46.35	543.84	-497.48	ARM
BS2POLmsg	3.23	2.83	-0.39	ARM
for2	0.56	2.58	-2.02	ARM
POLT2BS	0.86	1.70	-0.84	ARM
indcpa_kem_enc	363.62	2433.71	-2070.10	ARM
BS2POLVECq	2.866	7.58	-4.71	ARM
BS2POLVECp	1.52	7.07	-5.55	ARM
InnerProd	46.94	544.12	-497.18	ARM
BS2POLT	0.23	1.77	-1.53	ARM
for1	0.41	2.58	-2.18	ARM
POLmsg2BS	0.60	3.09	-2.49	ARM
indcpa_kem_dec	127.94	574.89	-446.95	ARM

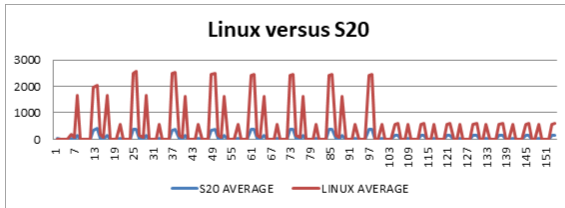

Figure 26: Linux versus S20 Performance in Round Three

4.4 Performance Evaluation Conclusions

Saber code was improved in third round of NIST contest as shown in graphs for time consuming of key generation, encryption and decryption. SABER can be improved in part of code related to polynomial multiplication. Some recent articles achieved the same conclusions, e.g. (Roy, 2020). It was also shown that the function "MatrixVectorMul" for round three code had a better performance execution time in a ARM architecture and it seems to be less bottleneck than in a x64 architecture as shown in percentage of usage.

5 Round Three Code Improvements

5.1 Usage of Shift Operations

It was found that polynomial multiplication algorithm used in Saber code could be improved in performance by changing division operations of multiples of two by shift operations. It is known that shift operations have better performance than division operations that use dividends multiple of two (https://www.geeksforgeeks.org/bitwise-algorithms/). The improvements were made on *Toom-Cook* and *Karatsuba* algorithms used by SABER.

5.2 Some Code Adjustments

It has also been performed a code adjustment in "MatrixVectorMul" function to decrease algorithm's cost. This function was divided in two according to use of matrix transposed or not. Below it is shown the old and new functions and how it's called in Saber code in key generation and encryption.

```
void MatrixVectorMul(
const uint16_t A[SABER_L][SABER_L][SABER_N],
const uint16_t s[SABER_L][SABER_N],
uint16_t res[SABER_L][SABER_N])
{
    int i, j;

    for (i = 0; i < SABER_L; i++) {
        for (j = 0; j < SABER_L; j++) {
            poly_mul_acc(A[i][j], s[j], res[i]);
        }
    }
}

void MatrixVectorMulTranspose(
const uint16_t A[SABER_L][SABER_L][SABER_N],
const uint16_t s[SABER_L][SABER_N],
uint16_t res[SABER_L][SABER_N])
{
    int i, j;

    for (i = 0; i < SABER_L; i++) {
        for (j = 0; j < SABER_L; j++) {
            poly_mul_acc(A[j][i], s[j], res[i]);
        }
    }
}

void indcpa_kem_enc(
const uint8_t m[SABER_KEYBYTES],
const uint8_t seed_sp[SABER_NOISE_SEEDBYTES],
const uint8_t pk[SABER_INDCPA_PUBLICKEYBYTES],
uint8_t ciphertext[SABER_BYTES_CCA_DEC])
{
   ...
   MatrixVectorMul(A, sp, bp);
   ...
}
void indcpa_kem_keypair(
uint8_t pk[SABER_INDCPA_PUBLICKEYBYTES],
uint8_t sk[SABER_INDCPA_SECRETKEYBYTES])
{
   ...
    MatrixVectorMulTranspose(A, s, b);
   ...
}
```

5.3 Improvements

The "MatrixVectorMul" function was improved and its many execution times were profiled in a file to be evaluated statistically. The same steps for testing Saber third round's code was used to test our improvements on the "MatrixVectorMul" and assert its increase in performance. For each round of the thirty rounds of code performance test, it was calculated "MatrixVectorMul" time consumption and plotted in a table to compare to its unchanged code time consumption (Table 2). For each round, the difference between time consumption of "MatrixVectorMul" new and old code was calculated and found improvement percentage (table 2). Upon improvements calculated for each round, it was calculated the improvement's average and found that Saber improved *3.26* of percentage compared to the original code.

Table 2: MatrixVectorMul Before and After Improvements

Before	After	Round	Improvement
138.3	133.2	R1	3.9
138.2	133.8	R2	3.2
137.8	133.3	R3	3.3
138.2	133.4	R4	3.4
137.4	133.8	R5	2.6
137.6	133.7	R6	2.9
138.4	133.0	R7	3.9
138.5	133.1	R8	3.9
138.4	133.3	R9	3.7
137.6	133.4	R10	3.1
136.8	133.2	R11	2.6
137.6	134.2	R12	2.5
138.0	133.4	R13	3.3
138.2	133.3	R14	3.6
138.3	134.3	R15	2.9
138.1	133.8	R16	3.1
137.5	133.3	R17	3.0
138.0	133.5	R18	3.3
140.7	132.9	R19	5.6
138.1	132.6	R20	4.0
137.7	133.2	R21	3.3
137.8	133.7	R22	3.0
138.1	133.7	R23	3.2
137.8	133.8	R24	2.9
137.9	133.4	R25	3.3
137.8	133.8	R26	2.9
137.6	134.5	R27	2.3
138.2	132.4	R28	4.2
137.5	132.6	R29	3.6
138.0	132.9	R30	3.7

6 CONCLUSIONS

Saber is a post-quantum cryptographic algorithm that seems to be suitable for Android environments according to performance results. It was also found that the Saber reference implementation code could be improved if some arithmetic operations would be changed to shift operations in polynomial multiplications. A small code improvement was also proposed to better algorithm cost in calls to multiplication of polynomials.

REFERENCES

Howe, J., Prest, T., , and Apon, D. (2019). Sok: How (not) to design and implement post-quantum cryptography. Cryptology ePrint Archive, Report 2021/462.

Roy, A. B. S. S. (2020). Optimized polynomial multiplier architectures for post-quantum kem saber. Technical report, eprint.iacr.org.

Saarinen, M.-J. O. (2019). Mobile energy requirements of the upcoming nist post-quantum cryptography standards. Technical report, arxiv.org.

Vercauteren, J.-P. D. A. K. S. S. R. F. (2018). Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. Technical report, eprint.iacr.org.

Xu, R., Cheng, C., Qin, Y., and Jiang, T. (2018). Lighting the way to a smart world: Lattice-based cryptography for internet of things. *CoRR*, abs/1805.04880.