Post-Quantum Cryptography on IoT: Merkle's Tree Authentication

Gokay Saldamli[1], Levent Ertaul[2] and Bharani Kodirangaiah[2]
[1]San Jose State University, San Jose, CA, USA
[2]California State University, East Bay, Hayward, CA, USA
gokay.saldamli@sjsu.edu, levent.ertaul@csueastbay.edu, bkodirangaiah@horizon.csueastbay.edu

Abstract **– Security and privacy issues are considered as the biggest challenge for Internet of Things (IoT) in today's world. Unlike smart phones, tablets or desktop computers, IoT gadgets have either non or very little built-in security functionalities. This is mostly due to cost pressure; device manufacturers prefer not to implement security countermeasures that might increase the cost with more hardware requirements or slow down their development cycles. With growing IoT attacks, this choice would not be an option. Hence, more and more IoT devices are likely to be equipped with stronger cryptographic primitives in the coming years. However, as quantum computing is approaching, today's most popular cryptosystems would not withstand to these new machines. Fortunately, there are still algorithms including Merkle Tree signature scheme remain secure to quantum computing attacks. In this study, we attempt to find out the performance analysis of Merkle Tree signature on a Pyboard IoT device with STM32F405RG microcontroller. Analysis of this algorithm for message signing and verification is reported with respect to parameters like execution time, memory consumption, CPU utilization and power consumption.**

Keywords: IoT security, Lamport one-time signature, Merkle tree signature scheme, post-quantum cryptography, Microcontroller.

1. INTRODUCTION

Internet of Things (IoT) devices are small embedded systems integrated with objects that we use daily [1]. These devices often consist of sensors and other components which can be monitored and controlled over networks and the Internet. IoT devices are often used to create "smart" systems. For example, a healthcare monitoring system equipped with some wireless medical sensors can continuously collect health information in a variety of scenarios. Such a sensor based IoT system may simply provide preventive healthcare or may be evolved so that it can alert the patients and their families in case of any potential health issues [2].

Security surely plays a significant role in such applications; it ensures the privacy and safety of the users and their sensitive data. In general, data security is achieved by encryption using mostly standard cryptographic algorithms. However, in the presence of quantum computers, there are known quantum methods that will be able to cryptanalyze the most commonly used public key-based key agreement and digital signature algorithms [3]. Quantum computers are systems making use of quantum mechanics in order to perform computations on data. Quantum computing is still in its infancy but there is big hype both in government and industry research (e.g. national institutes, Microsoft, IBM, Google Research and D-wave [5] [6]) to build real world quantum computers.

One of the main differences between classical digital computers and quantum computers is that with quantum computers, the accuracy of the result can increase linearly with the input size. In fact, this characteristic makes quantum computers much more powerful in solving very hard problems including the underlying problems (i.e. integer factorization and discrete logarithm) of most popular cryptosystems. Quantum computing represents a new technological leap in solving real world problems that are not practicable with ordinary computers but meanwhile, nobody would like to see the fall of our security infrastructures with the tremendous power of these machines. Therefore, there is a search of cryptographic algorithms, sometimes also called post-quantum cryptography (PQC) that will withstand against the attacks using these new machines [4].

A secure digital signature is the most essential component of an IT-security solution, and several schemes, such as the Digital Signature Algorithm DSA and the Elliptic Curve Digital Signature Algorithm ECDSA are already used in practice [7]. These digital signatures, however, could be broken in a quantum setting by Shor's algorithm [8]. However, there are cryptosystems that rely on "other" problems remain secure to quantum attacks. For instance, Merkle tree signature scheme (MSS) [12] which is a hash-based signature scheme is considered as quantum-resistant (secure even against theoretical quantum level attacks [9] [10]) and studied as a strong candidate for PQC. One good thing about MSS is that if the underlying hash function (e.g. SHA-256 [11]) is broken or found to be no longer collision resistance, one can replace the broken hash by another collision resistant hash function

The aim of this study is to examine the performance of Merkle tree signature scheme on Pyboard IoT device. One of the biggest challenges is to implement MSS on the such a constrained hardware having very limited memory (192KB of RAM and 1MB of Flash) and CPU (168 Megahertz). In the next section, Lamport one-time signature scheme [13] and MSS algorithm and its computational analysis are discussed. Section 3 presents the Pyboard devkit specification and implementation of MSS on Pyboard devkit. Section 4 reviews the performance and power consumption results obtained from MSS implementation. Lastly, in Section 5, we present concluding thoughts on MSS implementation on Pyboard.

2. HASH-BASED SIGNATURE SCHEMES

2.1. Lamport one-time signature scheme (LOTSS)

LOTSS is a signature scheme in which the public key can only be used to sign a single message [14]. The security of the LOTSS is based on cryptographic hash functions. Any secure hash function can be used, which makes this signature scheme very flexible. If a hash function becomes insecure, it can easily be exchanged by another secure hash function. In the following first the key generation, then the signing algorithm and finally the verification algorithm of LOTSS are described.

Key Generation:

- For any given message, perform hash on message to produce fixed length hashed message.
- Generate random number random numbers X_{ij} with $1 \leq i \leq k$ and $j = \{0, 1\}$.
- For each i and j compute Y_{ij} = Hash(X_{ij})
- Y_{ij} are the public key also called verification keys
- X_{ij} values are the private key also called Signing keys

Signing a message:

- Hashed message M is converted to bits ($M = m_1, m_2, \ldots, m_k$ with $m_i \in \{0, 1\}$)
- For each bit in message select the corresponding 0 or 1 from private key (X_{ij} with $1 \leq i \leq k$ and $j = \{0, 1\}$).
- Example: If message bit is 0 them $sig_i = X_{i0}$ otherwise $sig_i = X_{i1}$.
- Signature is the concatenation of all sig_i for $i = \{1, \ldots, k\}$ and k is the length of hashed message in bits
- Final signature is $sig = (sig_1||sig_2||\ldots||sig_k)$

Signature Verification:

- $sig = (sig_1||sig_2||\ldots||sig_k)$ is the signature of message M= $m_1, m_2, \ldots, m_k$ with $m_i \in \{0, 1\}$
- Y_{ij} is the public key of corresponding private key used for signing.
- Compute for each $1 \leq i \leq k$ the hash value $H(sig_i)$
- If $m_i = 0$ then $H(sig_i)$ must be $H(sig_i) = Y_{i0}$ otherwise $H(sig_i)$ must be $H(sig_i) = Y_{i1}$ to be a valid signature.

Although the potential development of quantum computers threatens the security of conventional cryptographic algorithm. LOTSS with large hash functions would still be secure [15]. LOTSS key can be used just to sign single message. However, combined with Merkle hash tree scheme, a single key could be used to sign multiple messages which is discussed in the next section.

2.2. Merkle tree signature scheme (MSS)

MSS is hash-based signature scheme makes use of LOTSS to sign a message using single key-pair. Using a single key pair to sign multiple messages makes attacker easy to forge the signature. Merkle [16] proposed solution to this problem with the binary hash tree. In this study, we implement Merkle tree signature scheme using LOTSS to generate multiple key pairs for signing multiple message with one universally known public key (root of the binary tree). MSS is Binary tree structure shown in Figure 1 where each leaf nodes are hash value of LOTSS public key. Each inner node is hash of the concatenation of left and right nodes. The root node is used to authenticate all the leaf nodes i.e. all the LOTSS public key.

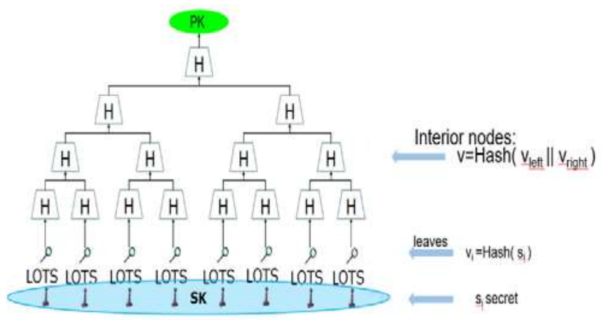


Figure 1: MSS binary tree construction with 8 leaf

Figure 1 explanation is as follows; SK are signing/secret keys which are random numbers. LOTSS key generation is used in MSS to generate secrete keys with 2xN matrix where N is the output of hash in bits (e.g. for SHA-128, matrix is 2x128). V_i in the figure is verification key (or public key) which is hash on 2xN secrete key matrix. Results in one more 2xN verification key matrix. The input to the leaf node hash is concatenation of the elements in 2xN verification key matrix (column-wise concatenation). There are 2^H key pairs, H being the height of the binary tree. Figure 2 explains the algorithm used to generate tree from leaf nodes. Stack is used to store binary tree nodes. key.

```
Algorithm 1 Treehash
Input: Leaf $l$, stack $\mathcal{S}$
Output: updated stack $\mathcal{S}$

1. push $l$ to $\mathcal{S}$
2. while top two nodes of $\mathcal{S}$ have the same height do
   2.1. pop $n_1$ from $\mathcal{S}$; pop $n_2$ from $\mathcal{S}$
   2.2. push $H(n_2||n_1)$ to $\mathcal{S}$
3. return $\mathcal{S}$
```

Figure 2: Algorithm to generate hash tree

Generated MSS key pair can sign/verify 2^H documents, where H is the level of binary tree. Signer generates 2^H one-time key pairs (X_j, Y_j), $0 \leq j < 2^H$. X_j is the signature key and Y_j is the verification key The leaves of the MSS are the digests $Hash(Y_j)$, $0 \leq j < 2^H$. The inner nodes of the Merkle tree is concatenation of left and right children. The MSS public key is the root of the Merkle tree.

The root of the tree represents the public key, the set of all LOTSS signing keys becomes the secret keys. For hash based LOTSS the secret keys are random bit strings. Hence, instead of storing all LOTSS secret keys, one can store a short seed and (re-)generate the LOTSS secret keys using a cryptographically secure pseudorandom generator [17]. To prevent reuse of LOTSS key pairs, they are used according to the order of the leaves, starting with the leftmost leaf. To do this, the scheme keeps as an internal state the index of the last used LOTSS key pair.

Signing message M using LOTSS discussed in Section 2 results in sig'. Signing the message is done by randomly selecting one of the key pairs (X_j , Y_j). Signature of the message is $SIG = (sig', pk_{LOTSS,i}, Auth_i)$, the LOTSS signature sig' on the message using the i^{th} secret key(SK), the i^{th} LOTSS public key is $pk_{LOTSS,i}$ (represented with in Figure 3), authentication path $Auth_i$ (represented with in Figure 3) consists of neighbouring nodes on the path to the root of the MSS tree. Authentication path is used build the leave and reach the root of the tree to verify the public key.

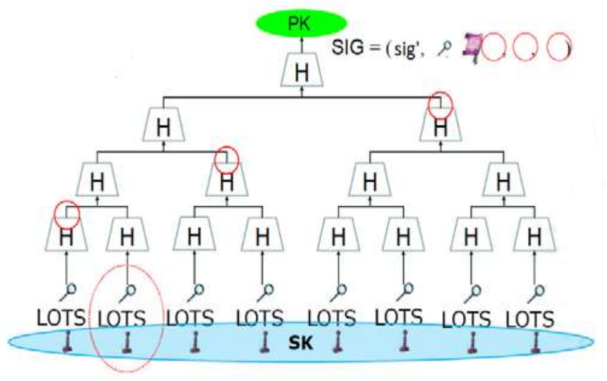


Figure 3: Signature produced by MSS using Lamport one-time signature scheme

To verify the signature SIG on message M, the verifier first validates the LOTSS signature on the message, with $pk_{LOTSS,i}$ using LOTSS signature verification mechanism (discussed in Section 2). If this verification succeeds, the LOTSS public key is verified. Towards this end, the i^{th} leaf is computed as the hash of $pk_{LOTSS,i}$. Then, a root value is computed, using the nodes in $Auth_i$. If this root value matches the one given as public key, the signature is accepted, otherwise it is rejected.

Recall that a binary tree of height h has 2^h leaves. Hence, a MSS with a tree of height h can be used to sign $N = 2^h$ messages. For runtimes, the determining parameter is N. Key generation requires about 2^h hash function calls and is hence linear in N. Signing consists of one LOTSS signature and the authentication path computation. This can be done in time logarithmic in N using e.g. the Binary digital search (BDS) tree traversal algorithm from [18] to compute the authentication path. Verification time is also logarithmic in N.

Now that we know MSS, in Section 3 we calculate complexity for generating nodes, generating tree using authentication path and binary tree traversal using BDS [19],

2.3. MSS algorithm complexity analysis

The big advantage of the MSS is, that many signatures can be generated with using only one public key. However, this advantage comes with an increase of computation time and signature length. In the following we will examine the computation time of each part of the signature process. To generate the public key p_k, 2^n one-time signature keys must be generated. Then every node of the hash tree must be computed. The tree consists of $2^{n+1} - 1$ nodes. One hash operation is needed to calculate a node, so that $2^{n+1} - 1$ hash operations are needed to generate the public key. It is obvious, that the size of such a tree is limited. To compute 240 nodes is very costly, to compute 280 nodes is impossible.

For sizes, the important parameter is the output length of the hash function n. The public key is a n bit hash value. The secret key consists of a n bit seed (assuming pseudorandom key generation; and a public state for the Binary digital search tree (BDS) algorithm in the order of $n \log N$ if BDS is used). The signature size of the classical MSS using LOTSS is $\approx 2n^2 + n\log_2 N$, i.e. quadratic in n, where the $2n^2$ is caused by the LOTSS and the $n\log_2 N$ by the authentication path. Typical values for n and N are $n = 256$ and $N = 220$. For a more detailed overview, also describing tree traversal algorithms, see [20].

Analysis of MSS in this section explains signature increases linearly with time for different n value. MSS is implemented on PyBoard to analyse the performance in resource constraint environment discussed in the next section.

3. PyBOARD DEVKIT IMPLEMENTATION OF MSS

It is decided to test MSS scheme on a PyBoard development kit (devkit) shown in figure 4 with STM32F405RG microcontroller belongs to a family of 32-bit RISC [21] MCUs operating at a frequency of up to 168 MHz with wo general-purpose 32-bit timers. a true random number generator (RNG) [22] [23].

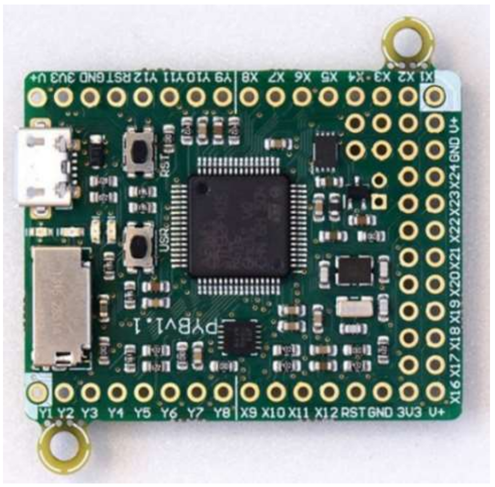


Figure 4: A picture of Pyboard

Since the aim of this study is to implement MSS algorithm in IoT environments, PyBoard with STM32F405RG is considered as the target platform, which is specifically designed for these environments [24] and supports the specifications mentioned in Table 1.

Table 1 – Pyboard devkit Specification

Device Feature	Specification
CPU	168 MHz Cortex M4
Dynamic RAM	192KiB
Flash memory	1024KiB
Serial I/O	Micro USB
Connection	Serial Communication

The devkit features many I/O pinouts for component interfacing, however for this study these pins are not used. The PyBoard is built with a small microSD card slot and its firmware will automatically load the card as the root file system for the PyBoard [26]. Flash memory on PyBoard is 1MB, which can store limited data. The python program developed for this study uses micro USB connection for printing console statements with reporting results and debug messages.

The PyBoard devkit was connected to ASUS Zenbook UX305 laptop running Microsoft Windows 10 Professional via Micro USB cable. To monitor and program the activity, micro python console pre-installed on the PyBoard was used. PyBoard supports its own version of python, which is called micro python [27]. PyBoard is connected to the laptop using serial communication mode. Successful connection takes to micro python v1.3.2 prompt. The program uploaded to the PyBoard devkit's flash memory runs initialization code from boot.py to import basic libraries. Python scripts flashed on the PyBoard are Merkle_key_generation.py(17KB), Lamport.py(26KB), Signarture_verification.py(16KB). Total 59 KB out of 1MB flash memory is used for Merkle tree signature scheme micro python scripts i.e. 965KB is available space.

The MSS algorithm on the PyBoard was based on reference implementations from MediumCorp [28]. PyBoard supports micro python, hence python code was modified from python3 to Micro Python. Performance of the algorithm is evaluated and modified based on libraries supported by PyBoard. Initially Key generation, generating tree were tested, alongside with signing message and verification of signature.

MSS authentication is hash-based algorithm, hash used in this study is SHA [29], PyBoard supports just SHA-1 [30] without SD card. SHA-256 and SHA-512 [31] [32] can also be used if the corresponding micro python libraries copied to SD card. For this study due to memory constraints SHA-128 has been used for MSS, compromising on security to run the MSS algorithm on PyBoard. Evaluate the performance i.e CPU and memory utilization of PyBoard running MSS which is discussed in the next section.

Considering memory constraints on Pyboard suitable hash function is used to implement. Analysis of execution time for generating different key pairs, signing and verification of MSS is also evaluated in the next section.

4. PERFORMANCE ANALYSIS

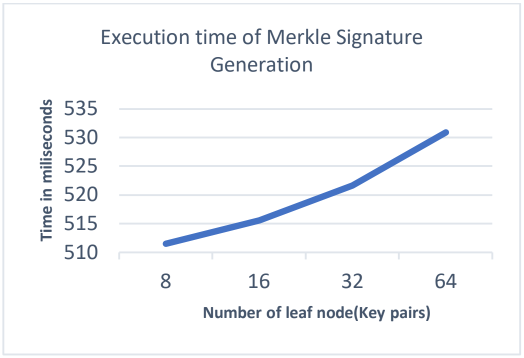
Execution time for MSS key generation, signature and verification is tested on PyBoard and results are evaluated. Performance of the implementation was calculated using the micro python library ***utime*** [33]. Private keys are generated using random numbers. Public keys are generated using hash (SHA-128) of private keys. Table 2 results are evaluated using SHA128 and size of random number used for each private key is 6 KB.

Table 2: Key generation time

	Key generation time (in seconds)	Memory required (Private + Public key)
8 - key pairs	18.31	48KB + 32KB = 80KB
16 - key pairs	35.56	95KB + 65KB = 160KB
32 - key pairs	70.69	189KB + 131KB = 320KB
64 - key pairs	140.83	378KB + 262 KB = 640KB

From Table 2 we can see that increasing the number of key pairs, increases key generation time which almost doubles in each step. Memory required to store these generated key pairs is also increasing rapidly. We stopped at 64-key pairs because available memory on board was 965KB as discussed in Section 3. Memory required for 128 key pairs is 1289KB which results in shortage of memory.

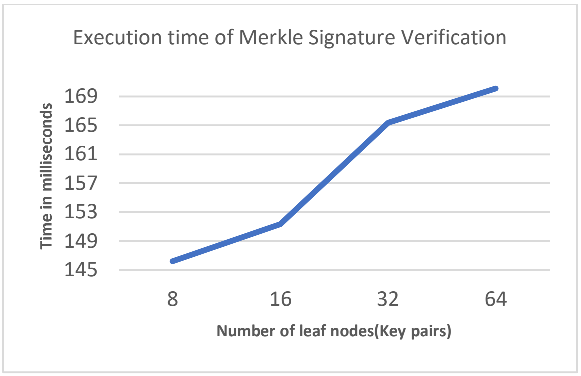
MSS signature generation includes generating signature, writing it to json file [34]. Json file format is used because signature includes 3 part: sig', public key and path nodes, reading these part during verification is easy with json format. We use corresponding micro python library ujson [35] to read and write json file. In our study, key pair selected from different binary tree level 3, 4, 5 and 6 to sign the message is timed as shown in Figure 5. As we can see from the Figure 5 for increase in number of key-pairs signing time increases in milliseconds. PyBoard devkit is capable of signing the message given the key pairs.



	Level	Signature generation time (ms)
8- Key pairs	3	511.53
16- Key pairs	4	515.56
32- Key pairs	5	521.67
64- Key pairs	6	530.94

Figure 5: MSS Signature generation execution time

To verify the signature which contains multiple sections i.e *sign', pk, Auth* in json format is read and each section is extracted separately. Verification time includes above mentioned task along with the major part of verifying the signature and public key using path nodes. These tasks are timed and analysed using the graph mentioned in Figure 6. Verification of the message with different key pairs as shown differs in milliseconds which are suitable for PyBoard devkit.



	Level	Signature verification time (ms)
8- Key pairs	3	146.21
16- Key pairs	4	151.31
32- Key pairs	5	165.36
64- Key pairs	6	170.11

Figure 6: MSS Signature verification execution time

As shown in Table 2, MSS key generation on Pyboard takes more time, hence key generation is not recommended on Pyboard. Analysing the graphs from Figures 5 and 6, we show that for different key-pairs of various levels, signing and verification can be done on the Pyboard devkit. We can see that it takes around few milliseconds for both.

Since Pyboard supports MSS signing and verification. We discuss in detail about CPU and memory utilization of these two operations on Pyboard in the next section.

4.1. CPU and memory utilization

The required space for our authentication algorithm is 59KB out of 1MB user programmable space with available 965KB flash memory. Memory required for different hashing assuming the number of key-pairs as 8 is shown in Table 3. As we increase the number of key-pairs, signature size increases. Based on the IoT device flash memory appropriate hashing is selected. Since we implemented SHA-128 on Pyboard, it requires 24KB space which provides enough space for an IoT application, limited RAM memory will limit the applications to develop on Pyboard.

Table 3: Signature size for different SHA using key-pair = 8

Hashing (SHA)	Signature Size
128	24KB
256	98KB
512	400KB

We calculate the RAM utilization in bytes and CPU Utilization in % for MSS signing and verification. Total available RAM 192KB and single core CPU with maximum CPU utilization of 100%. Calculating the RAM usage is done using micro python library [36]. Code sample shown below for MSS signature generation, likewise we can call verification function to obtain memory usage for verification. Signing and verification with keys from different key pairs are calculated and plotted a graph.

```
def test():
    gc.collect()
    before = gc.mem_free()
    Merkle = Merkle_Signature_generatation_key(8)
    gc.collect()
    after = gc.mem_free()
    print("Merkle object takes up", before - after, "bytes")
```

In Figure 7, graph is plotted for RAM memory usage using key pairs 8, 16, 32 and 64. Noted memory usage few milliseconds before starting MSS signature. We can observe sudden raise from 15- 45ms in the curve which means that, at this point MSS signature has started. Signature includes *sig'*, *pk* and *Auth* which requires more memory.

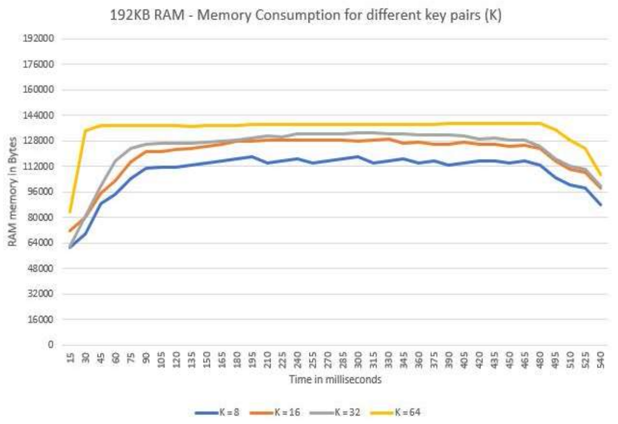


Figure 7: RAM memory consumption for Key pairs 8, 16, 32 and 64.

Message in bytes are hashed (SHA-128) and converted to bits. This operation is performed during initial stage of signing. Message in bits is signed using LOTSS as discussed in Section 2, signature requires more memory. Hence in the Figure 7 curve reaches maximum available RAM memory at 90 ms for 8,16 and 32 key pairs, but for 64-key pairs RAM memory usage reaches peak early at 30ms. Curve is almost flat once the peak RAM memory is reached indicating signing uses all the available RAM memory. To free the RAM memory CPU writes and reads from flash memory. For increase in key pairs available RAM memory in bytes increases slightly. Curve drops at 465ms indicating completion of MSS signature and verification as there is drop in RAM memory usage.

In Figure 8, graph is plotted for CPU utilization using key pairs 8, 16, 32 and 64. CPU utilization is monitored few milliseconds before the MSS signing. Observe the CPU utilization slowly reaches a peak value (>95%) at 45ms which means initial stage it is selecting a single key pair to sign the document and hashing the message to sign (converting into bits). It shows sharp raise initially which means CPU started signing processes. When the signing starts CPU utilization is at peak 100 % at 90ms for 8,16,32, but for 64 key pair it reaches peak early at 60ms. MSS signature process consumes all the available RAM memory space and needs more than available so it writes and reads from flash memory, hence CPU utilization for this rapid read and write is 100%. Once the authentication (signing and verification) is complete at 465ms CPU utilization starts decreasing. We can observe same pattern for different key pairs.

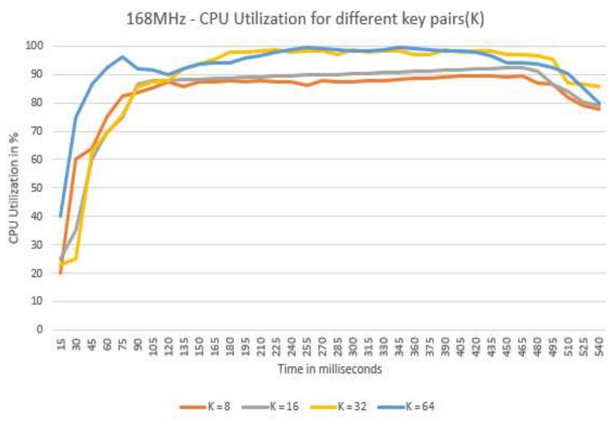


Figure 8: CPU usage for MSS authentication for key-pair 8, 16, 32 and 64

From Figures 7 & 8, we can see that MSS running on PyBoard with STM32F405RG microcontroller with 192KB RAM consumes more memory and CPU execution time. At any given point MSS consumes all the memory and CPU. We cannot perform other operations when the PyBoard is performing signing and verification. Power consumed by PyBoard for MSS signing and verification is calculated based on PyBoard specification.

4.2. Power consumption

The power consumption of the authentication algorithm running on PyBoard can be estimated by Equation 1 [37].

$$E= V_{cc} * I * N * \tau \quad (1)$$

For Pyboard devkit version 1.1, the specification [38] produce the formula with values mentioned in Equations 2 and 3. Using internal flash memory 60mA standby current with 3.3V. pyboard serial monitor provides the value of cycles when testing performance of signing and verification.

$$V_{cc} = 3.3V \quad (2)$$

$$I = 0.06A \quad (3)$$

$$Time = N * \tau = cycles * 80\ e^{-6}\ s$$

Considered N * τ as average time for signing and verification from Table 2. Table 4 lists the calculated power consumption for authentication using 128-bit block size. At 168MHz from flash without I/O, Energy is calculated by substituting values.

Table 4: Calculated power consumption

Signing	Verification
9.763 J	7.974 J

5. CONCLUSION

In this study, we have evaluated the performance of MSS on PyBoard with STM32F405RG microcontroller with respect to energy consumption, execution time, CPU usage and memory consumption which place significant role in implementing MSS for resource constrained environments. It is proven that key generation requires more memory than the available memory and takes more execution time on PyBoard, hence generation of key pairs is not recommended on PyBoard.

MSS signature and verification can be handled on PyBoard devkit based on the analysis of performance and power consumption. MSS signature and verification algorithm require more memory as the key-pairs and number of levels increases. MSS authentication mechanism consumes all the memory and CPU. We cannot perform other operations when the PyBoard is executing MSS. Hashing used in this study is SHA-128 supported by PyBoard without using SD card. Using SHA-128 we are compromising on security to make it implement on PyBoard devkit. The memory constraints of the PyBoard devkit limit the amount of data that can be buffered for processing. Considering of post-quantum authentication, MSS can be implemented on PyBoard applications with moderate processing time (5ms for signing and 1ms for verification) with peak memory and CPU utilization

REFERENCES

[1] H. Petersen, E. Baccelli, and M. Wählisch. *"Interoperable Services on Constrained Devices in the Internet of Things"*. In W3C, editor, W3C Workshop on the Web of Things, Berlin, Germany, June 2014.

[2] Yang, Z. Zhou, Q. Lei, L. Zheng, K. Xiang, *"An IoT-cloud BasedWearable ECG Monitoring System for Smart Healthcare"*. J. Med. Syst. 2016, 40, 286.

[3] Lin, J. Ding, X. Xie, Xiaodong . *"A Simple Provably Secure Key Exchange Scheme Based on the Learning with Errors Problem"* in University of Cincinnati Chinese Academy of Sciences Rutgers University. 01 Jan 2012

[4] Alkim, E. Ducas, L. Pöppelmann, T. Schwabe, Peter. *"Post-quantum key exchange - a new hope"* (PDF). Cryptology ePrint Archive, Report 2015/1092. Retrieved 1 September 2017.

[5] *"IBM Raises the Bar with a 50-Qubit Quantum Computer"* IBM releases. Web. https://www-03.ibm.com/press/us/en/pressrelease/53374.wss#release, November 2017.

[6] Imanuel, *"Computing: The quantum company"*. 2018; Available from: https://www.predictiveanalyticstoday.com/what-is-quantum-computing/#quantumcomputingmodels

[7] Daniel R. L. Brown *"Designs, Codes and Cryptography: Generic Groups, Collision Resistance, and ECDSA"* by Certicom Research, Canada, 119–152, Accepted: 24 June 2003

[8] R. Wolf, CWI and University of Amsterdam *"Quantum Computation and Shor's Factoring Algorithm"* , January 12, 1999, 9 page postscript document

[10] R. C. Merkle. *"A certified digital signature. In Proceedings on Advances in cryptology",* CRYPTO '89, pages 218 - 238, Springer-Verlag New York, Inc, NY, USA, 1989.

[11] *"Secure Hash 256-bit"*. NIST. Retrieved 2010-11-25.

[12] Merkle, Ralph C.: *"Secrecy, Authentication, And Public Key Systems"*. Ph.D. thesis, Stanford University, 1979

[13] L. Lamport. *"Password authentication with insecure Communication"*. in Communication of the ACM 24(11):770 - 772, Nov. 1981.

[14] L. Lamport. *"Constructing digital signatures from a one-way function"*. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, 1979.

[15] P. Ray; Cooper (2009). *"Quantum Resistant Hash-Based Cryptography: A Survey"*. NIST. Retrieved 23 Apr 2015

[16] R. Merkle. *"A certified digital signature"*. In Gilles Brassard, editor, Crypto'89, volume 435 of Lecture Notes in Computer Science, pages 218–238. Springer Berlin / Heidelberg, 1990. 1, 2

[17] B. Elaine, B. William, P. William, S. Miles (July 2012). *"Recommendation for Key Management"* (PDF). NIST Special Publication 800-57. NIST. Retrieved 19 August 2013.

[18] J. Buchmann, E. Dahmen, and M. Schneider. *"Merkle tree traversal revisited"*. In Johannes Buchmann and Jintai Ding, editors, Post-Quantum Cryptography, volume 5299 of Lecture Notes in Computer Science, pages 63–78. Springer Berlin / Heidelberg, 2008. 1, 2, 6, 7

[19] Jarc, Duane J. *"Binary Tree Traversals"*. Interactive Data Structure Visualizations. University of Maryland. 3 December 2005

[20] J. Buchmann, E. Dahmen, and M. Szydlo. *"Merkle signature computation analysis"*. In Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors, Post-Quantum Cryptography, pages 90–93. Springer Berlin Heidelberg, 2009. 1, 2

[21] W. Andrew, L. Yunsup, P. David A. Krste. *"The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA version 2 (Technical Report EECS-2014-54)"*. University of California, Berkeley. Retrieved 26 December 2014.

[22] "STM32F405RG." STM32F405RG - STM32 *Dynamic Efficiency MCU, ARM Cortex-M4 core with DSP and FPU, up to 512 Kbytes Flash, 84 MHz CPU, Art Accelerator - STMicroelectronics*.N.p.,n.d.Web.

[23] *"ARM Information Center."* ARM Information Center. N.p.,n.d.Web. http://infocenter.arm.com/help/index.jsp?topic=%2Fcom.arm.doc.ddi0337h%2FBIIFBHIF.html.

[24] *"STM32 – 32-bit ARM CORTEX MCU"* STMicroelectronics. N.p.,n.d. Web. http://www.st.com/en/microcontrollers/stm32-32-bit-armcortex-mcus.html?querycriteria=productId=SC1169"

[26] "Mount SD card" [Online]. Available: https://docs.micropython.org/en/latest/pyboard/pyboard/general.html

[27] "Micro python," [Online]. Available: http://docs.micropython.org/en/latest/pyboard/

[28] "Repository" [Online]. Available: https://medium.com/@evankozliner/merkle-tree-introduction-4c44250e2da7

[29] "Hashing algorithm" [Online], Available: http://docs.micropython.org/en/v1.9.3/wipy/library/uhashlib.html

[30] D.Cannière, Christophe; R. Christian. *"Finding SHA-1 Characteristics: General Results and Applications"*. 15 November 2006

[31] D. Khovratovich, C. Rechberger & A. Savelieva (2011). *"Bicliques for Preimages: Skein-512 and the SHA-2 family"* (PDF). IACR Cryptology ePrint Archive. 2011:286.

[32] FIPS PUB 180-1, *"Secure Hash Standard, SHA-512"*. Available at www.itl.nist.gov/fipspubs/fip180-1.htm.

[33] *"utime for timing code"* [Online]. Available: https://docs.micropython.org/en/latest/pyboard/library/utime.html

[34] *"draft-wright-json-schema-01 - JSON Schema: A Media Type for Describing JSON Documents"*. json-schema.org/. Retrieved 23 July 2017.

[35] *"json file"* [Online]. Available: https://docs.micropython.org/en/latest/pyboard/library/ujson.html

[36] *"RAM memory usage"* [Online]. Available: https://forum.micropython.org/viewtopic.php?t=1747

[37] H. A. Kader ,D. Salaman, and M. Hadhoud, *"Studying the Effects of Post-Quantum Algorithms,"* International Arab Journal of e-Technology, vol. 2, no. 1, 2011.

[38] *"Pyboard voltage"* [Online] Available: https://forum.micropython.org/viewtopic.php?t=229