[[1]](#footnote-1)

Non-Volatile Memory Integrity using Merkle Tree.

Maher Sanalla, Adham Khalil, Meni Orenbach

*Abstract*-- With the advance of technology, the amount of personal data that is stored on computer-devices increased substantially, such data can be easily manipulated or lost. Non-Volatile memory is very commonly used by those computer devices. Non-Volatile memory system is vulnerable to many attacks that vary from faulty operating system to data manipulation and disruption, Thus the demand for ways to ensure the integrity of NV-memory has increased correspondingly. Many solutions were proposed to provide data integrity of the memory values, each with its pros and cons. In this paper we will analyze the Merkle-Tree Integrity system, which provides security, confidentiality, and authentication of the memory values, and prevents malicious acts of manipulations or damage to the memory data. The use of Merkle-Tree authentication has been widely adopted to various applications, such as blockchain technologies, e.g., Bitcoin industry. We will analyze its benefits, cost, and performance thoroughly.

*Index Terms*-- Merkle Tree, Integrity Tree, Non-Volatile RAM, Memory authentication, Memory security, memory integrity, Attack-prevention, Oblivious-NVRAM.

# Introduction

A

As we know, secure processors do not trust the main memory of the system because it’s prone to tampering, e.g., an attacker with physical access to the system can read/modify memory content without the right permissions, therefore the processor is required to secure the content of the memory, various technologies has been used to accomplish this goals, for example Intel’s Software Guard Extensions (SGX) was built on CPU’s to enclave certain regions of the memory and secure it in a way that no other process is allowed to access it.

Since a computer system’s memory (DRAM) is prone to attacks, all security principals should be addressed, that is, the integrity, confidentiality and availability of the DRAM content, in our project we’ll be focusing on integrity and confidentiality, various approaches has been done regarding data leakage and integrity problem, we will be using merkle tree, the data blocks will be encrypted instead of hashed so that we can perform read/write operations without the need of data in our memory (for confidentiality purposes), instead we’ll be saving the ciphertexts and tags (HMAC), we also assume that we have a trusted memory region, inside it we will save our encryption keys, nonces and merkle root.

# Functionality

The NV-RAM which uses Merkle-Tree Integrity system differs substantially from the typical RAM that we know. The use of Merkle-Tree changes the functionality of RAM Read/Write operations, and provides a new function called “Verify Integrity”, which verifies that the memory content is valid and not corrupted at any given time.

Since the NV-RAM cannot be trusted, Merkle-Tree needs a storage that can be trusted, Otherwise - if there is no such trusted memory area - then the whole idea of integrity could be compromised by an attacker.

Thus, hardware support is needed. Our system requires a small memory area on the microprocessor, which cannot be accessed or corrupted by an attacker, and its content integrity is guaranteed presumably. This area will store the secrets of the NV-RAM content on it, such as encryption-keys, Merkle tree root, and other data that we will discuss thoroughly later.

The operation of this system begins with partitioning the whole memory segment into size-identical blocks, as shown in Figure 1.a below.

 Fig. 1(a). The NVM partition to three sections, whereas for every data block the ciphertext is stored in the first section, the HMAC in the second section and the corresponding nonce in the third section.

After that, we proceed to encrypt every single block with a strong unique key and a random nonce using AES-GCM encryption method, each encryption process yields a ciphertext (which overwrites the original block plaintext) and a tag (HMAC). The encryption key and the corresponding nonce are stored in the trusted area and are not accessible by anyone other than the superuser.

At this point, the whole memory data is encrypted and secured, the ciphertext overwrites the plaintext, thus even if memory leaks happen now, they are useless to the attacker.

In the upcoming sections, we will discuss the functions of the Merkle-tree NV-RAM system and how they operate. Such as Read/Write/Verify/getRoot.

## getRoot

This method builds the Merkle-tree and returns its root. The trees’ leaves are the blocks-HMAC values that we calculated earlier. Now, for every two consecutive HMACs “A” and “B” we calculate the SHA-256 of their concatenation, the result will be the data of their parent in the tree. This process is repeated recursively on each level of the tree, until we reach the first level of the proposed tree where there are only two nodes. Then, we calculate the SHA of their concatenation and that will the root of the tree, as shown in Fig 1(b).

The root of the tree is the ambassador of the memory content. Later, we will see how it is used to decide if memory data is corrupted or valid. We store the root of the tree in the trusted area, so that no one can modify it.

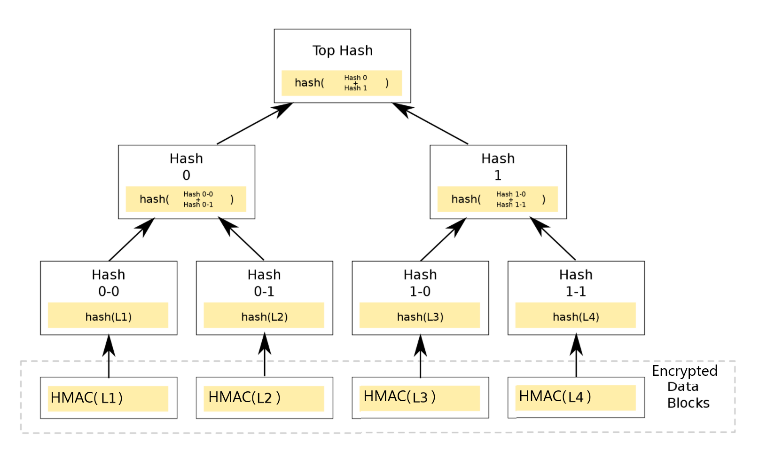


Fig. 1(b). The design of Merkle-tree which is built bottom-up. Its leaves are the encrypted NVM blocks (HMACs), and in the upper levels each node is the hash of its sons’ data concatenation.

## Verify Integrity

After the initiation process is over, we can check if the memory integrity is maintained at any given time in the future. The verify integrity method is the function that checks the integrity of the memory at any given time. The way it functions is as follows: if at a given time t>0 in the future, this function is called, it will calculate the “status-quo” Merkle tree whose leaves are the current-memory blocks. Using the same tree-building process as we discussed earlier, the function will obtain the current-status Merkle-tree root using “getRoot” function. This root is an indicator of the current-status of the memory, if its equal to the root that we trust, which is stored in the trusted-area, then the current-status of the memory is as it was before, thus we deduce that the memory integrity is kept, and it was not corrupted or manipulated. Otherwise, if the current-status root differs from the trusted-area root, then the current-memory is assumed to be corrupt and not valid.

To prove the correctness of this algorithm, let us assume that after the initiation of the Merkle-tree system is done, an attacker came and changed block A data for malicious purposes to A’. Then when verify integrity method is called, it will build the Merkle-tree Bottom-up. It will begin with taking the current memory-blocks as leaves and calculate the higher levels recursively. Let B be the neighboring block to A’, When the parent of A’ and B is calculated, it will be the outcome of SHA-256(A’ || B). But since A’ differs from A, then A’ || B differs from A || B, due to the collision-resistance characteristic of SHA-256 function, the outcome of the SHA(A||B) is different than SHA(A’||B). Meaning the current parent of A’ and B is different from what it was before when the trusted tree-building process was performed.

Using Induction, we can prove that the grandfather of A and B will differ from its previously calculated value, and so on, till reaching the root. So, we deduce, that changing the content of any memory block will cause a direct change in the root of Merkle-tree. Thus, calculating the current-memory Merkle tree root and comparing it to the previously calculated and trusted root, we can infer directly if the memory blocks were changed from their previous values, or not.

## Read Block

When the user wants to read a block, he provides the corresponding block index. Before supplying the user with the data, he wants, we must verify the integrity of the tree to be sure that the data that we supply him is not corrupted. Thus, we call the verify integrity and continue to read the block only in case the tree’s’ integrity is kept.

When the integrity is assured, we extract the corresponding key and nonce for the requested block from the trusted area, then we decrypt the block and return the plaintext to the user. In case the integrity of the block is manipulated, then we inform the user that the block data was corrupted and do not supply him with the data.

The correctness of this method is deduced straightforwardly from the verify integrity correctness.

## Write Block

Let us assume that the user wants to update block “A”, he provides the corresponding index of the block with the new plaintext/data. We generate a new encryption-key with nonce for the new plaintext, encrypt it using AES-GCM which generates new ciphertext and new HMAC. We overwrite the previously stored cipher and HMAC with the new ones and update the blocks’ nonce value in the trusted area.

After that we calculate the new Merkle tree root using the same methodology as before – traversing the tree bottom up. At last, we update the stored root in the trusted area with the new root, then inform the user that the write operation was successfully completed.

# additional features

The definition of non-volatile memory is that it can retain stored information even after power is removed.

So, our integrity system must not be corrupted or manipulated in case the power went off. For example, if there was an ongoing write operation to the memory, and shutdown happened, the write operation must be completed when the power comes back automatically. Otherwise, the user will suffer from inconveniences and will have to recall the write operation again. Thus, we implemented the Log feature.

## Log

The Log is a record of operations that are performed on our NVM. It stores the most recent non-complete write operation on our system such. For example, when write is requested from the user, we store the request info in the Log temporarily. Meaning the index of the block, the data to be written, the time of the request, and so on. When the write operation is fulfilled successfully, we empty the log, since the log is an indication of only uncomplete write-requests.

The benefit of the log comes when there is a power-crash in the middle of write-operation, then when power comes back up the previous write operation is lost. But at that time, the Log which is stored in NVM will still hold the previously uncompleted write-operation, and our system will deal with each request in the Log before continuing its work after the restart. The integrity status of the NVM will not be affected by the uncompleted write, since after the system-restart the write will be performed again, which means the Merkle tree root will be updated accordingly, with no loose ends. Thus, our system is still considered non-volatile.

# Optimizations

Performance is a key aspect when talking about memory. The user most important need from the memory system is availability, which means the data must be available at any time and accessible in an efficient manner. Integrity is another aspect the user wishes for in a memory system, but it should not be provided on account of efficient access. We cannot deny that the integrity system we explained above has a large overhead on performance, since the verify integrity is a heavy cost to pay for every write/read operation. Thus, we must look for optimizations, we will introduce few features that will improve the functionality of our memory system.

## A. LRU Cache

Cache is a reserved memory space to store temporary data to help improve our functions performance. In our case, we used an on-chip cache which we consider trusted since its stored in a special hardware next to the microprocessor that is not accessible by a regular user or an outsider.

We will use a Least-recently-used caching system to store nodes from the Merkle-tree to improve the performance of Verify integrity operation. Every block that is stored in cache is trusted and valid. The use of cache will change the functionality of read/write/verify integrity operations as we will elaborate below.

A.1 getRoot:

it operates the same way it did before the addition of cache, but with a slight modification. Now for every node we access/calculate when building the Merkle tree, we store it in the cache as well. This function is only used when initiating the memory system.

A.2 Verify Integrity:

The previous verify integrity function used to calculate the whole Merkle tree to get to the root (it used getRoot function in the past) and then compare it to the trusted root. This operation is heavy, especially in large-memory systems (e.g., having leaves). Thus, with the help of cache, we can change the way verify works. Now instead of calling getRoot and building the whole tree, we will only work on the path from the block -in-question to the root of the Merkle tree.

For example, let “A” be the HMAC of the block who we are verifying its integrity. The new verify algorithm is as follows:

1. Check if “A” is in the cache.
2. If yes: return “A” = = “cached-A”
3. Else
   1. find “A”’s brother – lets name it “B”.
   2. Calculate the hash of A||B which is basically C - parent of A and B.
   3. If “C” is the root of the tree, then return “trusted\_area\_root” = = “C”
   4. Repeat step 1 with “C” as the HMAC in question.

The correctness of this algorithm is based on the assumption that the cache contains only trusted data. Meaning, let “A” and   
“B” be two neighboring nodes in the tree, and let “C” be their parent. If C = = hash(A||B) and C is also found in cache, meaning C is trusted, then its safe to deduce that A and B are also trusted and valid. This is due to the fact hash is a one-way function, meaning its hard to manipulate A/B and still get the same hash result C.

This verify method is much more efficient than the previous implementation, since here we traverse the height of the tree only, and for each level we perform O(1) operations of comparisons. This the complexity is where n is the number of leaves/blocks in the tree. The previously used method used to calculate every single node of a tree with n leaves, which means the previous complexity is O( ). It is visible that this method is much more efficient.

A.3 Read:

Now read will use the new verify integrity implementation, which is based on cache-existence, instead of the previous costly verify-integrity method. And in addition to reading the block, this function will insert the read HMAC into the cache since it was accessed (Updating the cache happens only if the read was performed successfully iff the verify integrity succeeded)

A.4 Write:

After adding the cache, write method will work as follows:

At first, the function will check the integrity of the tree by calling the new verify integrity implementation, if the tree integrity is compromised, we abort and do not allow the write operation to be completed. Otherwise, we will perform the write operation as we did before, but instead of calling getRoot to calculate the new root of the tree, we update the root of the tree in a quicker manner. We iterate over the path from the blocks corresponding HMAC to the root, and eventually when we get to the root level, we update it with the new HMAC, and update the trusted area root as well. Important thing to note is that while we are iterating over the path, we update our cache along the way to the root. For each node there are two options, or it is already in cache and then we need to update its cache-entry with the new corresponding hash, or it is not in cache and then we insert it as a new entry to the cache.

# Simulation

To test our implementation, we wrote benchmark tests that simulate random access read/write operations on NVRAM. Our implementation was written in C++, while the tests were written in Python and the testing was done using Linux terminal.

Different use cases were tested, which vary in memory size that ranges from 64MB to 128MB. Also, different block sizes were used, 64B and 4KB. Cache size was set to be 6400B. We recorded the latency of read/write operation under different environments, such as with/without integrity check, with/without cache system, etc.

The results were collected and analyzed, then processed using MATLAB R2016a. Figures 3.a-3.b show the data we collected.

Chart, bar chart

Description automatically generatedFig 3(a). Simulation of Read Block operation on NVM that contains 16,384 data blocks of 4KB/64B.

Chart, bar chart

Description automatically generated

Fig 3(b). Simulation of Write Block operation on NVM that contains 16,384 data blocks of 4KB/64B data.

# Evaluation

As we can see from the Fig 3(a). the Integrity check is heavy and affects the latency of read operation. Without integrity the performance is better, as expected.

# References

*Basic format for books:*

1. J. K. Author, “Title of chapter in the book,” in *Title of His Published Book,* xth ed. City of Publisher, Country if not USA: Abbrev. of Publisher, year, ch. x, sec. x, pp. xxx–xxx.

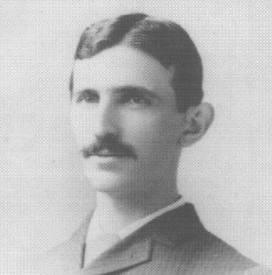
*Examples:*

1. G. O. Young, “Synthetic structure of industrial plastics,” in *Plastics,* 2nd ed., vol. 3, J. Peters, Ed. New York: McGraw-Hill, 1964, pp. 15–64.
2. W.-K. Chen, *Linear Networks and Systems.* Belmont, CA: Wadsworth, 1993, pp. 123–135.

*Basic format for periodicals:*

1. J. K. Author, “Name of paper,” *Abbrev. Title of Periodical,* vol. x, no. x, pp. xxx–xxx, Abbrev. Month, year.

Examples:

**Nikola Tesla** (M’1888, F’17) was born in Smiljan in the Austro-Hungarian Empire, on July 9, 1856. He graduated from the Austrian Polytechnic School, Graz, and studied at the University of Prague.

His employment experience included the American Telephone Company, Budapest, the Edison Machine Works, Westinghouse Electric Company, and Nikola Tesla Laboratories. His special fields of interest included high frequency.

Dr. Tesla received honorary degrees from institutions of higher learning including Columbia University, Yale University, University of Belgrade, and the University of Zagreb. He received the Elliott Cresson Medal of the Franklin Institute and the Edison Medal of the IEEE. In 1956, the term “tesla” (T) was adopted as the unit of magnetic flux density in the MKSA system. In 1975, the Power Engineering Society established the Nikola Tesla Award in his honor. He died on January 7, 1943.

1. [↑](#footnote-ref-1)