Parallelized C++ Implementation of a Merkle Tree

Andrew Flangas, Autumn Cuellar, Michael Reyes, Frederick C. Harris, Jr.
Department of Computer Science and Engineering
University of Nevada, Reno
{andrewflangas, acuellar24, michaelreyes}@nevada.unr.edu, fred.harris@cse.unr.edu

***Abstract*—Merkle trees are primarily known for being an attribute found in blockchain technology. They are used for encrypting data by hashing values multiple times to avoid incidents such as hash collisions, or the successful guessing of hash values. Merkle trees are not only a useful feature found on the blockchain but in the field of Cyber Security in general. This paper outlines the process of implementing a Merkle tree as a data structure in C++ and then parallelizing it using OpenMP. The final result is a Merkle tree password storing program with reduced running-time and the ability to operate on multiple processors. The validity of this program is tested by creating a Merkle tree of the correct passwords, storing the value of the root node, and then building a second tree where a single incorrect password is stored within that tree. The two trees are passed through an audit function that compares the root nodes of the two trees. If they are different, then the tree in question has been tampered with.**

Keywords: Merkle tree, parallel programming, serial programming, hash, OpenMP, processors, running-time, root node, cyber security, MD5, UPC, MPI, password, data structure, login data, threads, blockchain, performance

I. Introduction

In order to ensure security in a rapidly changing field of technology, one must use techniques that are on the frontier of said field. A Merkle tree is an attribute primarily associated with blockchain, but it can also be used in the field of Cyber Security, in general, in ways such as storing sensitive data. By implementing a Merkle tree in C++, we are hoping to disassociate it from blockchain and use it primarily as a tool for Cyber Security purposes. This tool will then be parallelized for optimization aspects such as reduced running time, as well as the ability to run on multiple processors. The Merkle tree will also be implemented as a data structure for a program that will be able to test the efficacy and correctness of this tool. The Merkle tree data structure will be implemented twice: a serial version of the program and then a parallelized version in order to analyze the differences in the running-time and the number of processors being used.

The method that is used to parallelize the Merkle tree data structure is the application programming interface OpenMP [1], due to its simple and flexible interface for creating parallel programs in multiple languages including C++. The type of application that is used to test the Merkle tree data structure is that of a system for storing login data, in which the Merkle tree is used to hash and store the user passwords. In order to properly test the validity of this program, the known passwords for the login system will be fed into the application which will be the nodes for the construction of the Merkle tree. A second tree will also be constructed in which a single password from the original tree is altered. The root nodes of the two trees will then be passed through an audit function, which will compare the two roots to see if they are the same. If one of the roots is different, then it proves the tree in question has been tampered with. The running-times and number of processor threads being used for both the serial and OpenMP versions of the program will be output to the terminal as well.

The rest of this paper is structured as follows: Merkle Trees and Parallelism is described in Section II. Login Application Implementation is presented in Section III, Performance Results are discussed in Section IV, and Conclusions and Future Work are covered in Section V.

II. Merkle Trees and Parallelism

A. Merkle Trees

A Merkle tree is a data structure that combines binary trees and hash tables into one. Each node of the tree is a hash function of the combined children hash functions. The leaves are just a hash of the input value received while the root node is a hash of the whole tree, the structure can be seen in (Fig. 1). This tree can be very helpful in cyber-security applications. The root node's value is known based on the given input values. Any changes to the inputs, resulting from an attack, will cause the root node to change suddenly. This change will be detected which can allow users to determine the system as corrupt. Merkle trees are primarily used on blockchain applications, where the data is hashed using a Merkle tree and then the root node hash of the tree is added to a block. This allows for increased security measures when encrypting data that are not available when using traditional hashing methods.

Imagine a scenario of an online sealed-bid auction, where bidders simultaneously submit sealed bids to the auctioneer, so that no bidder knows the amount the other auction participants are bidding. Assuming the bid amounts of the participants are stored using traditional hashing methods, there is a way for one of the participants to easily figure out the other prices. All the malicious participant would have to do is guess the other bid prices and then run those values through a typical md5, sha1, or other traditional hashing algorithms to see if the hash values line up with the ones being stored. If the hashes do line up and the malicious participant was able to discover that the highest bid hash being stored is 30 dollars, then he or she would simply need to bid 31 dollars to win the auction. A Merkle tree prevents this scenario from happening by combining the

hashes of the other bids into multiple levels of hashes until it reaches a root node, which adds extra encryption.

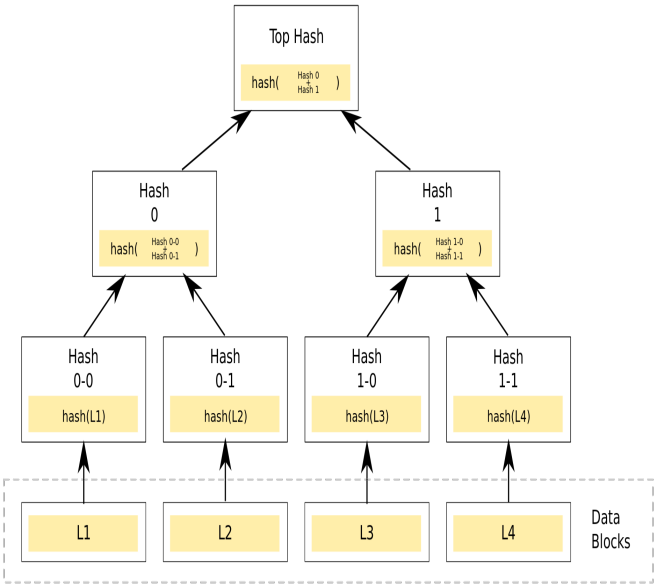

Fig. 1: The structure of a Merkle tree [2]

B. OpenMP

OpenMP is a C++ library that helps parallelize shared-memory programs. This library is comprised of library routines, environment variables, and compiler directives that influence the behavior of the running time. It does this through the use of threads and gives programmers a simple yet flexible interface for developing parallel programs. The main distinction between OpenMP and its close counterpart Message Passing Interface (MPI), is that OpenMP is used for parallelism within a multi-core node, whereas MPI is used for parallelism between nodes. The one that will most likely see more favorable results in terms of decreasing the running-time is MPI over OpenMP, however, MPI is more complex and not as intuitive as OpenMP.

One of the main reasons OpenMP was used for this project over MPI was to prove that by parallelizing a Merkle tree using the most straight-forward method, will show a significant decrease in the overall running time of the program. OpenMP will be an efficient way to parallelize a Merkle tree program by splitting up the workload on a single machine. Each node can have its own thread to calculate the value at that specific node. This will be a quick computation to do because the thread can easily read the children node's values to use in the parent node's hash function. Locks will need to be used so valid data is used at every level of the tree. The parallelization of the Merkle tree will be done using the OpenMP pragma operations on the primary region of the code that heavily influences the running time of the program.

C. Unified Parallel C

Although Unified Parallel C or UPC++ was not used in this project, it has the potential to show promising results in regards to future work and is thus worth mentioning. UPC is a high-performance computing extension of the C programming language that can be used for large-scale parallel machines. UPC utilizes a shared global address space along with distributed memory that gives the programmer the ability to use a single shared, partitioned address space. This allows for variables to be directly read and written by any processor, but the variables are also physically correlated with a single processor.

The amount of parallelism is fixed at the startup time of the program due to the use of a single program, multiple data computation models, which usually results in a single thread of execution per processor. UPC combines the control over the data layout and performance of the message passing programming paradigm, with the programmability advantages of the shared memory programming paradigm which makes it a powerful tool for parallel programmers. This allows for the workload of a program to not only be split between other processors, but also different machines with their own sets of processors. Out of all the methods mentioned so far, UPC++ would probably give the best results in terms of running-time and processing power compared to the others.

D. Related Work

There is a method described in [3] that differentiates from the current standard of using naive locking for Merkle tree updates of the entire tree. This method is known as Angela and is a distributed and concurrent sparse implementation of a Merkle tree. The method is distributed by utilizing Ray onto Amazon EC2 clusters, and then retrieves and stores the state using Amazon Aurora. The main aspect that Angela is motivated by is Google Key Transparency, which comes in direct inspiration from its underlying Merkle tree known as Trillian. Angela publishes a new root after some amount of time after assuming that a large number of its 2256 leaves are empty, which is the same task offered by Trillian. The approaches used by Google Trillian and the concurrent algorithm offered by Angela are compared, which shows nearly a 2x speedup.

Some other related research is stated in [4] which claims to be the first to provide complete, succinct, and recursive sparse Merkle tree definitions, along with related operations. These definitions show, when applied, that efficient space-time trade-offs for different caching strategies are enabled. It is also shown that utilizing SHA-512/256 to generate verifiable audit paths to prove (non-)membership, is done in nearly constant time which is less than 4ms. These results were concluded despite there being a limited amount of cache space, as well as there being a minimal effort of complete security embedded in the multi-instance setting. The size of the cache structure was smaller than the underlying data structure that was the target for authentication.

The paper [5] discusses hash functions derived from three modes of operation considering an inner Variable-Input-

Length function. The inner function mentioned can be a sponge-based hash function, or a prefix-free MD and single-block-length(SBL) hash function. This paper discusses numerous techniques used for optimization purposes pertaining to developing parallel hash functions derived from trees in which all the leaves possess the same depth. The first result is comprised of a scheme that optimizes the topology of the tree to decrease the running time. The second result shows that the number of required processors can be minimized by slightly modifying the corresponding tree topology, without affecting the optimal running time. Therefore, this technique proves to reduce not only the running time but the number of required processors as well.

The hardware cost of implementing hash-tree based verification of untrustworthy external memory via a high-performance processor is reviewed in [6]. Certified program execution can be a result of this verification enabling these types of applications. Multiple schemes are displayed offering different integration levels that are between the on-processor L2 cache and the hash-tree machinery. A set of simulations also display the best version of the performance overhead that is less than 25 percent as a result of these methods. This is a significant decrease from the naive implementation, which normally presents a 10x overhead result.

Authenticated Data Structures (ADS) are discussed at length in [7], which defines an ADS to be data structures whose operations can be carried out by an untrusted prover. This results in a verifier being able to efficiently check the authenticity of these operations. To create this scenario, the prover produces a compact proof which is then checked by the verifier, along with the results of each operation. Therefore, ADS supports the processing of tasks to untrusted servers without having to worry about the loss of integrity of the data, as well as outsourcing data maintenance. This paper also introduces a generic method that uses a simple extension to a programming language similar to what is used in machine learning algorithms, with which one can program authenticated operations over any data structure that is defined by standard-type constructors.

III. Password Storing Application Implementation

A. Program structure

The first part of the implementation procedure was to build the serial version of the program. This entailed creating two data objects: a Merkle tree class and a Node struct. The class members of the Merkle tree include a Node pointer variable, as well as a printTree and deleteTree function. Included in the header file of the Merkle tree class is the declaration of the audit function that is used to detect whether the tree has been tampered with. The members of the Node struct include a string variable to hold the hash values, two Node pointers for the parent nodes, and lastly a function that passes a string to hold as data, that being the passwords. The main file is set up with the leaves of the tree declared as a vector of Node pointers and a Merkle tree pointer is used to build the tree of leaves. Using a for loop, the data is assigned and hashed to the parent nodes in the leaves vector. The output of a simplified built tree can be seen in (Fig. 2) along with the result of the audit function.

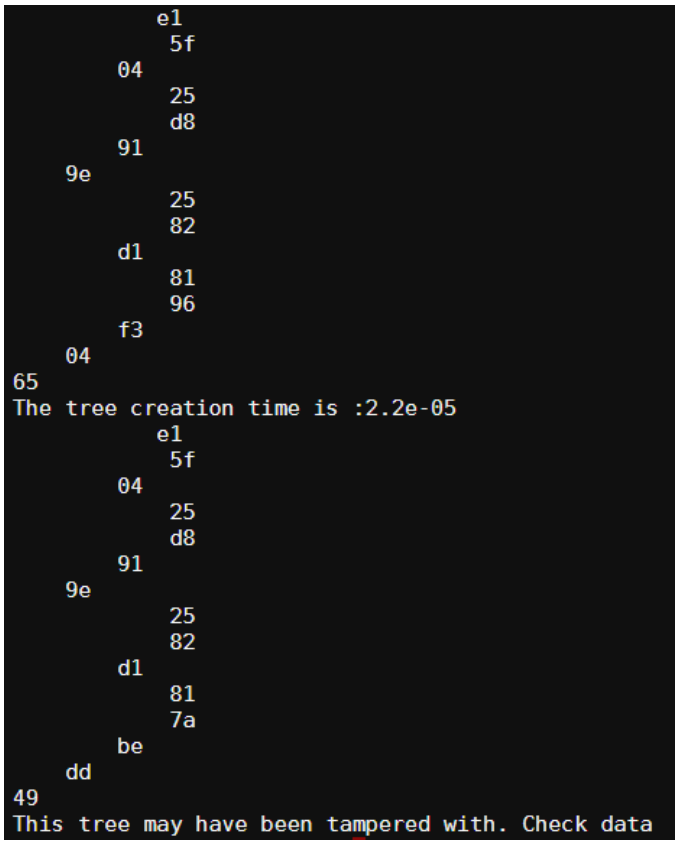

Fig. 2: The output of the first two characters of the Md5 hashes for a version of the Merkle tree program with a small number of nodes. The output also displays the result of the audit function.

B. Program functionality

TThe program operates by reading in strings of passwords from a file containing at max, 100,000 different passwords. It stores these passwords as the data that is used for the leaves of the Merkle tree. The passwords are hashed using the md5 hash library, after which the tree is fully constructed and output to the screen. This is the process for building one of the trees, but in order to test the validity of the program, two trees were constructed. The first tree is used to store all of the correct password information taken directly from the file whereas the second tree reads in the same passwords, except for altering one of the password values. The two root nodes of the trees are then compared using the audit function, which compares the second tree, that being the tree in question, with the first tree which is already known to be correct. If for any reason the root hashes differ, then it can be assumed that one of the password values is incorrect and the tree then becomes obsolete. A separate program was also included in a header file to keep track of the running times and print them to the screen.

```
        for (unsigned int l = 0, n = 0; l < blocks.size(); l = l + 2, n++) {
            if (l != blocks.size() - 1) { // checks for adjacent block
                nodes[n] = new Node(md5(blocks[l]->hash_val + blocks[l+1]->hash_val));
                nodes[n]->mom = blocks[l]; // assign children
                nodes[n]->dad = blocks[l+1];
            } else {
                nodes[n] = blocks[l];
            }
        }
      //std::cout << "\n";
      blocks = nodes;
      nodes.clear();

   }

   this->root = blocks[0];
}
```

Fig. 3: The serial implementation used in defining the Merkle tree.

```
      #pragma omp parallel
      {
          #pragma omp for ordered
           for (unsigned int l = 0, n = 0; l < blocks.size(); l = l + 2, n++) {
               if (l != blocks.size() - 1) { // checks for adjacent block
                   nodes[n] = new Node(md5(blocks[l]->hash_val + blocks[l+1]->hash_val));
                   nodes[n]->mom = blocks[l]; // assign children
                   nodes[n]->dad = blocks[l+1];
               } else {
                   nodes[n] = blocks[l];
               }
           }

      }
       //std::cout << "\n";
       blocks = nodes;
       nodes.clear();

   }

   this->root = blocks[0];
}
```

Fig. 4: The parallel implementation used in defining the Merkle tree.

C. Making the program parallel

The primary region of the program that influences the running time the most is located in the file where the Merkle tree class is defined. The differences between the parallel and serial implementations can be shown in (Fig. 3) and (Fig. 4).

```
[mreyes98@login006 final]$ ./main
The tree creation time is :0.00241
```

Fig. 5: The output of the running time for the serial program when fed 1,000 passwords.

```
The tree creation time is :0.001867
The tree creation time is :0.001816
The tree creation time is :0.001811
The tree creation time is :0.001816
The tree creation time is :0.001893
```

Fig. 6: The output of the running time for the parallel program when fed 1,000 passwords.

The main difference between the two implementations is the use of OpenMP in the parallel implementation, in which two pragma commands are utilized. The pragma omp parallel command is used to make the program parallel, and declare multiple threads to be run on a certain number of processors. The pragma omp for order operation is then used on the for loop to further optimize the parallelization of the program. The results of the two implementations can be seen in the next section.

IV. Performance Results

To test the difference in running time between the parallel and serial implementation of the program, different test cases were executed on the Bridges supercomputer. There were three different test cases in which the number of passwords fed to the program was administered. The number of passwords fed to both the serial and parallel program for the first test case was 1,000 which can be shown in (Fig. 5) and (Fig. 6), 10,000 for the second which can be shown in (Fig. 7) and (Fig. 8), and 100,000 for the third which can be shown in (Fig. 9) and (Fig. 10). The reason for choosing this method to test the differences in running time was to illustrate the effectiveness of the parallelization which could be more easily observed when increasing the number of nodes in the tree.

The results for the 1,000 password test case show that there is a significant decrease in the running-time in the parallel implementation when run on all the processors by a factor of .0001. When tested using 10,000 passwords, however, there

```
[mreyes98@login006 final]$ ./main
The tree creation time is :0.018375
```

Fig. 7: The output of the running time for the serial program when fed 10,000 passwords.

```
The tree creation time is :0.020607
The tree creation time is :0.020549
The tree creation time is :0.017677
The tree creation time is :0.019503
The tree creation time is :0.020123
```

Fig. 8: The output of the running time for the parallel program when fed 10,000 passwords.

```
[mreyes98@login006 final]$ ./main
The tree creation time is :0.192599
```

Fig. 9: The output of the running time for the serial program when fed 100,000 passwords.

```
[mreyes98@login006 final]$ ./testFile
The tree creation time is :0.185375
The tree creation time is :0.176204
The tree creation time is :0.171894
The tree creation time is :0.176826
The tree creation time is :0.178371
```

Fig. 10: The output of the running time for the parallel program when fed 100,000 passwords.

seemed to only be a decrease in the running-time in the parallel implementation when split between three threads, which was by a factor .001. The final case when tested using 100,000 passwords showed a significant decrease in running-time in the parallel implementation when running all processor threads by a factor of .01. These results show significant decreases in running-time when tested with the parallel implementation in all cases in at least one, or all of the processor thread options. All of the parallel implementation results show that the fastest running-time is achieved when the workload is split between three threads.

However, significant does not mean a large decrease in this context, it only means that it decreased enough to show that there were some optimization benefits to using OpenMP. A line graph of the running-times for each of the parallel implementations can be viewed in (Fig. 11), (Fig. 12), and (Fig. 13). The Merkle tree data structure is a complex target to optimize due to the already quick processing capabilities inherent in its structure. In retrospect, perhaps a method involving the use of UPC++ would prove to be a better option for decreasing the running-time even more, due to its ability to split up the workload between different machines rather than several processors on the same machine.

V. Conclusions and Future Work

A. Conclusions

These implementations of a Merkle tree, both the parallel and serial versions, show that Merkle trees can be used with success for certain aspects unrelated to blockchain applications. First, the serial version of the Merkle tree was built and tested three times with passwords ascending in quantity with each iteration, and the results of the running-time were recorded. The method of testing involved two Merkle trees, one that held the correct list of passwords and another with the same list with one of the passwords being altered. The two trees were then passed through an audit function to determine the authenticity of the trees. If the root hashes of the two trees differed in any way, then the audit function would return an error message that the tree had been tampered with. In order to parallelize the program, it simply took two OpenMP pragma operations to do so. Then, the parallel implementation was

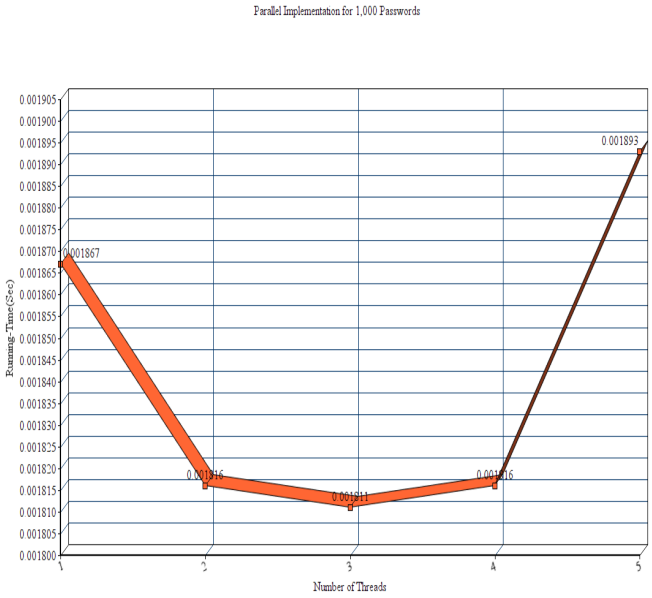

Fig. 11: Line graph of the running-time for the parallel implementation using 1,000 passwords.

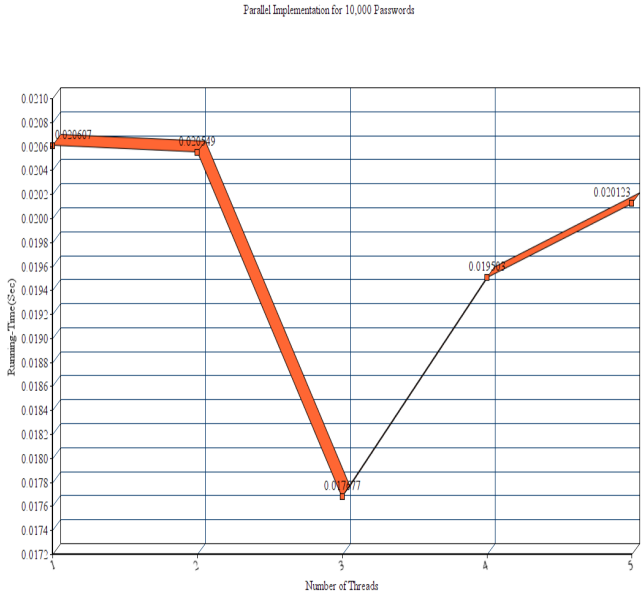

Fig. 12: Line graph of the running-time for the parallel implementation using 10,000 passwords.

tested using the same methods as for the serial version, and the results were again recorded.

Since there were multiple different running-times for the parallel version depending on the number of processor threads being used, the results were used to construct three line graphs for each of the test cases. The results showed a significant

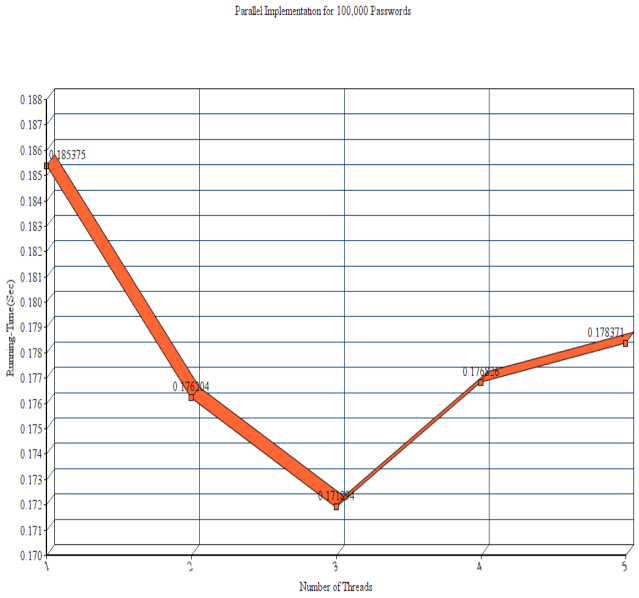

Fig. 13: Line graph of the running-time for the parallel implementation using 100,000 passwords.

decrease in running-time in the parallel version for at least one, or all of the processor thread options. The line graphs more clearly represent the data and it shows that the favorable processor thread amount for each of the test cases was three processor threads. For an unclear reason at the moment, the test case of running the program with 10,000 passwords only shows a decrease in running-time when run with three threads, every other processor thread option is slower than the serial version. The best way to remedy this situation may be to use a more effective parallel programming method like MPI or UPC++ to achieve greater results.

B. Future Work

Some future work to consider for this project could be to develop a parallel version of the same serial implementation as this Merkle tree, but using a more complex MPI version, rather than OpenMP, to see if the running-times can be reduced even further. It would also be beneficial to get better results for the 10,000 password test case as a result of using an MPI implementation, where there is a decrease in running-time from the serial implementation when run on all processors. OpenMP has the ability to parallelize the program by creating threads to run on multiple processors on a single machine, but there are other techniques like that of using UPC++ that can split the workload between different machines and thus, decrease the running time even more. According to the results recorded in this project, the optimal amount of threads to run the Merkle tree program on was three, each running on processors located on the same machine.

The next step for future work could also be to develop a full login application system, where this Merkle tree can be used in a real-life scenario where a user enters a username and password to log into a system, and if entered incorrectly, will change the root hash. A creative strategy can be used to integrate the root hash of a Merkle tree into the main functionality of the program. At the moment it is unclear what such a program would look like, but the result could be a far more secure system than the ones that are currently accessible to the public. Known exploits for log-in systems can be tested against this system that uses a Merkle tree to store the passwords and then it can be observed if this system is as vulnerable as traditional ones.

REFERENCES

[1] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[2] Wikipedia. Merkle tree. https://en.wikipedia.org/wiki/Merkle_tree, 2020. https://en.wikipedia.org/wiki/Merkle_tree.

[3] Janakirama Kalidhindi, Alex Kazorian, Aneesh Khera, and Cibi Pari. Angela: A sparse, distributed, and highly concurrent merkle tree. Technical report, UC Berkeley, 2018.

[4] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse merkle trees: Caching strategies and secure (non-)membership proofs. Cryptology ePrint Archive, Report 2016/683, 2016. https://eprint.iacr.org/2016/683.

[5] K. Atighehchi and R. Rolland. Optimization of tree modes for parallel hash functions: A case study. *IEEE Transactions on Computers*, 66(9):1585–1598, 2017.

[6] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *Proceedings of The Ninth International Symposium on High-Performance Computer Architecture, 2003 (HPCA-9 2003)*, pages 295–306, 2003.

[7] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. *ACM SIGPLAN Notices*, 49(1):411–423, 2014.