A Novel Approach to Password Cracking Using Round-Robin & MPI

Sean Rice[1], Judy Fan[2], Stanislav Ondruš[3]

***Abstract*— Passwords are a commonly used security feature to authenticate an authorized user for a variety of applications, such as a database. However passwords may need to be recovered by system administrators or passwords may need to be recovered in a computer system. As they are not (typically) stored in plain-text but rather hashed and salted, it is noted that hash comparisons are a computationally expensive task. This paper goes into detail about increasing the performance of password cracking by the parallelizing a rainbow table attack with MPI.**

I. INTRODUCTION

Within information systems, valid user authentication most commonly done by the use of a password. A user presents a user-name and a password which is then forwarded to a database for value comparison. The password is then processed by a hash function, where the output is compared to a stored hash within the system, associated with the user-name.

A well implemented hash function is one-directional, which means the hash cannot be reversed into plain-text by reversing the function, yet it provides a unique output . The one-directional attribute of hashes allows passwords to be stored safely on disk as hashes. Hash functions are used as users typically introduce very little randomness into their passwords by using familiar patterns and may reuse the same passwords on other systems. These familiar patterns are easily brute-forced.

Despite the predictability of users' passwords, the hashing of user input and comparisons against hash strings are a computationally expensive task. Hellman proposed a that a cryptosystem could have n keys could recover a key with $n^{2/3}$ words of memory after a precomputation which requires n operations. [1] However, as hash comparisons are done individually per password this is a task that is able to be implemented in parallel.

An example of a precomputation is the use of rainbow tables, proposed by Hellman. When in use, rainbow tables are faster than using a brute-force hash cracker, which involves a high computational cost for complex and long passwords. Rainbow tables are precomputed tables containing hash values paired to plaintext rendition of passwords in the form (startpoint, endpoint). [2] Each pair is chained with a hash function applied to the plaintext. Following the application of the hash function, a reduction is applied in an alternating fashion. An example of a chain is: $hash \rightarrow plaintext \rightarrow reduction \rightarrow hash$

[1]Rensselaer Polytechnic Institute, seanerice@gmail.com
[2]Rensselaer Polytechnic Institute, dear.porcelain@gmail.com
[3]Rensselaer Polytechnic Institute, stanley.ondrus@gmail.com

A. An Aside About IBM Blue Gene Q

The IBM Blue Gene Q used for the execution of our experiments is located in Rensselaer's Computational Center for Nanotechnology Innovations (CCNI). This particular system is fitted with 32,768 cores, 32 terabytes of RAM and 64 I/O nodes. [7]

B. A Small Aside About Security Ethics

Cryptoanalysis attacks such as rainbow tables should be conducted on systems where there is permission to do so. Our analysis was conducted on a database constructed and owned by ourselves. Even for practise, it is not advisable (or permissible by law) to carry such attacks on systems where permission is not granted, especially when there is sensitive user data stored on said system.

C. Team Contribution

This project is completed by Sean Rice, Stanislav Ondruš and Judy Fan. The concept of the parallization of a rainbow table attack with round-robin messaging was conceived by Rice. Majority of the code were crafted by Ondruš and Rice. This paper and research was mostly complied by Fan.

II. RELATED WORKS

There has been an increase in the investigation of cracking hashes and increasing the performance of password recovery tools with the use of multiprocessing. Further research has been conducted, especially given the increasing affordability and availability of high performance computing.

A. Improving the Performance of RainbowCrack with MPI

RainbowCrack is an implementation of a hash cracker with a time-memory trade-off algorithm, aptly named, rainbow table. This package contains a few tools that will be discussed: *rcrack* - rainbow table password cracker, *rtgen* - rainbow table generator and *rtsort* - rainbow table sorter. As the generation of rainbow tables and hash comparisons are computationally expensive, parallel processing can be integrated to improve the performance of such tools.[2] Sykes and Skoczen presented a parallel implementation of RainbowCrack using MPI on the Whale SHARCNET supercomputer. Several rainbow table generation methods were investigated to improve the speed of rtgen. These techniques included standard non-MPI, MPI gathering technique, MPI ranks writing to different (or same) file with either the MPI library file writing function or the use of *stdlib* for writing to file.

The generation and the sorting of the rainbow tables in *rtgen, rtsort* is where the bottleneck occurs in the execution

of RainbowCrack. As the generation of tables does not require interprocess communication, it is a viable candidate for parallization. On the other hand, sorting incurs a large communication overhead but each process is able to execute somewhat independently.

Sykes and Skoczen concluded that their MPI implementation drastically improved execution time of hash cracking, rainbow table generation and rainbow table sorting. Their implementation was able to complete the entire process of table creation for all alpha-numeric within 6 days by running multiple instances of their MPI *rtgen* for 32 processors on the SHARCNET system.[2] This would be a 400 percent speedup from the serial implementation of table generation in RainbowCrack.

B. Integrating CUDA Architecture

In a hybrid high performance computing approach to password recovery, it is possible to reach high scalability and increased performance gains, using MPI and graphic processing units with nVidia CUDA cores. Apostal, Foerster, Chatterjee and Desell researched three algorithms for the distribution of data to the GPU cores, where they found that using a divided dictionary approach yielded better performance. [5]

CUDA does not support parallized read/write operations and hence compared to MPI and even sequential execution, CUDA offers the worst performance in this aspect. [6] This is due to the fact that during the creation of every hash chain, a new CPU thread is created. In addition, communication needs to occur between the CPU and the GPU, adding further overhead.

Furthermore, one should consider the performance factors that can impact CUDA parallel programs, such as a password cracker implemented with CUDA. Such factors include integer arithmetic, memory access latency, data transfer, shared memory and register usage. Shared memory in CUDA cores is the most significant factor in the performance of CUDA programs. [3]

III. Design and Implementation

Our password cracker is compiled in C++ and the parallel Round-Robin communication is powered by the MPI framework. Benchmarking and execution of the cracker is conducted on the *IBM Blue Gene/Q* supercomputer.

The program utilizes custom written iterator libraries, that when initialized with the input, they return all possible hash combinations, one by one upon request. The main iterator library goes through multiple layers of combination iterators, to provide an exhaustive coverage over all possible combinations. Each provided word is used to generate all possible substrings within the provided length. Each of these substrings is then salted.

The salt generator uses even more layers of combinations, where the salt is being iteratively selected from all combinations of all alphanumeric characters within all specified lengths, and all pivots. Each of the salted passwords are hashed using SHA-256 function. The advantage using of this library is that it returns the unique hash dynamically - on request, which saves on memory usage.

The main operation principle relies on the implemented Round-Robin topology, that can be seen on Figure 1. For simplicity, only the scenario with 4 ranks is depicted and explained, even though the algorithm is highly scalable and can be used with different number of ranks.

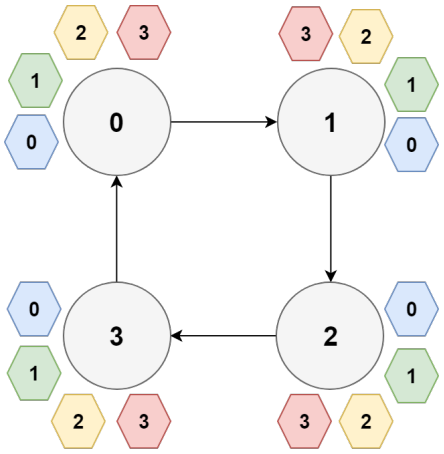

Fig. 1. Implemented topology

The light-grey, numbered circles represent individual ranks (0, 1, 2, 3), while the colored hexagons represent the array of hash structs that each rank is handling (i.e. each rank will store n local hashes, where $n = number\_of\_ranks$). The hash struct contains the hash, plaintext password, rank id of the generator, and the flag, that denotes whether the hash was matched with the database, together with the potential user id.

The program generates all possible hash combinations and test them against the database. The program contains a vector of strings (dictionary), together with other values necessary, used for the hash generation. Each rank is responsible only for their certain section of the dictionary. This is done to prevent duplicate processing and increases throughput. The same principle is applied on the actual database of passwords.

The database is stored in a file and specific sections of the database is being loaded by each rank into the map. The map uses the hashed password as a key, and user id as value. This ensures the complexity of *O(log(n))*. Due to the generic nature of the main algorithm, it is executed by every rank, and in the loop. On each iteration, each of the ranks requests a new hash from the iterator, and stores it in the hash struct array, under the index that matches the rank id.

Afterwards, it sends out the whole array to the succeeding rank, and performs receive operation on the previous rank. At this point, the each rank exchanges the array with its' neighbours, and new hashes that has not been matched yet with the local database. Therefore, each of the hashes within the array, whose internal "matched" flag variable is set to 0 is being checked against the database. In case the match is found, the flag is set to 1, and the user id is being stored

inside. At this point, the match can be recorded in a different structure. This whole principle repeats every iteration. This ensures that a hash is matched with a different part of the database each iteration, and once it finishes the round trip ($number\_of\_iterations = number\_of\_ranks$), it is discarded to prevent redundant matching.

The above described, chosen, algorithm is not the only that can be used in this scenario. One of the alternative topologies can be seen on Figure 2

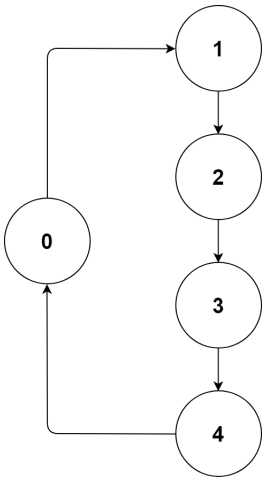

Fig. 2. Alternative topology

For simplicity, the figure above depicts only 5 total ranks, despite its' scalability. Each iteration, rank 0 generates a new hash to be compared, and sends it to the succeeding rank. This hash is being propagated through ranks 1-4, where each rank matches the hash with the corresponding section of the database, and forwards the hash further. Therefore, when the generated hash completes its' round-trip (returns back to 0), it has been matched with the whole database, and therefore it is possible to determine if the hash was valid or not. The advantage of performing this logic distributed across the ranks is that none of the ranks are idle at any point, since every cycle they match a different hash.

IV. Performance

The rainbow table generator and password cracker was executed the Blue Gene Q system on several node configurations, relative to the amount of MPI ranks to be used. Due to time constraints only a single (serial) rank up to 32 rank configurations were tested. The passwords embedded to our program are between 2 and 7 characters long. The salts generated are between 0 and 4 in length and include every permutation of the alphanumerics.

The results shown in Table 1 show an increase of reading time and cracking time as the amount of MPI ranks increase. However, it is important to note that in our implementation, the amount of passwords and hashes generated and read also increase in relative to the amount of ranks. Our program runs 5000 ticks per MPI rank. Hence, the amount of passwords generated and hashed is equal to: 5000 * *number of MPI ranks*

Ranks	Reading time (s)	Cracking time (s)	Cracks per minute
1	0.000905	1.224176	33773
2	0.001148	1.219409	42235
4	0.002263	1.367574	127315
8	0.001252	1.765964	228449
16	0.002476	2.248306	232785
32	0.004993	3.329699	490951

TABLE I
EXECUTION TIME OF OUR PASSWORD CRACKER

Another aspect of consideration is that message passing throughout ranks increases execution time, however the parallel nature of our implementation is enough to negate lost time that occurs through interprocess communication.

One should then consider the ratio between the execution time to crack the hashes and how many passwords overall were processed. Given that consideration, the speedup from serial to 32 ranks is 534% for hash comparisions. For parallel reading and writing operations, there is a 263% speedup.

V. Conclusions

Passwords are a pervasive method of user authentication used in many information systems such as databases, bank accounts and web interfaces. Plaintext passwords are stored as a hash and with a salt appended somewhere within the password.

Integrating MPI into the creation of rainbow tables and hash comparisons leads to a speedup compared to a serial implementation. There is a 5.34 speedup when conducting hash comparisons when MPI is implemented. When conducting read/write operations to the database in parallel, speedup of 2.63 is produced. This is a low-cost method of speeding up cryptoanalysis, even feasible to be conducted by a small group of computer science students. This shows that cryptoanalysis of passwords can be conducted more effectively. Moreover, systems with vulnerabilities that expose hashed passwords are not completely safe given that brute-force hash cracking is becoming more feasible, even on consumer hardware with the assistance of multiprocessing.

References

[1] M. Hellman, "A cryptanalytic time-memory trade-off," IEEE Transactions on Information Theory, vol. 26, no. 4, pp. 401–406, 1980.

[2] E. R. Sykes and W. Skoczen, "An improved parallel implementation of RainbowCrack using MPI," Journal of Computational Science, vol. 5, no. 3, pp. 536–541, 2014.

[3] I. L. S. Hendarto and Y. Kurniawan, "Performance factors of a CUDA GPU parallel program: A case study on a PDF password cracking brute-force algorithm," 2017 International Conference on Computer, Control, Informatics and its Applications (IC3INA), 2017.

[4] J. Bengtsson, "Parallel Password Cracker: A Feasibility Study of Using Linux Clustering Technique in Computer Forensics," Second International Workshop on Digital Forensics and Incident Analysis (WDFIA 2007), 2007.

[5] D. Apostal, K. Foerster, A. Chatterjee, and T. Desell, "Password recovery using MPI and CUDA," 2012 19th International Conference on High Performance Computing, 2012.

[6] Gómez J., Montoya F.G., Benedicto R., Jimenez A., Gil C., Alcayde A. (2010) Cryptanalysis of Hash Functions Using Advanced Multiprocessing. In: de Leon F. de Carvalho A.P., Rodríguez-González S., De Paz Santana J.F., Rodríguez J.M.C. (eds) Distributed Computing and Artificial Intelligence. Advances in Intelligent and Soft Computing, vol 79. Springer, Berlin, Heidelberg

[7] P. D. Barnes, C. D. Carothers, D. R. Jefferson, and J. M. Lapre, "Warp speed," Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation - SIGSIM-PADS 13, 2013.