Enhancing CII Firewall Performance Through Hash Based Rule Lookup

Pabilona Jaime Lee
Department of Electrical & Computer Engineering
National University of Singapore
Email: jaime.pabilona@u.nus.edu

Huaqun Guo
Institute for Infocomm Research
A*STAR
Singapore
Email: guohq@i2r.a-star.edu.sg

Bharadwaj Veeravalli
Department of Electrical & Computer Engineering
National University of Singapore
Email: elebv@nus.edu.sg

***Abstract*—It is important to develop defense mechanisms to bolster the cyber-physical security of critical infocomm infrastructure (CII) systems. A basic method of defense for CII systems is a firewall. Since SCADA / ICS systems may be negatively impacted by latencies and delays introduced by firewalls, which will translate to real world impacts, any implemented firewall in the network should attempt to minimize the latency it introduces. The latency in typical firewalls stems from packet classification, i.e. matching network traffic to firewall rules. It is this lookup time that we aim to improve through the development of a hash-based packet classification algorithm.**

Keywords-cyber-physical security; critical infocomm infrastructure; firewall; SCADA; hash table; packet classification

I. INTRODUCTION

Every day, millions of people rely on critical infrastructure to go about their daily lives without even realizing it, from public transport to the large-scale systems to provide resources such as water and electricity. A disruption to this critical infrastructure would have a huge impact on society, and by extension, the economy. With cyber and physical attackers becoming more and more advanced with each passing minute, defense mechanisms are needed to ensure that the cyber-physical security of critical infocomm infrastructure (CII) systems is maintained [1, 2].

In this paper, we will be focusing on the CII system, with the aim of developing a defense mechanism, in the form of a firewall, to prevent attackers who can manipulate the input from being able to disrupt operations. The firewall should be able to defend the system from a variety of malicious inputs, and since the system in question is crucial for everyday activities, the defense should take into consideration all forms of attackers, which can either be external actors or insider threats. The insider threats are the more difficult adversaries to defend against because they are able to not only tamper with inputs to the system, but also potentially physically tamper with components in the system.

To simulate the CII system, a switch, a remote terminal unit (RTU), and a programmable logic controller (PLC) are to be connected and configured to simulate certain system functions, with human machine interface (HMI) software to manage the connections. The goal is to create a software defined firewall, and place it in between the system components in order to detect and block attacks.

One of the key factors under consideration is the latency of any developed firewall solutions, since the CII system in question operates in real time, and any significant delay in firewall can have great ramifications for operations. Hence in this paper, the focus area on the firewall system is looking at ways to improve the performance of the lookup stage in packet classification.

The reason for focusing on trying to improve the lookup performance of firewalls is that after performing preliminary research, it was determined that most firewalls operate using a list of rules (or several lists of rules for different purposes, e.g. direction of traffic, protocol, action, and so on), with network traffic being matched to the rules in the list sequentially, i.e. the firewall would take the packet headers and compare them with each rule in the rule list sequentially until a matching rule was found to determine the appropriate action for the rule.

Especially in larger industrial networks, the number of rules employed by an organization could be very large, so it becomes clear that the worst case performance of the firewall would be the case where all network traffic only matches rules that are at or near the bottom of the rule list, due to the linear / sequential nature of packet classification. Therefore, any performance gains in the lookup stage would have big impact in the latency introduced by the firewall in the network. Consequently, the focus of the performance enhancement section of this paper is on improving the packet classification latency, by the design and performance evaluation of a novel classification / lookup algorithm. A hash-based packet classification algorithm was designed, along with several component algorithms, whose collision rate and memory usage properties were evaluated, with the results discussed in the performance evaluation section.

II. LITERATURE REVIEW

When performing research on methods to improve firewall performance, several concepts appeared often in multiple sources, including some papers which attempt to do comparisons between other papers / sources.

For firewalls that use basic lists of rules, the performance can be enhanced by performing rule compression or rule order

optimization, followed by the optimization of the matching algorithm, with linear, geometric, heuristic, and dynamic methods of matching [3]. Rule compression pertains to reducing the number of rules in the firewall rule list by performing analysis on the rules themselves, removing redundant (or unreachable) rules, and combining rules where possible. Rule order optimization refers to making sure the order of rules is such that the entire list is in order of decreasing probability while not changing the meaning of the rules, since the ordering of the rules oftentimes is related to the priority / importance of the rule.

For matching algorithms, several interesting concepts drew our attention, including database methods, statistical/dynamic methods, and hash-table based packet classification.

Database methods were promising since there are many existing algorithms built around optimizing the searching process in large databases, since this is something that most modern networks benefit from, with everyone from service providers to cloud storage to even content hosting companies trying to improve performance for users while using less compute time to improve efficiency. MySQL [4], ODBC [5], and Berkeley DB [6] are database systems that either have implementations in C, or have APIs for the C language, which is used by most ICS (Industrial Control system) / SCADA (Supervisory Control and Data Acquisition). In particular, MySQL and Berkeley DB are still being used and actively developed in the present day. However, there does not seem to be much open source data available to suggest the latencies that would be expected when performing queries, as well as any associated overheads with running / storing the database system either on the same machine as the firewall or on another machine in the network (which would also add additional network latency in the form of transmission and propagation delays).

Statistical / dynamic methods of performance improvement are centered around using a statistical or learning method to optimize the packet classification of the firewall, by either analyzing the frequency of the packet fields/headers as well as how often the rules are hit to build an alphabetical tree [7] , or by analyzing network traffic to detect flows and optimizing the firewall rule order such that rules that are matched often in a given flow are moved as far up the rule list as possible during the duration of the flow [8]. These methods seem interesting, and have been shown to achieve anywhere from a 10% increase in performance to a 60% increase in performance.

Last but not least, hash tables can be used to improve the performance of firewalls by storing the rules as tuples, which can be looked up in an efficient manner [9]. However, one of the highlighted issues is that collisions from the different hashing algorithms used can cause the hash table methods to suffer in performance. It is also pointed out that storage and memory requirements are an important factor for this family of methods, since hash tables may use a large amount of both.

III. Packet Lookup Algorithm Design

In this paper, a basic design was formulated around the hash table based concepts, in which the lookup time for the firewall would be constant time. The main idea is to take the headers of network packets (or rules) and pass them through a hashing algorithm, generating an integer which would then be used as the index number in a one dimensional array containing the action to be performed on the packet (allow, drop, reject, log, or other actions). Essentially, the only computation time needed would be that to compute the hash of a network packet, after which it would be possible to find the associated action by simply reading the array's contents at the output index number.

To begin with, a suitable hashing mechanism would need to be selected, based on the aforementioned criteria when dealing with hash table based firewalls (and also just firewalls in general), namely the memory usage, which is determined by the final array index size, as well as the security and collision characteristics of the algorithm. Research showed that most hashing algorithms have a fixed digest length, with more secure hashing algorithms in general requiring a longer digest length, and hence a larger memory requirement.

Non cryptographic hashing algorithms like MD5 (supposedly cryptographic in design, but proven to be broken) have shorter digest lengths, which would potentially mean a smaller memory / storage footprint for the array, but having an insecure hashing algorithm determine which rule a packet gets matched to would defeat the purpose of having a firewall, since it would then be possible for an attacker to generate a malicious packet that collides with a rule whose action is "allow". Therefore, we turned to SHA-3, the latest cryptographic hash specification from NIST, released in 2015.

More specifically, the algorithm in question is the SHAKE128 algorithm, which is one of several variants of SHA-3. The reason for this choice is because SHAKE128 has the peculiar ability of producing an arbitrary length digest. This property has two advantages, the first of which is the ability to modify the number of bits in the digest, which would then allow us to manipulate the memory usage of the array. Secondly, after inspection of the SHAKE128 implementation source code in C, if it is determined through experimental results that only a small number of bits is needed in order to ensure a low collision rate and reasonable memory usage, the computation speed would be much greater than a fixed length SHA-3 function (e.g. SHA3-224 to SHA3-512). This is because there would be fewer calls of the KeccakF1600_StatePermute function, since the length of digest output determines how many times this function is called.

The next step in the design was to determine the algorithms for actually generating an array index number from a packet's digest. Four methods are proposed, with the performance of each to be compared in the next section.

A. Method 1 - Digest Character Integer Addition

The first method is to take each character byte from the digest and add their ASCII values together, using the final sum as the array index number.

B. Method 2 - Digest Character Integer Concatenation

The second method is to take each of the digest character's ASCII values and concatenate them to form one long number, followed by some form of truncation or division to make the array index size reasonable.

C. *Method 3 - Digest Character Bitwise XOR*

The third method is to do a bitwise XOR between the ASCII values of each character in the digest to come up with the final array index number. As we will see in the performance evaluation, this method has a high collision rate, and as such a fourth method is devised.

D. *Method 4 - Digest Character Squared Bitwise XOR*

The last method is an extension of the third method, with the additional step of squaring the ASCII value of the digest characters, increasing the index size in an attempt to reduce the collision rate. Division or other means of truncation would then be possible to reduce the index size in a fine grained / controlled manner for the purposes of reducing memory usage if the resulting index size is too large.

Figure 1 depicts the steps the firewall takes in order to determine the action that will be applied to a network packet.

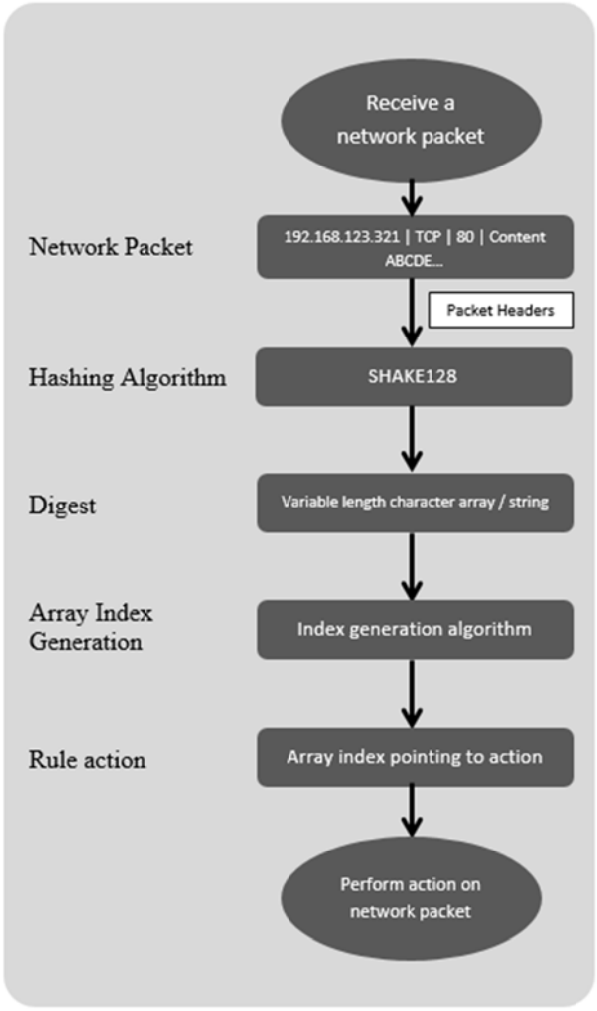

Figure 1. Final packet lookup design schematic

IV. Performance Evaluation

In order to measure the performance of the various methods of generating the array indices, several variables would have to be examined as necessary, namely the number of rules in the list, the number of bits for the SHAKE128 digest output, as well as the order of magnitude of any form of truncation method (e.g. division).

The main metrics for this section are the memory usage of the hashed rule / action list (indicated by the size of the array index) and the number of collisions for the various permutations of the variables listed above. These metrics are a direct result of experimental data. However, another interesting pair of metrics to compare between the methods are the collision rate, defined here as the number of collisions as a percentage of the total number of rules in the list, and the collision to size ratio, defined here as the number of collisions divided by the size of the generated array index.

We will be comparing the performance of the four previously discussed array index generation methods (Digest Character Integer Addition, Digest Character Integer Concatenation, Digest Character Bitwise XOR, and Digest Character Squared Bitwise XOR) in this section and discussing the results, noting that not all experimental data is shown in the interest of brevity.

Table I shows the results of Method 1 - Digest Character Integer Addition in terms of Number of Collisions and Index Size vs the number of rules and the output length of the digest of SHAKE128.

TABLE I. Collision and Index Size Results with Method 1

SHAKE128 output length	10		100		400	
Number of rules	Number of collisions	Index size	Number of collisions	Index size	Number of collisions	Index size
10	0	357	0	2851	0	2839
50	1	896	1	3719	0	5751
100	3	1053	3	4495	0	6073
200	33	1189	7	4587	2	8024
300	61	1192	13	4587	4	8751
400	101	1496	33	4587	16	8751
500	151	1496	50	4587	20	8751
600	204	1496	71	4587	32	8942
700	285	1543	98	4587	48	9087
800	365	1543	130	4587	62	9324
900	464	1543	167	4764	77	9324
1000	596	1579	199	4764	85	9324

We can see Method 1 results in a small index size, but also a large number of collisions. The index size grows as both the number of rules and the length of the digest increases. However, the index size grows logarithmically when compared with these two variables. At an output length of 10 digits, collision rate at 1000 rules is 59.60%, which is clearly unacceptable. Increasing the output length to 400 digits brings the collision rate down to a more reasonable, but still high 8.5%. As such, it is possible to keep increasing the number of digits of SHAKE128 output in order to reduce the collision rate, but at a computational tradeoff.

Table II shows the results of Method 2 - Digest Character Integer Concatenation in terms of Number of Collisions and Index Size (normalized to use the same amount of memory) vs the number of rules and the output length of the digest of SHAKE128.

TABLE II. COLLISION AND INDEX SIZE RESULTS WITH METHOD 2

SHAKE128 output length	2		3		5	
Number of rules	Number of collisions	Index size	Number of collisions	Index size	Number of collisions	Index size
10	0	249012	0	201724	0	203182
50	0	249041	0	201816	0	204733
100	0	253150	0	204035	0	205052
200	0	253208	0	204035	0	205052
300	0	253215	1	205198	0	205148
400	0	253215	1	205198	1	205802
500	0	254074	1	206718	1	205802
600	1	254982	2	206718	2	205910
700	3	254982	4	206718	3	205910
800	3	255065	4	206750	5	206655
900	3	255129	4	206750	6	206655
1000	6	255129	6	206750	8	206655

Since this method concatenates the ASCII values, we can see that the index size will grow by three digits (or orders of magnitude) for each bit of digest length. As such, to prevent the index size from growing too large, we can reduce the size of the index, by dividing the entire index by a random integer that is the same order of magnitude as how much we want to reduce the index size by. This is done instead of simply dividing by powers of 10, which effectively truncates the index, since random numbers provide fewer collisions.

We can also see from this method that even with very short digests, we can produce relatively few collisions, but with large index sizes, and that collision performance does not seem to improve as longer digests are used.

Table III shows the results of Method 3 - Digest Character Bitwise XOR in terms of Number of Collisions and Index Size vs the number of rules and the output length of the digest of SHAKE128.

TABLE III. COLLISION AND INDEX SIZE RESULTS WITH METHOD 3

SHAKE128 output length	10		400	
Number of rules	Number of collisions	Index size	Number of collisions	Index size
10	0	250	0	251
50	2	250	2	252
100	12	250	13	253
200	74	252	72	253
300	169	253	180	254
400	309	253	358	255
500	485	255	501	255
600	688	255	649	255
700	936	255	936	255
800	1197	255	1235	255
900	1507	255	1657	255
1000	1861	255	1914	255

From Table III, we can see that the index size is equal to the total number of ASCII character, and as such the index size is smaller than the number of rules at the higher end, hence there is the large number of collisions. The reason is that in Method 3, it is clear that the maximum index size is the same as the total number of ASCII characters and hence the giving index space is too small.

Table IV shows the results of Method 4 - Digest Character Squared Bitwise XOR in terms of Number of Collisions and Index Size vs the number of rules and the output length of the digest of SHAKE128. Differing from the simple XOR of Method 3, Method 4 squares each digest character before XOR-ing it with the next one. We can see that it is effective in increasing the size of the index and reducing the number of collisions. In addition, there is only a slight improvement as we use longer digests. Method 4 shows few collisions with a moderately large index size, showing little improvement when hash output is increased.

TABLE IV. COLLISION AND INDEX SIZE RESULTS WITH METHOD 4

SHAKE128 output length	10		400	
Number of rules	Number of collisions	Index size	Number of collisions	Index size
10	0	51336	0	50392
50	0	61952	0	61237
100	0	62040	0	64832
200	0	62555	1	64832
300	1	63621	1	65200
400	2	64217	1	65200
500	5	65052	1	65200
600	9	65055	1	65200
700	11	65284	7	65369
800	14	65284	8	65369
900	16	65284	11	65476
1000	20	65284	15	65476

In Method 4, the collision rate performance is good, especially when the size of the index is taken into consideration. We can also see that using a longer digest for this method yields very small performance improvements in terms of collision rate. As such, it would not make sense to generate long digests in an environment with finite computation power, as this would increase latencies without significant performance improvement.

We can see that Method 1 - Digest Character Integer Addition is the only method that grows in index size as the number of hash bits increases. The index size increase appears to be logarithmic with respect to the length of the hash output, which would mean that trading more computation time to generate a longer digest would be worth it to reduce the collision rate, which is unacceptably high at the larger ruleset sizes with a small hash output. However this would increase the delay between receiving a packet and being able to compute the hash, which would probably negate the constant time lookup provided by this firewall design.

Method 2 - Digest Character Integer Concatenation grows in terms of index size very rapidly as more bits are output from the hash, since each character adds 3 digits to the final array index number. This is why index reduction techniques need to be used in order to maintain a reasonable index size. However, using division or truncation to reduce the index size causes the collision rate to skyrocket for the larger ruleset sizes. If the memory usage of the firewall is not a concern, the concatenation method is certainly an appealing option, given that even at extremely low hash bit lengths, it can produce results with very low collision rates, meaning less time spent in computing the hash and less latency.

Method 3 - Digest Character Bitwise XOR has a fatal flaw, which is that the maximum array index size is equal to the number of ASCII characters available, which is 255. Clearly, once the ruleset size is increased past 255, there is bound to be collisions, hence the development of the 4th array index generation method.

Method 4 - Digest Character Squared Bitwise XOR is quite attractive, since the results show that there is very little improvement in terms of collision rate as the number of hash output bits is increased. This means that, like the concatenation method, a small number of output bits can be used to produce the array index. Furthermore, the performance of the method is great, producing low collision rates at index sizes which can be easily scaled up or down to meet differing memory requirements. Squaring the digest characters' ASCII values before XOR-ing them is also a simple computation that should not introduce a large delay.

When compared to Method 1 - Digest Character Integer Addition, Method 4 - Digest Character Square-XOR outperforms it with fewer collisions, unless the length of digest used is past the 200 digit mark, which is a large computational penalty. Method 4 - Digest Character Square-XOR also outperforms Method 2 - Digest Character Concatenation when index size is similar, trading a small collision rate penalty for a greater reduction in memory usage (index size). Method 1 - Digest Character Integer Addition should only really be considered if the computation time needed to generate the hash for the large number of output bits is not important for the system.

V. CONCLUSIONS

In this paper, a novel firewall design is discussed with a constant lookup time so as to reduce the latency in typical firewalls in matching network traffic to firewall rules. The main idea is to take the headers of network packets and pass them through a hashing algorithm, generating an integer which would then be used as the index number in a one dimensional array containing the rule action to be performed on the packet. In other words, the header of network packet is used as input of the hashing algorithm and the output is the index number of a one dimensional array containing the rule action to be performed on the packet. In this way, the delay of matching network traffic to firewall rules is very little.

Four methods, namely Digest Character Integer Addition, Digest Character Integer Concatenation, Digest Character Bitwise XOR, and Digest Character Squared Bitwise XOR, are presented and evaluated in terms of Number of Collisions and Index Size vs the number of rules and the output length of the digest of SHAKE128. The performance evaluation shows that when compared to Method 1 - Digest Character Integer Addition, Method 4 - Digest Character Square-XOR outperforms it with fewer collisions, unless the length of digest used is past the 200 digit mark, which is a large computational

penalty. Method 4 - Digest Character Square-XOR also outperforms Method 2 - Digest Character Concatenation when index size is similar, trading a small collision rate penalty for a greater reduction in memory usage (index size).

A possible extension of our work is to add a second dimension to the array, for the sake of dealing with the inevitable collisions that occur in real life. For example, colliding rules could be added to the array in the same row but in different columns. The packet classification algorithm could then implement a linear search in all the columns in a given row for packets whose digest points to a row with collisions. The performance of such a system could also then be tested. This idea is of course, only feasible if appropriate index generation methods are used to ensure that the collision rate for the system is low enough such that there would not be too many collisions requiring a linear search, or too many columns for a given row.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation (NRF), Prime Minister's Office, Singapore, under its National Cybersecurity R&D Programme (Award No. NRF2014NCR-NCR001-31) and administered by the National Cybersecurity R&D Directorate. The special thanks are also given to SMRT Trains Ltd for providing domain knowledge and technical support.

REFERENCES

[1] I. Nai Fovino, A. Coletta, A. Carcano and M. Masera, "Critical State-Based Filtering System for Securing SCADA Network Protocols," in IEEE Transactions on Industrial Electronics, vol. 59, no. 10, pp. 3943-3950, Oct. 2012.

[2] H. Rasha G. Mohammed, S. Tarek, E. Khaled, and M. Ausif, " Data Mining Based Network Intrusion Detection System: A Survey," in International Journal of Application or Innovation in Engineering & Management, vol. 2, Issue 3, pp. 338-343, Mar. 2013.

[3] A. Kolehmainen, "Optimizing firewall performance", Helsinki University of Technology, January 2007. Retrieved from http://www.tml.tkk.fi/Publications/C/23/papers/Kolehmainen_final.pdf

[4] Oracle Corporation, 2017, Version 26.8 MySQL C API, retrieved from https://dev.mysql.com/doc/refman/5.7/en/c-api.html

[5] Manoj Debnath, Database Programming with C/C++, July 11, 2016. Retrieved from: http://www.codeguru.com/cpp/data/database-programming-with-cc.html

[6] Oracle Corporation, 2017, Oracle Berkeley DB 12c overview http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html

[7] Hazem Hamed and Ehab Al-Shaer. 2006. Dynamic rule-ordering optimization for high-speed firewall filtering. In Proceedings of the 2006 ACM Symposium on Information, computer and communications security (ASIACCS '06). ACM, New York, NY, USA, 332-342.

[8] H. Hamed, A. El-Atawy and E. Al-Shaer, "Adaptive Statistical Optimization Techniques for Firewall Packet Filtering," Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications, Barcelona, Spain, 2006, pp. 1-12.

[9] V. Srinivasan, S. Suri, and G. Varghese. 1999. Packet classification using tuple space search. In Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM '99). ACM, New York, NY, USA, 135-146.