TriBiCa: Trie Bitmap Content Analyzer for High-Speed Network Intrusion Detection

N. Sertac Artan
ECE Department
Polytechnic University
Brooklyn, NY
(sartan01@utopia.poly.edu)

H. Jonathan Chao
ECE Department
Polytechnic University
Brooklyn, NY
(chao@poly.edu)

Abstract—**Deep packet inspection (DPI) is often used in network intrusion detection and prevention systems (NIDPS), where incoming packet payloads are compared against known attack signatures. Processing every single byte in the incoming packet payload has a very stringent time constraint, e.g., 200 ps for a 40-Gbps line. Traditional DPI systems either need a large memory space or use special memory such as ternary content addressable memory (TCAM), limiting parallelism, or yielding high cost/power consumption. In this paper, we present a high-speed, single-chip DPI scheme that is scalable and configurable through memory updates. The scheme is based on a novel data structure called TriBiCa (Trie Bitmap Content Analyzer), which provides minimal perfect hashing functionality. It uses a trie structure with a hash function performed at each layer. Branching is determined by the hashing results with an objective to evenly partition attack signatures into multiple groups at each layer. During a query, as an input traverses the trie, an address to a table in the memory that stores all attack signatures is formed and is used to access the signature for an exact match. Due to the small space required, multiple copies of TriBiCa can be implemented on a single chip to perform pipelining and parallelism simultaneously, thus achieving high throughput. We have designed the TriBiCa on a modest FPGA chip, Xilinx Virtex II Pro, achieving 10-Gbps throughput without using any external memory. A proof-of-concept design is implemented and tested with 1-Gbps packet streams. By using today's state-of-the-art FPGAs, a throughput of 40 Gbps is believed to be achievable.**

Index Terms—**TriBiCa, NIDPS, minimal perfect hashing**

I. Introduction

High-speed Network Intrusion Detection and Prevention Systems (NIDPS) have gained a lot of attention recently as part of the effort to keep up with the ever-increasing bandwidth requirement of today's networks. The most time-consuming task of NIDPS is Deep Packet Inspection (DPI). DPI also has applications in other networking areas, such as layer-7 switching, URL inspection, and spam, virus, and worm detection [1], [2]. DPI is the task of searching for a static or dynamic set of strings within each incoming packet. In the NIDPS context, DPI searches for pre-defined attack signatures in the incoming packets so as to identify malicious content.

Unlike most network applications, such as IP lookup and packet classification, whose complexity is proportional to the packet rate in packets/sec, DPI's complexity is determined by the data rate in bytes/sec, making it computationally harder than other applications. DPI's complexity is also increased by the number and length of strings in the set (signatures). As a result, the issue of designing a DPI system that is scalable in processing speed independent of the string set remains a challenge. Moreover, the application may have a dynamic signature set that is updated when necessary. Although in NIDPS these updates are relatively infrequent, the need to easily update the NIDPS signature set when required is still a challenge and often creates conflicts when designing a high-speed system.

Our goal is to design a high-speed, scalable, and easily updateable data structure to target these challenges. Specifically, the data structure will be small and its size scales with the number of strings and the average string size in the set. In addition, the updates can be achieved without hardware modifications. The proposed data structure, called TriBiCa (**Tr**ie **Bi**tmap **C**ontent **A**nalyzer), provides minimal perfect hashing functionality while intrinsically supporting low-cost set-membership queries. In other words, it provides at most one match candidate in the signature set that is used to match the query. It also filters out most of the irrelevant (*i.e.,* legitimate) traffic without referring to any string matching operation and thus increases the average search rate. Following the data structure, a hardware architecture is presented that tailors this data structure to the NIDPS. The proposed architecture fits into a fraction of a modest FPGA without the need for any external memory. More specifically, using parallel engines, the architecture can provide 10-Gbps throughput in the worst case on a Xilinx Virtex II Pro FPGA. If current state-of-the-art FPGAs are used, the proposed architecture can easily achieve DPI at 40 Gbps. The updates can be done through on-chip memory without any reconfiguration of the on-chip logic (*i.e.,* without any hardware modification), allowing faster response to new attacks. Avoiding external memory access not only improves speed, but also allows parallel designs to fit into the same chip.

The rest of this paper is organized as follows. Section II briefs the related work in high-speed hardware NIDPS. Section III describes the proposed data structure – TriBiCa. Section IV outlines the proposed NIDPS architecture based on TriBiCa. Section V analyzes the TriBiCa data structure and Section VI provides performance results. Section VII concludes the paper.

II. Related Work

Software DPI methods are not scalable for high-speeds [3] because general-purpose hardware running software DPI is intrinsically slow and has limited parallelism. Hence, here we only consider the hardware approach. Research to increase the DPI speed focuses on two aspects: (1) increasing the speed of unit inspection operation (*i.e.,* operation for each byte of the incoming packet) and (2) reducing the number of DPI operations by identifying possible malicious packets at the early stages of the inspecting process, while passing most of the packets that are legitimate.

To increase the speed of detection, some approaches use external memory structures [4], [5] such as TCAMs, SRAMs or both. The former is more expensive and consumes more power, while the latter suffers from speed limitation. Other approaches implement the DPI on a single-chip (most of the time on a single FPGA). The first generation of the single-chip solutions [6]–[10], with the exception of [10], tailor string matching circuits to the input set. Although it is high-speed, it requires hardware reconfiguration for updates. A recent proposal [11] takes a hybrid approach, using reconfigurable circuits and on-chip memory. The approach in [11] is similar to our proposal in that they also use perfect hashing (though not minimal perfect hashing). Unlike our proposal, [11] requires reconfiguration and has less than 100% signature memory utilization due to using perfect hashing rather than minimal perfect hashing. In [12], a minimal perfect hashing scheme is provided with $O(n)$ space complexity and low construction time. This approach, however, requires a complex addressing scheme, where additional logic is required to calculate the address in the hash table, to locate the signature for an exact match. Other recent proposals such as [10], [13], [14] also use on-chip memory for signature-specific data and for avoiding hardware reconfiguration for updates.

The pioneering work on single-chip methods without reconfiguration for signature updates [10] set the stage by modifying the classical Aho-Corasick String Matching Algorithm [15] for hardware implementation. Authors in [13] use small state machines to further improve memory requirements and fit the entire Snort [3] signature database to 0.4 MB memory. It is claimed that it can run at 10 Gbps with an ASIC implementation. It is noteworthy that the ASIC-based solution has a technology advantage over other proposals, most of which are FPGA-based. Authors in [16] later showed that FPGA implementation of [13] can achieve lower throughput while using larger memory. Authors in [14] use a sparse hash table to store signatures so that the hash collisions are minimized or, more likely, eliminated. Although the authors use indirection to improve memory utilization, it is still lower than many other proposals. In addition, the authors use glue logic to detect long patterns, which may require reconfiguration for signature updates.

Since most of the incoming packets are legitimate, running DPI for every single byte of an incoming packet is overkill. Methods exploring this property of intrusion detection were proposed to skip most of the legitimate packets through simple and fast pre-processing [17]–[19], thus significantly reducing the string matching operation that allows few queries before attempting any string matching. However, these methods still require additional full string matching for suspicious data, and do not improve the worst-case performance.

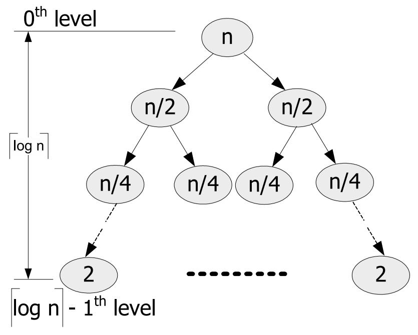

Fig. 1. Binary trie of TriBiCa. The value in each node shows the maximum number of items in that node. At each level l, there are 2^l nodes, each with a disjoint set of items from the set S. The total number of items in each level is always n.

III. TriBiCa

A. TriBiCa Data Structure

This section presents TriBiCa, our proposed data structure. Suppose a set S with n items is stored in a table with exactly n slots and each slot stores a single item from S. Our objective is to design a data structure that can represent this set S and respond to a membership query on S (1) by stating there is a possible match or not, and (2) if there is a match, pointing to a single possible match candidate in S (*i.e.,* pointing to a single table slot) without any prior exact matching between the query and the items in the set. To achieve the latter objective requires finding a minimal perfect hash function for S, which maps each item of S into one of n consecutive integer values in the table without any collisions. To achieve the former objective, the data structure should skip most, if not all, of the non-element queries by providing a simple, low-cost set-membership query mechanism.

TriBiCa achieves minimal perfect hashing by carefully partitioning the n items into equal-sized groups at each level so that at the end, all n items are uniquely partitioned, *i.e.,* one item per group. Let us start with a binary trie with $l = \lceil \log(n) \rceil$[1] levels where the root node is at level 0 as shown in Figure 1. The root node of this trie has all the n elements of S. To simplify our discussion, let us assume n is a power of two. Then arbitrarily partition n items into two equal-sized groups ($n/2$ each) and put one group to the left child node and the other to the right child node. Let us assume that there exists a partitioning algorithm that can do just that. Let us also assume that this algorithm will provide a query mechanism such that the correct group of a query item can be determined. These algorithms are described in Section III-B. This operation is

[1]All logarithms in this paper are in base 2.

then repeated in a recursive manner for each child node. More specifically, each child node will inherit precisely half of the items that belong to its parent. When a leaf node is reached, there will be two items in this node. The algorithm completes its partitioning by designating one of the items in each leaf node as the left child and the other as the right child. The path traversed from the root to each leaf node followed by the direction of an item in that node (left or right) is unique and thus defines a unique ID for each element of the set S. These IDs are used to address the table where S is stored providing a minimal perfect hash function for S.

A TriBiCa node is shown in Figure 2. Each node consists of a *data bitmap (DB)* and a *next node bitmap (NB)*. The DB indicates whether a given item is stored in this node. If this item is stored in this node, then the NB shows the child node inheriting this item. Both bitmaps have equal sizes and are addressed with a single universal hash function. We call each NB/DB bit pair in a node a *bin*. All nodes at the same level have equal bitmap sizes and are addressed with the same hash function. To insert an item into a node, we simply hash the item with this hash function, and set the bit corresponding to the hash result in the DB. The corresponding NB bit will show which child (left = 0 or right = 1) node inherits this particular item. As an example, Figure 3 shows a TriBiCa with 3 levels and 8 items (I_1 - I_8). These items are first hashed to the root node. Half of the items in the root node (I_1, I_2, I_4 and I_5) are inherited by the left child ($NB = 0$) and the rest by the right child ($NB = 1$). The same operation is repeated in the child nodes (*e.g.,* I_5 first goes to right ($NB = 1$) and then to the left ($NB = 0$)). The path traversed is encoded by the NB values and determines the address of the item in the table. The partitioning algorithm determines the content of NB. All items hashed to the same bin in a node share the same next node information since they share the same NB bit. So, items hashed to the same bin in a node must go to the same child node. We call this constraint the collision constraint and any partitioning algorithm should follow this constraint.

When n is not a power of two, the binary trie will not be balanced and the partitioning will be left-aligned, *i.e.,* nodes at level l will be filled from left to right with $2^{\lceil \log(n) \rceil - l}$ items as long as there are enough items left. The remainder of the items will be put into the next node and the rest of the nodes (and their children) will be removed. Optionally, TriBiCa can be designed as a k-ary trie instead of a binary trie, where each node has k children (k may differ between levels). This option, however is not covered in this paper and is left as future work.

Now, let us show the set-membership query mechanism provided by TriBiCa, which has similarities with Bloom Filters (BF) [20]. As with BFs, TriBiCa allows low-cost set-membership queries through bitmaps; however TriBiCa bitmaps are embedded in a trie structure. Additionally, TriBiCa provides member identification with a response to a matched query, a feature that BFs do not provide.

Once all items are hashed into TriBiCa, TriBiCa is ready for membership queries. The input is first hashed to the DB of the root node. If the corresponding bit value is zero, then this

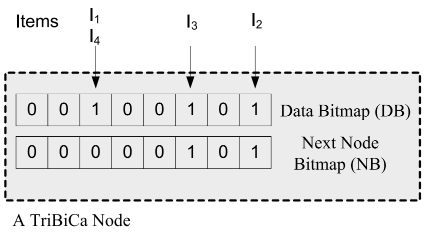

Fig. 2. An example TriBiCa node showing 4 items hashed

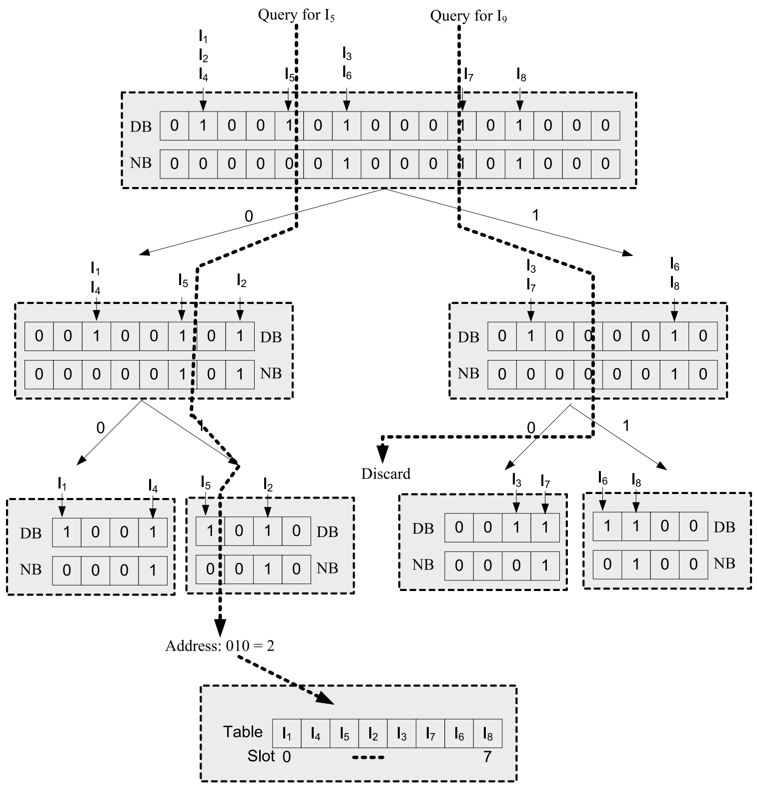

Fig. 3. An example TriBiCa with 8 items and 2 example queries. The first query, queries for item I_5 matches the item I_5 in the table. The second query, queries for a non-member, I_9, which is discarded by TriBiCa.

query item is not a member and it is discarded immediately, without further processing. On the other hand, if the bit is set to one, this query item may be a member of S. The corresponding bit in the NB will show the child to which to branch. The trie is then traversed until any node gives a no match (at least one node DB returns a zero) or the leaf node is reached. If at any node, the DB yields a zero, TriBiCa discards the input, allowing sub-linear processing time on average. Otherwise, if the input is hashed at all levels only to the bins with $DB = 1$, the NB values on the path will be used as an address to the table to retrieve the item. Then the input will be compared with this item for final membership decision, guaranteeing, at most, one string matching operation per input. A non-member input may cause a false positive, rendering one unnecessary string matching operation that resolves the false positive. These false positives determine the average

Bin No.	0	1	2	3	4	5	6	7
Occupancy	0	0	2	0	0	1	0	1
Item list	-	-	1,4	-	-	3	-	2

Fig. 4. Occupancy map for the example node in Figure 2

performance but do not impact the worst case (*i.e.,* the worst case is one string matching operation per query). In Figure 3, the query for I_5, which is in TriBiCa, gives matches in all nodes it is queried (all DB values are 1). The corresponding NB values through the path, 010, encodes the location of the query item in the table. To complete the matching, the item is fetched from the table and compared with the query. Finally, for the query with a non-member item, I_9 at level 1, the DB gives a false positive and TriBiCa discards the input. If this were a false positive, the query would not be discarded and would reach the final level. The final comparison with the item in the table would then determine that this item is not a member. If the average performance is not critical, then data bitmaps can be removed from TriBiCa nodes, reducing the memory to half without affecting the worst case.

B. Offline Partitioning Algorithms

As noted in Section III-A, a partitioning algorithm that guarantees a non-conflicting partitioning is required for TriBiCa to achieve minimal perfect hashing. In this section, we describe such algorithms. For offline partitioning algorithms described here, an auxiliary data structure for each node called an *occupancy map* is used. The occupancy map holds the occupancies for each bin (*i.e.,* the number of items hashed to that bin) in the node along with the list of items hashed to that bin. Such a map for the example node given in Figure 2 is shown in Figure 4. This map is not needed for online operation (*i.e.,* on-chip operation).

Following the collision constraint, partitioning the items in a node with η items into two[2] equal-sized groups, is simply partitioning the occupancy values of that node into two partitions with equal sums. In other words, let q be the number of bins with occupancies larger than 0 in a node and let Q be the set of these occupancy values. Then, finding two such disjoint subsets of Q each with sum $c = \eta/2$ is the same problem as partitioning the items in a node into two equal-sized groups under the collision constraint. This problem is equivalent to the classical number partitioning problem, which is NP-complete [21].

Fortunately, our instance of the number partitioning problem has a characteristic that helps us develop a good heuristic. In [22], Hayes argued that the best predictor of difficulty in the number partitioning problem is

$$\psi = w/\eta \tag{1}$$

where w is the number of bits required to represent the largest number in the set (*i.e.,* $w = \log M$ and M is the maximum occupancy value among bins of a given node for the purpose of our discussion). The maximum value in the occupancy map (that is the occupancy of the bin with maximum items) is low even for a load factor of $\rho = \eta/\mu = 1$, where μ is the size of each bitmap (NB or DB) for this node. It can be shown that the expected maximum occupancy for a load factor of 1 is [23],

$$E[M] = O(\log \eta / \log \log \eta) \tag{2}$$

From the two equations above, the difficulty of our partitioning problem instance decreases with η, since the expected value of M is sub-linear in η. As a result, for the high levels of the trie (*i.e.,* levels close to the root where η is larger), the problem is simpler, whereas it gets harder as we approach the leaf nodes. Intuitively, it is easier to find a subset with a desired sum selected out of many small numbers (high-level nodes), rather than finding it among a few large numbers (low-level nodes). So, for high-level nodes, a naïve algorithm is likely to find a solution, since it is likely that there are many solutions available. As we approach the leaf nodes, however, this is unlikely since there will be fewer solutions, if any at all. On the other hand, using a brute-force algorithm that is basically trying all possible subsets of Q to find if there is any possible equal partitioning is feasible at low levels. There are a total of $s = 2^q$ subsets for a given occupancy map. q cannot be larger than the sum of occupancies (*i.e.,* η), so s is bounded with 2^η. This bound, however, is not tight. Among these subsets only the subsets with a sum of $\eta/2$ is of interest for the purpose of equal partitioning. For instance, if a subset s_i has a sum larger than $\eta/2$, the brute-force algorithm will take into consideration any of s_i's supersets. Also, it can easily be observed that for a given η, as the number of collisions in the hash table increases, the number of subsets of Q decreases. So, if there are a few hash collisions, a brute-force algorithm is expected to reach an equal partitioning fast, since the number of subsets with sum $\eta/2$ is large. On the other hand, if there are many hash collisions, there are a few subsets so the algorithm covers all subsets faster, although with a lower probability of finding an equal partitioning. We next show an equal partitioning algorithm suitable for high nodes (Blackjack Algorithm) and another for low nodes (Greedy Algorithm).

C. Blackjack Algorithm

The naïve algorithm proposed here to partition high levels — The Blackjack Algorithm — is a straightforward algorithm relying on the fact that for nodes with many items *i.e.,* large η, the probability of failing to find an equal partitioning is slim. Besides, in most cases there are too many possible solutions. On the other hand, for nodes with such large η, the brute-force approach is not practical. For each node to be partitioned, starting from the leftmost bin among the occupied bins (*i.e.,* any bin, where DB value is 1) and walking to the right, the items in each bin are added to the left partition unless adding the next bin causes the size of the left partition to go over $\eta/2$ (thus the name, blackjack). If adding the items hashed to the

[2]If the trie is not balanced, then the sums at the remainder nodes will not be equal. If a TriBiCa is designed as a k-ary trie, the occupancy values should be partitioned into k partitions each with equal sums of n/k.

bin causes the size of the left partition to go over $\eta/2$, this bin is skipped (*i.e.,* added to the right partition) and the following bin is tried until a perfect size of $\eta/2$ is reached or all bins are tried. If the perfect size is reached, the remaining items will be put into the right partition and the partitioning is completed. Otherwise, the algorithm restarts from the second occupied bin on the left and retries. If all starting points are exhausted, either of the following two options can be selected. (1) For the nodes that failed to partition, their parent nodes are tried to be partitioned again with a different starting point to allow new options for the failed node. (2) A new hash function can be issued for the level, and partitioning of this level restarts again. This algorithm can be extended to a k-ary trie node by repeating the same operation for $k-1$ times for each partition except the last and with a target sum of η/k.

D. The Greedy Algorithm

For the low levels of TriBiCa, the solutions are not as abundant as for the high levels. It is specifically critical at the last level where to reach perfect hashing, there should not be any collisions in any node at all. The algorithm thus achieves perfect hashing by trying all possible solutions over the few low levels. As more levels are included in the search (*i.e.,* more levels are designated as low levels), the probability of reaching a perfect hashing increases as demonstrated in Section V. However, as more levels are included, the solution space increases substantially; thus, running time may increase unacceptably. For the application of the NIDPS for Snort signature database and the possible expansion of this database in the future, designating the last 4 levels (levels with 16 or fewer items in their most crowded node) as *low-level nodes* is a suitable selection for the given size of the problem as discussed in Section V.

To simply the discussion, we define a *super node* as a subtrie that is rooted from any node of the first level of low-level nodes and has last level nodes as its leaf nodes. The Greedy Algorithm works on each super node independently. The algorithm starts by partitioning the root node of a super node. To do this partitioning, the algorithm walks through all subsets of the set of occupancy values (*i.e.,* subsets of Q) to find a subset with sum $\eta/2$. The items in this subset are inherited by the left child. The remaining items are inherited by the right child. The algorithm then partitions the child nodes recursively in the same way as it partitions the root node until all nodes are partitioned successfully. If all the leaf nodes are free from collisions, the processing of this super node is completed successfully. If at some level, all the possible partitionings failed, the algorithm goes back one level up and finds the next valid configuration at that level and resumes recursive behavior.

Note that the partitioning should be successful for all the subtries rooted from the highest low-level to reach the goal of finding a minimal perfect hashing for the entire input set S. If any subtrie exhausts all partitionings without any success, either the parent high-level nodes are partitioned again to allow new options for the failed nodes, or a new hash function set will be issued for the low levels and the partitioning of low levels restarts from scratch. It can also be very effective to use a few additional hash sets for the failed super nodes as detailed in Section VI.

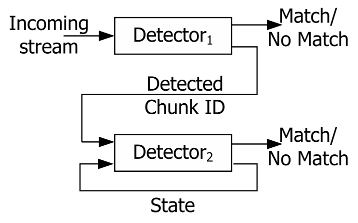

Fig. 5. A deterministic time string matching model

IV. TriBiCa for Network Intrusion Detection and Prevention

In this section, we first present an efficient string-matching model that achieves deterministic time matching using input independent state machines. Then, we show that the NIDPS architecture based on TriBiCa following this model achieves deterministic throughput. The model, similar to [4] and [24], emulates the behavior of the Aho-Corasick state machine, using a memory and a small fixed state machine. This way, the model can detect arbitrary length signatures, avoiding scalability issues for long signatures. As in previous work [4], [24], we start by chopping each signature into c-byte fixed size chunks. If a signature's length l is not a multiple of c, then the last $l \mod c$ bytes of this signature will be identified by a separate detector. Specifically, all signatures with length i, where $i \mod c \neq 0$ will share a detector to detect their final suffixes. For this purpose, $c-1$ detectors are needed.

To carry multiple-string matching over an input stream in deterministic time, the following conditions should be satisfied. (1) The string matching system should be able to detect the starting point of a signature in the payload, even when another match is in progress. (2) It should be able to follow a long signature until it is complete. (3) In case of a failure on detecting a long signature, the DPI system should be able to continue with the longest prefix detected (*i.e.,* the prefix that ends in the last detected chunk). (4) If a signature contains another signature, the contained signature should be reported whether the large signature is detected or not.

To satisfy the four conditions above, thus achieving deterministic detection time of any input for signatures with length l_c, such that $l_c \ mod \ c = 0$, the two detectors shown in Figure 5 that work on c-byte chunks of the input are sufficient.

In Figure 5, $detector_1$ is responsible for detecting c-byte chunks on a sliding window input. In case of a match, $detector_1$ reports a unique ID of the detected chunk to $detector_2$. If this chunk is actually a complete c-byte signature, a signature found alert is also issued. $detector_2$ is responsible for detecting larger signatures in a stateful manner. Receiving two matches, c-byte apart from $detector_1$, $detector_2$ looks for ID_1, ID_2 to match (a prefix of) a long signature consisting of $chunk_1$ and $chunk_2$. If there is a

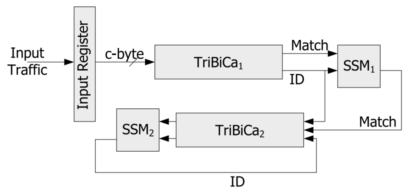

Fig. 6. TriBiCa for Network Intrusion Detection and Prevention

match, ID_1 is replaced with the match ID_{1-2} (representing the state "ID_1 followed by ID_2"). Next ID_{1-2} and ID_3 are queried in $detector_2$. If on the other hand, ID_1, ID_2 fails to match in $detector_2$, $detector_2$ continues with input ID_2, ID_3 to detect a possible new signature that starts with ID_2. This way, the beginning of the second signature is detected even when there is an ongoing match without any time penalty (satisfying conditions 1 and 4 above).

To follow signatures whose $2c$-byte or longer prefixes are a factor or a suffix of other signatures, one state is added as in [4]. For instance, let (1) ABCD and (2) BCEF be two signatures, where each character resembles a $c-$chunk byte. Then, an input string such as ABCEF will miss signature (2). However, if we add another state showing transition from ABC to CE then this will detect the second signature successfully. To be more specific, the new state should be added starting from the first location where signature (1) differs from signature (2) and points to the corresponding location in signature (2) (satisfying condition 3).

Note that for a sliding window input, c different stateful detections can be in progress for all c offsets, concurrently (between 0 and $c-1$). However, as pointed out in [24], by storing the state for each offset individually, this concurrent requirement can be satisfied.

There are signatures with length not multiples of c (*i.e.,* signatures with length $R = \alpha \cdot c + r$, where $\alpha > 0$ is an integer.). To detect these signatures, a similar two-stage approach is used for each $r < c$. The first stage ($detector_{r1}$) is responsible for detecting the $r < c$ byte signatures and $r < c$ byte suffixes. The second stage ($detector_{r2}$) detects signatures with length R bytes. The $\alpha \cdot c$ byte prefix is detected by $detector_2$ above resulting in a state of S_{cur}. Then $detector_{r2}$ makes one query with S_{cur}, ID_r, where ID_r is the ID of the r-byte suffix detected by $detector_{r1}$.

Based on this model, an NIDPS is designed using TriBiCa. TriBiCa along with an SSM (Single-String-Matching) Circuit is used to detect (S, chunk) type of queries for each stage in Figure 6. For the first stage, a TriBiCa that holds all c-byte chunks is used ($TriBiCa_1$). $TriBiCa_1$ is queried with each c-byte chunk of the incoming traffic. If it gives a match, it points to a single location in the corresponding signature table (not shown in the figure). The SSM_1 circuit then reads this signature and compares it with the input. If there is a match, the address of this signature is passed to the second stage as ID. The second TriBiCa works on ID pairs. All ID pairs involved in signatures are put into the second TriBiCa. The operation is similar to the first TriBiCa, other than the input type. If there is a match on SSM_2, this output is fed back to the second TriBiCa as the new ID. For signatures with length $r < c$ and for suffixes shorter than c-bytes, additional TriBiCas are used in a similar way. Note that, for certain cases (such as 1-byte signatures), trivial data structures (such as a bitmap showing matched characters) can be used [25].

V. Analysis of Minimal Perfect Hashing using TriBiCa

For a given input set, S, our main objective is to provide a minimal perfect hashing for this set S using TriBiCa. In this section, we analyze the probability of TriBiCa achieving such a minimal perfect hashing for a given set S. To achieve this goal, it is vital to find equal-partitionings for each and every node of the TriBiCa. We start by analyzing the probability of finding an equal-partitioning in a TriBiCa node. Let $\mathcal{N}(\eta, \mu)$ represent a TriBiCa node with η items and a next node bitmap (NB) of size μ. The simplest case of the equal-partitioning problem is when $\eta = 2$. In this case, an equal-partitioning is only possible if the two items occupy two different bins, which can occur with a probability of $(\mu - 1)/\mu$. To aid in the analysis for the general case (*i.e.,* when $\eta \geq 2$), let us define a *distribution* as any subset of the set of non-zero occupancy values (the set Q as defined in Section III-B). Furthermore, let us define a *configuration* as an equal-partitioning that can be achieved from a given distribution. If a total of d distributions $(D_0, \ldots, D_{d-1})$ are possible for a node $\mathcal{N}$ and each distribution D_j has c_j configurations $(C_{(j,0)}, \ldots, C_{(j,c_j-1)})$, then the probability that the distribution D_j occurs in node $\mathcal{N}(\eta, \mu)$ can be defined as

$$p_{D_j} = \frac{\mathcal{P}(\mu, q) \cdot \binom{\eta}{u_1} \cdot \binom{\eta - u_1}{u_2} \cdots \binom{\eta - \sum_{i=1}^{q-1} u_i}{u_q}}{\mu^\eta \cdot \mathcal{R}} \quad (3)$$

where $\mathcal{P}(\mu, q)$ shows permutation. (3) can be read as selecting q bins out of the μ possible bins in the node, then selecting u_i items from the input set to put into these bins, where at each step, the items are selected from the items remaining from the previous steps. The μ^η shows total possible ways to distribute η items into μ bins. It is possible that, one occupancy value, u_i can be repeated in more than one bin and re-ordering these bins in the distribution does not change the distribution. To avoid counting the same distribution more than once, a *repeat factor* $\mathcal{R}$ is added to (3). $\mathcal{R}$ is the product of all $r(u_i)$ factorials, where each $r(u_i)$ is the number of repetitions of the occupancy value u_i in the given distribution. c_j, the number of configurations corresponding to each distribution can be determined by counting the number of subsets of D_j that have a sum of $\eta/2$. The expected number of configurations (or equal partitionings) for this node can be given as in (4). Note that, for TriBiCa, since all nodes in the same level have same memory, $E[C]$ is the same for all nodes at the same level with the same number of items, η.

$$E[C_{\mathcal{N}}] = \sum_{j=0}^{d-1} p_{D_j} \cdot c_j \qquad (4)$$

For instance, for $\mathcal{N}(4,8)$ a four-item node, with a load factor of 0.5, there are five distributions, $D_1 =$`{1, 1, 1, 1}`, $D_2 =$`{1, 1, 2}`, $D_3 =$`{1, 3}`, $D_4 =$`{2, 2}`, and $D_5 =$`{4}`. Two of these distributions (D_3 and D_5) have no equal partitionings, *i.e.,* have no configurations. On the other hand, D_1 has three configurations ($c_1 = 3$). Any of the two items in D_1 can be partitioned into the same group resulting in an equal partitioning. There are $\binom{4}{2} = 6$ such groups. Since it does not matter whether the group is the left or right group, the total number of configurations for D_1 is 3. D_2 and D_4 provides one equal partitioning each ($C_{(2,0)} = \{(1,1),(2)\}$ and $C_{(4,0)} = \{(2),(2)\}$, where items in the same group are enclosed in parentheses.). From (3), p_j values can be calculated as $p_1 = 0.41016$, $p_2 = 0.49219$, $p_3 = 0.054688$, $p_4 = 0.041016$, and $p_5 = 0.0019531$. The probability that this node can give an equal partitioning for a random set of inputs is $P_{\mathcal{N}} = p_1 + p_2 + p_4 = 0.94$. The expected number of configurations for this node from (4) is 1.76.

This concludes the analysis for equal partitionings for a single isolated node. Now, let us look at the effect of the trie structure of TriBiCa in increasing the success probability of the equal partitionings. Suppose a set of items is programmed into TriBiCa and we want to find out the probability that we have equal partitioning for all nodes. The trie structure allows the items programmed into the nodes to be organized in different ways, based on the possible configurations in their ancestor nodes in the previous levels. For instance, for the example above, $\mathcal{N}(4,8)$ can have up to three configurations depending on its distribution. This means if an $\mathcal{N}(4,8)$ has distribution d_1, its child nodes will have three chances to find an equal partition if they are in the trie structure, instead of one chance if they were single isolated nodes. The Greedy Algorithm uses the trie structure to find an equal partitioning and eventually a minimal perfect hashing by trying all possible configurations at the low levels. High levels have a higher success probability of equal partitioning, as discussed in Section III-B, so the Blackjack algorithm relies on single node equal partitioning. The following analysis will focus on partitioning low-level nodes using the Greedy Algorithm to show the impact of the trie structure on achieving minimal perfect hashing with TriBiCa. Let $\mathcal{S}(l)$ represent a super node with l levels. Then, the success probability of achieving minimal perfect hashing for a set with 2^l items using a super node $\mathcal{S}(l)$ can be defined recursively with the initial condition of $\phi_0 = P_{last}$ as,

$$\phi_l = [1 - (1 - \phi_{l-1}^2)^{E_{root}}] \cdot P_{root} \qquad (5)$$

where P_{root} and P_{last} show the probability of equal partitioning for the root node and a last level node of $\mathcal{S}(l)$, respectively, if they were isolated single nodes. E_{root} shows the expected number of configurations at the root node of the super node. A TriBiCa with n items requires $n/2^l$ such super nodes. Assuming the probability of achieving equal partitions for all high-level nodes is P_H, the success probability for minimal perfect hashing with a TriBiCa using $\mathcal{S}(l)$ is,

$$\phi = P_H \cdot \phi_l^{n/2^l} \qquad (6)$$

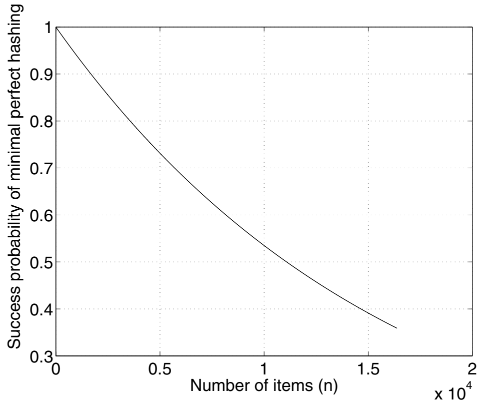

Fig. 7. The probability of achieving minimal perfect hashing successfully using TriBiCa for an arbitrary set of hash functions for the last 4 level with respect to number of items in the set, n.

Figure 7 shows the success probability of minimal perfect hashing with respect to the number of items in the set, where $\mathcal{S}(4)$ super nodes are used as low-levels. The effect of the high probability P_H is neglected. The load factor here in all levels is 0.5, except in the last level where it is 0.25. Since, the success probability is above 50%, even for 10 thousand signatures for TriBiCa, we choose to limit the low levels to four for the NIDPS application.

VI. Performance

In this section, performance of TriBiCa is investigated. First, simulations are carried out to characterize the construction of TriBiCa in C++. Then, the design is prototyped on an FPGA to show its runtime performance.

A. Simulations

In this section, the simulations investigating various parameters of TriBiCa construction (*i.e.,* achieving minimal perfect hashing for a given set of items) is presented. TriBiCa construction algorithms are implemented in C++ and tested on a Pentium 4 2.8-GHz computer with 512-MB memory. TriBiCa is constructed for three types of signature sets. First two sets are extracted from a Snort signature set with 1655 exact signatures. The first signature set, S_1, consists of 2724 4-byte signatures. These are $c = 4$ byte chunks of the Snort signatures. For NIDPS, this set is programmed into the first TriBiCa of the NIDPS, $TriBiCa_1$, which is described in Section IV. The second set, S_2, consists of 3226 ID pairs extracted from Snort signatures. This set is programmed into the second NIDPS TriBiCa, $TriBiCa_2$. The last set, S_3, consists of 65536 random integers. Each simulation below is

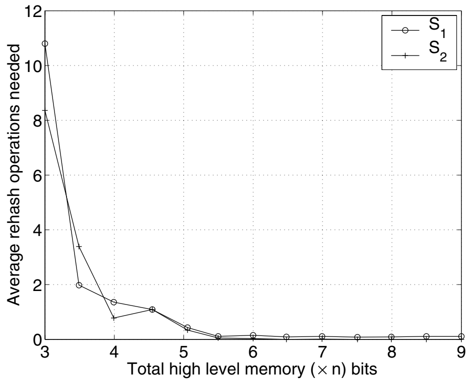

Fig. 8. Average number of rehashes required for different total high-level memory sizes

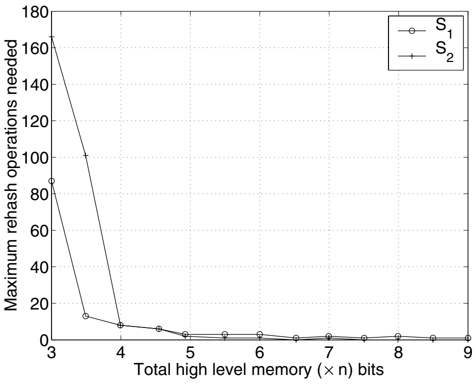

Fig. 9. Maximum number of rehashes required for different total high-level memory sizes

repeated for 100 different hash sets (*i.e.,* 100 trials). Then, average and maximum values of these trials are presented. Note that in these simulations, only the worst-case performance is considered, thus data bitmaps (DB) are not used.

In this first simulation, the partitioning of high-level nodes is considered using the Blackjack Algorithm. First, each high level is assigned a single hash function. If any node fails, a new hash function is assigned to the level of the failed node. Then, all nodes in that level are partitioned again with this new hash function. In Figures 8 and 9, average and maximum rehash operation requirements for all the high levels in each TriBiCa over 100 trials (100 trials for each memory configuration) are shown. For both sets, if the total high-level memory is at least $5 \times n$ bits, the number of average rehash operations required goes to 0.1 and the maximum rehashing requirement for these memory settings (*i.e.,* total high-level memory $\geq 5 \times n$ bits) is at most 3. The running times for these memory settings are all below 1 second.

In the second simulation, the whole TriBiCa is simulated; however, the focus is on the performance characterization of the Greedy Algorithm to partition super nodes. We verified that the effect of high-level memory on this performance is negligible so the total high level memory is fixed to $m_{high} = n$ for this simulation. Additionally, the last $l_l = 4$ levels of the

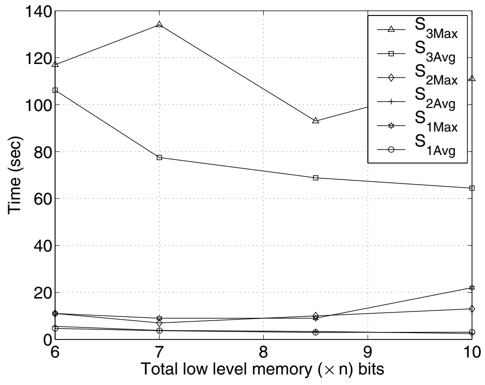

Fig. 10. Construction time for TriBiCa

TriBiCa are assigned to the super nodes. Thus, the Greedy Algorithm runs only on these nodes. The nodes above the last four levels are high-level nodes and are partitioned using the Blackjack Algorithm. For this simulation, first we run the Greeedy Algorithm on all super nodes using the same initial set of hash functions (*i.e.,* a total of l_l hash functions, one for each level of the super nodes). Then, any super node that fails to partition in this run is *rehashed.* This means a new set of l_l hash functions is assigned to these super nodes and the Greedy Algorithm is run again on these nodes only. This rehashing is repeated until all super nodes are partitioned successfully. In this way, instead of using l_l hash functions, we allow more hash functions to be used, but we limit the number of these hash sets by encouraging the usage of the same hash sets by different super nodes. If a total of H hash sets are used to partition the super nodes, then a hash set ID of $\log(H)$ bits is attached to each super node to represent the hash set assigned to this node.

Figure 10 shows the average and maximum construction times for each signature set over 100 trials each. Note that the total construction time for the first two sets, in other words the total construction time for the whole TriBiCa, is at most 35 seconds and 7.4 seconds on average. The construction time for S_3 is at most 134 seconds and 79 seconds on average. Figure 11 shows the average and maximum number of rehash operations required for super nodes for each signature set over 100 trials each. From Figure 11, it can be seen that for both Snort sets, the maximum number of rehash operations required is between 2 and 4, based on the total low-level memory used. For set S_3, the number of rehash operations required is never more than 6 and generally at or below 4. Considering even the worst case of 6 hash sets in addition to the original hash set, a 3-bit hash set ID per super node is enough to accommodate these different hash preferences. Since each super node consists of 4 levels, it can accommodate 16 signatures, thus the required additional memory for hash set ID, which corresponds to $3/16 \times n$ bits of memory.

Overall, from the above discussion, the two TriBiCas in the NIDPS design for the Snort signature set as given in Section IV can fit into a memory of 65.1 kbits, where 29 kbits $(5 \times n)$

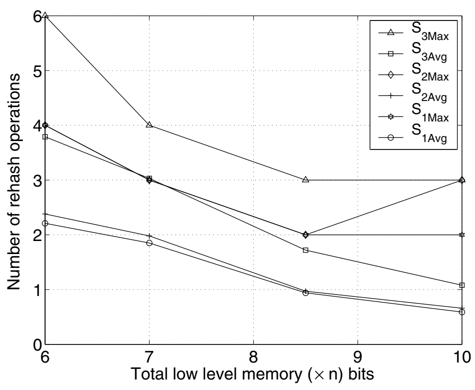

Fig. 11. Rehash operations required for low levels

are used for high-level nodes, and 36.1 kbits ($7.1 \times n$) are used for low-level nodes including the hash set ID storage. This design can be constructed around 8 seconds on average.

B. Hardware Implementation

We have designed the TriBiCa on a modest FPGA chip, Xilinx Virtex II Pro. The design can reach clock speeds over 300 MHz, achieving 10-Gbps throughput using 4 TriBiCa-based NIDPS engines working in parallel that can fit on this FPGA. The NIDPS architecture designed to detect the Snort signature set fits into less than 30-block RAMs (540 kbits) where most of the storage is dedicated to the actual signatures themselves. Each engine can process 8 bits per clock cycle. A proof-of-concept design is implemented and tested with 1-Gbps packet streams. By using today's state-of-the-art FPGAs, a throughput of 40-Gbps is believed to be achievable.

VII. Conclusion

This paper introduces a novel data structure, TriBiCa, for high-speed string matching. TriBiCa allows minimal perfect hashing of stored strings while providing low-cost set-membership queries to skip most of the irrelevant input. A Network Intrusion Detection and Prevention System architecture based on TriBiCa is proposed. It uses currently available, modest FPGAs to reach up to 10-Gbps worst-case throughput. The proposed architecture does not require any reconfiguration and all updates can be done through memory updates. The TriBiCa data structure can further be improved by using structural optimizations, such as using k-ary tries and by exploring the characteristics of signature sets such as character distribution of these sets. Additionally, the proposed data structure provides minimal perfect hashing on static data. Updates are done using offline tools and then writing the changes to the online memory. As a future work, operation on online updates will be explored. Finally, current NIDPS architecture supports exact signatures. Its extension to detect regular expressions will also be explored as a future work.

References

[1] C. Burns and D. Newman. (2006, Jan.) Vendors choose to tout disparate application-acceleration techniques. [Online]. Available: http://www.networkworld.com/reviews/2006/011606-wfe-features.html

[2] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," in *Proc. of the ACM/USENIX Symposium on Operating System Design and Implementation*, San Francisco, CA, Dec. 2004.

[3] [Online]. Available: http://www.snort.org

[4] F. Yu, T. Lakshman, and R. Katz, "Gigabit rate pattern-matching using tcam," in *Int. Conf. on Network Protocols (ICNP)*, Berlin, Germany, Oct. 2004.

[5] H. Song and J. Lockwood, "Multi-pattern signature matching for hardware network intrusion detection systems," in *IEEE Globecom 2005*, nov-dec 2005.

[6] J. Moscola, J. Lockwood, R. P. Loui, and M. P., "Implementation of a content-scanning module for an internet firewall." in *FCCM*, 2003, pp. 31–38.

[7] C. Clark and D. Schimmel, "Scalable pattern matching for high-speed networks," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, California, 2004, pp. 249–257.

[8] Y. H. Cho and W. H. Mangione-Smith, "Fast reconfiguring deep packet filter for 1+ gigabit network." in *FCCM*, 2005, pp. 215–224.

[9] Z. K. Baker and V. K. Prasanna, "High-throughput Linked-Pattern Matching
for Intrusion Detection Systems," in *Proceedings of the First Annual ACM Symposium on Architectures for Networking and Communications Systems*, 2005.

[10] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *Proc. of the 2004 IEEE Infocom Conference*, 2004.

[11] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A reconfigurable perfect-hashing scheme for packet inspection," in *Proc. 15th International Conference on Field Programmable Logic and Applications (FPL 2005)*, August 2005, pp. 644–647.

[12] Y. Lu, B. Prabhakar, and F. Bonomi, "Perfect hashing for network applications," in *IEEE Symposium on Information Theory)*, Seattle, WA, 2006, pp. 2774–2778.

[13] L. Tan and T. Sherwood, "Architectures for bit-split string scanning in intrusion detection," *IEEE Micro*, jan-feb 2006.

[14] G. Papadopoulos and D. N. Pnevmatikatos, "Hashing + memory = low cost, exact pattern matching." in *Proc.15th International Conference on Field Programmable Logic and Applications (FPL)*, August 2005, pp. 39–44.

[15] A. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[16] H.-J. Jung, Z. K. Baker, and V. K. Prasanna, "Performance of FPGA Implementation of Bit-split Architecture for Intrusion Detection Systems," in *Proceedings of the Reconfigurable Architectures Workshop at IPDPS (RAW '06)*, 2006.

[17] K. Anagnostakis, S. Antonatos, E. Markatos, and M. Polychronakis, "E2xb: A domain-specific string matching algorithm for intrusion detection," in *Proc. of the 18th IFIP International Information Security Conference*, 2003.

[18] S.Dharmapurikar, P.Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," in *Symposium on High Performance Interconnects (HotI)*, Stanford, CA, Aug. 2003, pp. 44–51.

[19] H. Song, T. Sproull, M. Attig, and J. Lockwood, "Snort offloader: A reconfigurable hardware nids filter," in *15th International Conference on Field Programmable Logic and Applications (FPL 2005)*, Tampere, Finland, Aug. 2005.

[20] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, 1970.

[21] M. Garey and D. Johnson, *Computers and Intractability: A Guide to NP-Completeness*. San Francisco, CA: Freeman, 1979.

[22] B. Hayes, "The easiest hard problem," *American Scientist*, vol. 90, no. mar-apr, pp. 113–117, 2002.

[23] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. The MIT Press, 2001.

[24] S. Dharmapurikar and J. Lockwood, "Fast and scalable pattern matching for content filtering," in *Symposium on Architectures for Networking and Communications Systems (ANCS)*, Oct. 2005.

[25] N. S. Artan and H. J. Chao, "Design and analysis of a multi-packet signature detection system," to appear in *Int. J. Security and Networks*.