High Throughput Signature Based Platform for Network Intrusion Detection

José Manuel Bande Serrano[1], José Hernández Palancar[1], and René Cumplido[2]

[1] Advanced Technologies Application Center, 7ma A ♯ 21406, e/ 214 y 216, Siboney, Playa, CP: 12200, Havana, Cuba
{jbande,jpalancar}@cenatav.co.cu
http://www.cenatav.co.cu

[2] Instituto Nacional de Astrofísica Optica y Electrónica, Luis E. Erro 1, Sta. Ma. Tonanzintla, Puebla, 72840, México
rcumplido@inaoep.mx

Abstract. In this work we propose the intensive use of embedded memory blocks and logic blocks of the FPGA device for signature matching. In our approach we arrange signatures in memory arrays (MA) of embedded memory blocks, so that every signature is matched in one clock cycle. The matching logic is shared among all the signatures in one MA. In addition, we propose a character recodification method that allows memory bits savings, leading to a low byte/character cost. For fast memory addressing we employ the unique substring detection, in doing so we process four bytes per clock cycle while hardware replication is significantly reduced.

Keywords: NIDS, string matching, content scanning, FPGA, unique substrings.

1 Introduction

Network Intrusion Detection Systems (NIDS) are designated to protect networks and services against attacks executed by insiders or outsiders. There are three kinds of NIDS: Signature-based, Misuse-based and Anomaly-based [1]. In Signature-based data flow is scrutinized in the search of attacks with signatures known beforehand. In Misuse-based signatures are automatically discovered through Supervised Learning methods. Finally, Anomaly-based, assumes that intrusions are, by nature, deviations from normal behavior. Of the three, only Anomaly-based intrusion detection is capable of detecting unknown attacks [1].

Although much progress has been made in Anomaly and Misuse-based detection, a fast and efficient signatures detection is still needed. The reason is that the types of NIDs exposed before, represent the natural mechanics of learning. This is, the unknown knowledge is perceived, then is characterized, and finally it becomes part of the current knowledge. In this integration, signature-based detection becomes into the first line of defense, because it makes, or helps to make decisions based on the current knowledge, as fast as possible [2].

J. Ruiz-Shulcloper and G. Sanniti di Baja (Eds.): CIARP 2013, Part II, LNCS 8259, pp. 544–551, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Since the speed of data streams will continue to grow for the next years, and fast responses to attacks are necessary in high security environments, signature matching is an active field of research. This demanding environment requires hardware solutions. In that direction, we propose a memory and logic based architecture where signatures are compressed and stored in memory arrays. Our matching logic allows the comparison of one signature per clock cycle. The entire signature set is partitioned. From each one of resulting subsets, only one signature is selected at each clock to be matched with the data flow. This is carried out by a predetection step. In order to store the entire signature set in memory we propose character recodification. In doing so the resulting architecture presents a better balance in the use of memory and logic, regarding other multi-character architectures.

The rest of the document is as follows. In the section two, we analyze the related works, paying special attention to those multi-character architectures. In the section three, the employed partitioning method is explained. In the section four, the architecture is presented. Section five is dedicated to experiments and comparisons with other works. Finally, conclusions are presented.

2 Related Works

Baker and Prassana in [3] proposed partitioning scheme that allows resource sharing in a logic-based architecture. Hardware implementation of the well-known string matching Shift-or algorithm is proposed in [4]. In [5], the well-known Aho-Corassick (AC) automaton structure is shared among several string matching modules in a time multiplexed access scheme. In [6], AC states with similar transitions are merged. The authors propose a mechanism to efficiently rectify the functional errors caused by the states merging. In doing so a reduction of 24% in the cost is achieved. Guinde and Ziavras [7] proposed a compression method for the string set where the required memory for storing the set is significantly reduced. In [11], they propose MIN-MAX algorithm for solving ambiguity and overlapped matching for Character Classes with Constraint Repetitions based Regular Expressions. A previous work was presented in [9] where the use of unique subsequences is introduced for reducing the hardware replication. In [8], a binary search tree state-of-the-art architecture is proposed, achieving the lowest memory cost per character but with a limited throughput.

3 Partitioning Methods

The present work is based on the partitioning methodology presented in [9] and then extended in [10]. Firstly, the initial signature set is partitioned into several sets denoted as u-sets. The partitioning criterion is that, every signature in a u-set must contain, at least, one unique substring. This is, a substring that is not contained in any other signature of that set. This substring is named unique substring, u-substring for short.

This partitioning allows that in a u-set, every signature can be mapped one-to-one with its corresponding u-substring. Therefore to find a u-substring in some data flow location, implies that its container signature, and no other, likely exist in that section of data stream, so there is no need to match any other signature. The likely-present signature is called candidate signature (CSig). This is fetched from memory every time its corresponding u-substring is detected. The section of data stream where the signature is expected to reside is named region of interest (RoI). A match occurs when CSig match character by character with the data stream, in a RoI. When extended to multi-character, it may happen that several u-substrings match in the same clock period. In order to avoid malfunctioning, a second partitioning is applied [10]. This is called security threshold partitioning. The output of this matching module is the signature ID, which is the signature address in the SMA, and a match enable output, signalling when a match occurs.

The first step in the construction of our architecture is to partition the signature set according to [9] and [10]. By using these methods, we guarantee only one possible signature match per clock cycle, for a u-set. The main contributions of this paper regarding ours previous works consist in: a) the use memory instead of logic, for storing signatures; b) the proposition of a character reconfiguration method which reduce, on average, the amount of memory bits required per character; and c) a different efficient masking solution, allowing to match non-uniform length signatures, using uniform hardware logic.

4 Architecture

Our method starts by representing every u-set as a matrix, with one character per cell and one signature per row. The signatures in this matrix are displaced, so that all u-substring first characters fall in the same column. In figure 2(a) there are three matrices. In the signature matrix, the top one, each row contains a signature where u-substrings are *"bb"*, *"lb"* and *"tl"*, respectively. The column where all substrings begin is called aligning-column, because it works as pivot for the Aligning. The dashed line in the figure2(a) marks the boundary between the head, i. e., the prefix up to Aligning Column, and the tail, which is the rest of the signature.

Since a signature matrix column, may contain repeated characters, the number of distinct characters in a column is lower, or equal, to the alphabet size. We build a second matrix called character matrix, the middle one in figure 2(a). This matrix collects only distinct characters in columns from the signature matrix. In real signature sets, the size of the character matrix columns tends to be lower than 256. This makes possible to reencode the characters in order to save memory. Let p be the number of characters in a character matrix column, the number of bits required to encode the signature characters is $log_2(p)$. This is what we have called character-recodification.

The bottom array of the figure 2(a) shows the bits that are required to store per column. Note that some columns have 0 bits, meaning that, we do not need to save this characters in memory. In these columns, the selected character is

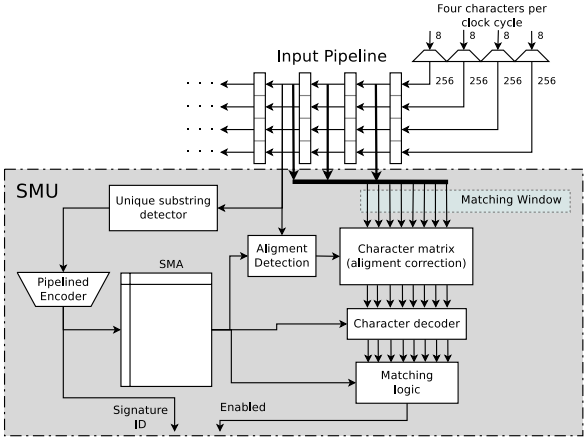

Fig. 1. General architecture view

always the same for any signature, so these characters lines can be hard-wired in the matching module, consuming no memory resources.

Figure 2(b) shows a histogram where bars represent the columns count with a specific width in bits. For a signature matrix with 1024 signatures extracted from the Snort rule database [12], there is a reduction of at least one bit in relation to the original character size. Note that 43 columns are six-bit wide, saving $43 * 2 = 86$ bits of memory. The matrix has 136 columns, without re-encoding $136 * 8 = 1088$ bits per row are required, while with re-encoding, this is reduced to 688, leading to a reduction of 36%. In terms of memory blocks, each of these contains 36 bits per entry, so $\lceil 1088/36 \rceil = 31$ are originally required, while with our method this is reduced to $\lceil 688/36 \rceil = 20$. This implies that the length of the memory entry can be shortened, making feasible the concatenation of embedded memory blocks, storing one signature per entry.

Each signature matrix is stored in an array of memories, called Signature Memory Array (SMA), occupying one entry per signature. The amount of entries of a MA is restricted to 1024. Therefore, the same u-set can require several SMAs. A Signature Matching Unit, SMU, is the basic component of our architecture, and its objective is to match the signatures contained in one MA. In figure 1, all but the input pipeline and the input character decoders, are components of the SMU. It performs five main tasks. First, match u-substrings from the data flow (Carried out by Unique Substring Detector). Second, fetch the Csig from the SMA corresponding to a matched u-substrings (Carried out by Unique Substring Detector and SMA). Third, align the Csig with the RoI (Carried out by SMA, Alignment Detection Component and Character Matrix Component). Four, execute the matching between the RoI and de Csig, comparing character by character (Carried out by Character Decoder component, Matching logic Component). Five, provide the match result, and the unique identifier of the recognized signature (Carried out by Matching Logic Component).

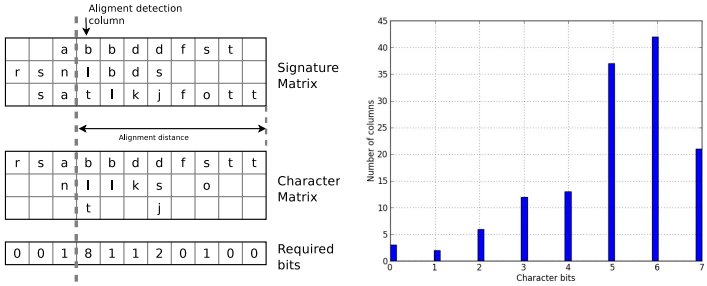

Fig. 2. (a)Signature Matrix example (b) Counting of columns at every width

As shown in figure 1, the architecture processes four characters per clock cycle. These characters are decoded into bit lines, and passed through a pipeline of register, denoted as i-pipeline. Every step in the i-pipeline, is divided into four sections, corresponding to four characters, resulting in a total of $256 * 4$ bit lines per step. The i-pipeline can be seen as a serial to parallel buffer, where the parallel outputs feed the SMUs inputs. The input of the SMU is called Matching window (MW). In the MW, every column of the signature matrix is related to four consecutive sections since a RoI can present four different alignments regarding to CSig. Therefore, the MW width, in number of sections, is the same as the signature matrix width, multiplied by four. One of the principal tasks of the SMU is to align the RoI with CSig in the MW, this is the process that we have called Aligning.

The module in charge of addressing the candidate signature from the MA is the The Unique Substrings Detector component. In this module, brute force detection is performed to find out u-substrings in the data flow. Meaning that every u-substring is detected by four matchers, one for each possible shift of the signature. Since the u-substrings are of short length, the matchers consumes few resources. The alignment detection component function is to find out the current RoI alignment. This is carried out by finding the location of the u-substring first character in the MW. Recall that these characters are contained in the aligning column. Character Matrix component is consistent with the character matrix representation as depicted figure 2. In this, a four-to-one multiplexor per matrix cell is deployed. Once the alignment of the RoI is known, this is used to control the array of multiplexers that performs the alignment.

The Character Decode Component receives a RoI aligned with the CSig. In this component, the characters of the CSig are decoded and compared against those of the RoI. There is one multiplexor per column controlled by the current column value (CSig re-encoded character). A character match occurs when the selected input of the multiplexor is asserted, meaning that the re-encoded character and the character in the RoI are equal. The output of the Character

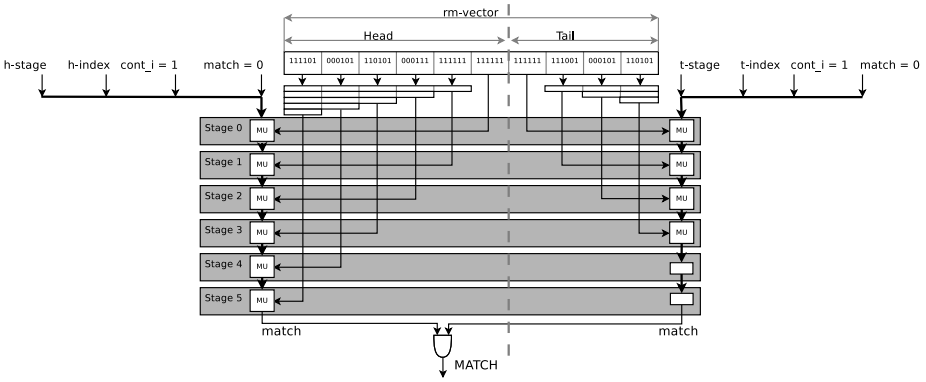

Fig. 3. Matching logic component

Decoder component is a bit vector named raw matching vector, rm-vector for short, and its length is equal to the signature matrix width. Each bit in this vector represents a match in the corresponding column. The central idea is to count the number of consecutive one's in the range of bits occupied by the candidate signature in the rm-vector, and signal a match result, when this number is equal to the signature size. This is performed by the Matching Logic Component presented in figure 3. The typical way of performing this operation is by saving a bit mask per signature with all ones out of the signature range and all zeroes in the signature range, then perform a typical masking operation when needed.

Reconsider the example with the signature matrix width of 136 columns, an equal number of bits would be needed to store at every memory entry, leading to an increment of $\lceil 136/36 = 4 \rceil$ additional memory blocks. We propose a different approach in the architecture presented in figure 3. In this, the number of mask bits required for the same example is reduced to only seventeen bits. The rm-vector is split into slices. Each slice takes six consecutive bits from the rm-vector. In the first step of the pipeline, the inner most slice of each section i.e. the closest to the dashed line, are processed. It continues with the next slice, and so on, until the outer most slice. The slice processing at each pipeline step is carried out by the Matching Units (MU) located at both sides of the pipeline.

The MUs also conform a pipeline of four signals, these are: stage, index, continuity, and match. We propose a bit mask composed by two pairs of stage and index values, one for the head section, and one for the tail section, these are h-stage, h-index, t-stage and t-index, respectively. These are stored in the entry together with the signature. Stage represents the outer most slice of the rm-vector occupied by signature, while index is a bit mask with all ones in the bits allowed by the signature in that slice. For example, a signature with rm-vector depicted in fig 3, whose signature has 15 characters in the head and 9 characters in the tail, his corresponding stage and index values are: *h-stage = 2*, *h-index = "000111"*, *t-stage = 1* and *t-index = "111000"*. The number of bits required for the h-stage and t-stage together is equal to the rm-vector width divided by six,

Table 1. Signature Matcher implementation

Virtex5FX100T implementation								
Signatures	**Characters**	**SMUs**	**Frec.**	**Thput.**	**BRAMs**	**LUTs**	**LUTs/char**	**Bit/char**
3,739	112,431	7	150MHz	4.8Gbps	60	35,508	0.32	20

Table 2. Comparisons with previous works

Comparison with previous works						
Arch.	**device**	**Input width**	**chars.**	**LE/chars**	**bit/char**	**Thput. Gbps**
Our approach	VirtexFX100T	32	112,431	0.32	20	4.8
Baker and Prasanna [3]	Virtex2P100	32	19,508	0.65	0	7.3
Hwang et. al. [4]	StratixERS140	32	3,028	1.5	0	11.6
Serrano et. al. [9]	VirtexFX100T	32	5,024	1.62	0	5.69
Kennedy et. al. [5]	Stratix	16	109,467	0.63	61	7.4
Prasanna and Le. [8]	VirtexFX200T	16	217,680	n/a	11	3.2
Lin and Chang. [6]	n/a	8	36,359	n/a	32	4
Guinde and Ziavras. [7]	virtex2P70	8	105,763	0.052	17.7	2.4

and the indexes sum twelve bits. For the example exposed before, the overall bits required are seventeen, compared with the original 139 bits required, this means a reduction of 86.3%.

5 Experiments and Results

Table 1 shows the results of the architecture implementation for a signature set of 3,739 signatures from Snort database [12]. For the Virtex-5 FX100T device containing 64,000 logic elements (LE) and 200 embedded memory blocks, the overall architecture occupies 60% of the resources. Table 2 shows the comparison against previous works. Our architecture presents the best logic cost regarding to others 32-bit-width architectures. The best memory cost is presented by Prasanna and Le [8]. However their throughput of 3.2 Gbps is achieved by using the double port memory feature of embedded memory blocks. By applying the same strategy, our architecture would double the throughput to 9.6 Gbps while maintaining the same memory cost. The largest Virtex5 device has 207,360 LE, the same architecture can be replicated up to 5 times in this device, achieving an aggregated throughput of 24 Gbps. Likewise, we estimate a character capacity of more than 500K characters.

6 Conclusions

We have presented a multi-character architecture which exploits intensively both, memory and logic resources. The replication of hardware is significantly reduced,

which leads to a better use of resources, lowering the cost per character compared to others multi-character architectures. Our character re-codification method allows storing one signature in a memory entry. Therefore we can compare the entire signature in one clock cycle. In addition, we have presented a uniform architecture capable of matching non-uniform signatures. If the double port access feature of embedded memory blocks is used the throughput can be doubled, taking into account the capacity of larger FPGA devices, a similar implementation as the one presented here can be replicated up to five times on a Virtex5-330T device.

References

1. Endorf, C., Schultz, E., Mellander, J.: Intrusion detection and prevention. Mc-Graw-Hill (2004)
2. Ghorbani, A., Lu, W., Tavallaee, M.: Network intrusion detection and prevention: concepts and techniques, vol. 47. Springer (2010)
3. Baker, Z.K., Prasanna, V.K.: Automatic synthesis of efficient intrusion detection systems on fpgas. IEEE Trans. Dependable Secur. Comput. 3(4), 289–300 (2006)
4. Hwang, W.J., Ou, C.M., Shih, Y.-N., Lo, C.T.D.: High throughput and low area cost fpga-based signature match circuit for network intrusion detection. Journal of the Chinese Institute of Engineers 32(3), 397–405 (2009)
5. Kennedy, A., Wang, X., Liu, Z., Liu, B.: Ultra-high throughput string matching for deep packet inspection. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2010, pp. 399–404 (2010)
6. Lin, C.-H., Chang, S.-C.: Efficient pattern matching algorithm for memory architecture. IEEE Trans. Very Large Scale Integr. Syst. 19(1), 33–41 (2011)
7. Guinde, N.B., Ziavras, S.G.: Efficient hardware support for pattern matching in network intrusion detection. Computers & Security 29(7), 756–769 (2010)
8. Prasanna, V.K., Le, H.: A Memory-Efficient and Modular Approach for Large-Scale String Pattern Matching. IEEE Transactions on Computers 62(5), 844–857 (2013)
9. Serrano, J.M.B., Palancar, J.H.: String alignment pre-detection using unique subsequences for FPGA-based network intrusion detection. Computer Communications 35(6), 720–728 (2012)
10. Serrano, J.M.B., Palancar, J.H., Cumplido, R.: Multi-character cost-effective and high throughput architecture for content scanning. In: Microprocessors and Microsystems (in press, 2013) (accepted manuscript), available online August 22: `http://authors.elsevier.com/sd/article/S0141933113000999`
11. Wang, H., Pu, S., Knezek, G., Liu, J.-C.: MIN-MAX: A Counter-Based Algorithm for Regular Expression Matching. IEEE Transactions on Parallel and Distributed Systems 24(1), 92–103 (2013)
12. Snort, `http://www.snort.org`