DESIGN OF AN ARCHITECTURE TO DETECT TOKEN-BASED CLAMAV VIRUS SIGNATURES

[1] **TATA RAGHUVARAN** (M.tech), [2] **K.VEERANNA BABU (**M.tech), assistant. professor

[1] Email-id: raghuvaran.tata727@gmail.com **,** Eluru College of Engineering and technology,Duggirala, west godavari

[2] Email-id:veerannababu.kottu@gmail.com, Eluru College of Engineering and technology,Duggirala, west godavari

Abstract: **We aim to implement a single-chip hardware detection engine for virus scanning. Our study is based on the ClamAV virus database, which contains 88.9K strings and 9.6K extended hex-signatures with restricted regular expression (regex) features. We have previously presented cost-effective hardware architectures to detect the 88.9K strings and 3.2K regex patterns that are composed of multiple string segments. In this paper, we shall present hardware architectures to detect the remaining 6.4K regex patterns. Our method is based on the information reduction approach. We transform the byte-oriented matching problem to a token-based matching problem. A regex pattern contains one or more segments, and a segment may be subdivided into multiple non-trivial tokens. In general, a token is associated with one or a few regexes only. The input byte stream is converted into a token-stream using dedicated hardware units, where the number of tokens is much less than the number of bytes. The token-stream is processed by a NFA-based aggregation unit to determine if any segment can be found. Detected segments are further processed by a scoreboard to determine if any multi-segment pattern can be found.**

Index Terms—string matching, regular expression matching, virus detection, hardware architecture.

I. INTRODUCTION

Every system connected to the Internet is susceptible to different kinds of attack such as virus and worm inflections, junk mail (spam messages) and email spoofing. Therefore there exists an increasing demand for network devices capable of inspecting the content of data packets in order to enhance network security and provide application-specific services. Firewalls were used considerably to prevent access to systems from intrusions but they cannot get rid of all security threats, nor can they identify attacks when they occur. Hence, next generation firewalls should provide deep packet inspection [3] capabilities, in order to provide prevention from these attacks. Network Intrusion Detection Systems (NIDS) performs the function of deep packet inspection. Matching engines inspect packet's payload searching for patterns that would alert security threats. Matching every incoming byte against thousands of pattern characters at wire rates such as for 3G, 4G high mobility communications is a complicated task.

Signature-based virus detection is a computation intensive task, in particular for the detection of virus patterns that contain regular expression (regex) features. ClamAV is an open-source anti-virus software for gateway email scanning. There are two types of rules in the ClamAV virus database, namely the MD5 file checksums and embedded virus signatures.

The number of non-wildcard bytes in the set of 6K regexes is about 450K. Modeling the matching problem using deterministic finite automaton (DFA) is out of question because exponential state explosion is inevitable. Conceptually if the conventional byte-oriented matching approach is applied, the underlying NFA will have more than a million states. To process one input byte the NFA needs to look up the transition rule table for several thousand times. As a result, the system complexity is high while the processing speed is unacceptably slow. Our proposed method is based on the information reduction approach. Instead of using the conventional byte-oriented matching, we transform the regexes into equivalent token-based patterns. A token is a non-trivial sub-pattern (which may contain regex features) extracted from the underlying patterns. The tokens can be seen as a hypothetical alphabet set with very large cardinality, e.g. the number of unique tokens is greater than the number of regexes in the pattern set. The input byte-stream is transformed into a token-stream using cost-effective hardware modules, where the number of tokens in the token-stream is much smaller than the number of bytes in the original input. When a token is received, the NFA-based detection

system will only need to make a few state transitions. As a result, the complexity of the underlying automaton can be reduced substantially, and cost-effective implementation in hardware is viable. We shall show that our method to detect the 6K regexes only requires about 1.84 MB of on-chip memory. Together with our previous results, it is possible to implement a hardware detection engine for the full set of ClamAV embedded virus signatures on a single FPGA. To the best of our knowledge, this is the first viable design of a hardware virus detection engine in one FPGA.

II. RELATED WORK

Implementation of NIDS signatures as deterministic finite-state automata (DFAs) may leads to state explosion problem. Although the overall memory has been reduced by Yu and Chen with the proposal of multiple DFAs, state-space explosion arises for complex signature sets. Another limitation is the increased scanning time because multiple DFAs should inspect payloads now. State explosion problems has not been resolved completely even after the implementations of DFA compression techniques based on transition rule reduction and data encoding. For the HFA method, hardware implementation details have not been revealed clearly. Bloom filters techniques proposed by Ho and Lemieux pose upper bound limit on the length of string segments to be handled. The major limitation of hardwired circuit approach to exploit parallelism is that they require FPGA to be reconfigured for every database update. State explosion problem has not been addressed in the approach presented by Lee to detect virus signatures with regex features based on generalization of Aho-Corasick string matching algorithm. In this paper we present a memory-efficient parallel string matching scheme using the pattern dividing approach and its hardware architecture for identification of patterns. Initially the byte stream is transformed into token stream by dividing the long target patterns into sub-patterns with a fixed length and then processed with bit-based comparisons. In this approach, there is an increased number of shared common states due to reduced length and this approach is very efficient when compared with the cases of the string matching with byte-based comparisons. Here, we present the results of our implementation with open-source IDS software ClamA. ClamAV virus database consists of basic, regular expression (or regex) and MD5 types. A basic signature is represented as a continuous byte string and a regex signature is an extension of the basic pattern with various wildcards. The amount of processing speed achieved with the optimizations of string matching algorithm and the improvement in throughput has been analyzed. The methodology that we developed here is efficient in terms of throughput and worst-case performance. The proposed method speeds up the processing rate of the string searching by three times.

III. PROPOSED DETECTION METHOD.

The set of 6K regexes contains 1866 instances of * and 16K counting blocks, and the count values can be very large. The large number of counting blocks poses great challenge to the design of the hardware detection engine.

We use different approaches to handle the 5 types of displacement counts. The arbitrary displacement * is equivalent to the at-least count {0-}. We use the * and atleast count {n-} as delimiter to divide a pattern into segments. Segments that are not pure string may be subdivided into 2 or more tokens. String tokens can be detected using our P-AC [7] and QSV [3] string detection methods. An extended P-AC detection method called PACX is used to detect fixed-length tokens with 4 to 15 bytes that contain a string component at the front plus a small number of wildcard bytes, nibbles and/or alternate bytes. MXNFA [8] is a more general regex detection method, and it is used to detect more complex tokens that may contain counting block(s) and other regex features.

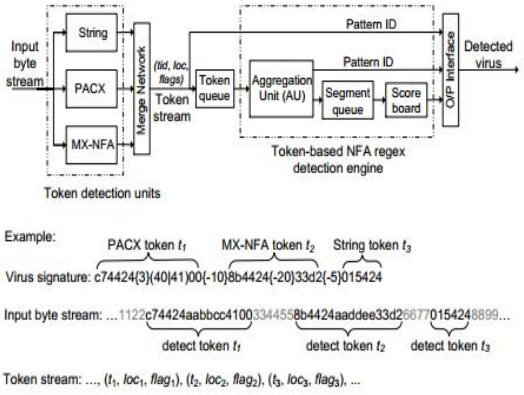


Figure 1. Block diagram of the virus detection engine.

The overall organization of the virus detection engine is depicted in Fig. 1. Hardware modules are built to detect the three types of tokens, namely string tokens, PACX tokens and MX-NFA tokens. The input byte stream is converted into a token stream, where the number of tokens is much less than the number of bytes. An example based on virus Worm.Allaple-11 is shown in Fig. 1 to illustrate idea. The AU is responsible for reassembling the tokens to form segments. If the detected segment is a pattern, then its ID is sent to the output interface directly. If the detected segment is part of a multi segment pattern, it is inserted into the segment queue for further processing by the Scoreboard.

Division of segment into tokens

If a segment is a string, then it is not subdivided. Short segments (e.g. 15 bytes or less) may be detected directly by the PACX pipeline or MX-NFA unit directly. If a long segment contains displacement counts, alternate bytes and nibbles, then the segment may be subdivided into 2 or more non-trivial tokens. One should note that tokens need not be disjoint. First, we shall use the displacement counts as delimiters to divide the segments into multiple components. If the component is a string with at least 3 distinct byte values, then it may form a string token. If the component is a single-byte or double-byte, then it is merged with the adjacent component or parts of the adjacent component together with the displacement count to form a more distinctive token. Detection of token with large exact-count value or range count with large lower-bound is expensive. Hence, the preprocessing module will try to avoid having these 2 types of count included in the token definition. If a token contains a range-count {n-m} and it is not the first token of the segment, the range-count can be replaced by an at most count {-m}. The lower-bound displacement can be verified by the AU.

PACX token detection unit:

The PACX method is an extension of our P-AC string detection method. The length of the token that can be detected by a PACX pipeline is bounded by the length of the pipeline. Second, wildcard byte, nibble, and alternate byte values can be supported by the PACX pipeline.

The structure of the PACX pipeline for processing tokens with 4 to 7 bytes is depicted in Fig. 2. Each pipeline stage is consisted of a lookup table and a processing unit. An entry in the lookup table may contain up to 6 fields, the character (char), a character mask (cm), the next state (ns), a bit-select mask (bs), the token ID (tid) and 4 control flags (isPattern, isSegment, isFirst, isSecond). The bits isPattern and isSegment indicate if the token is a pattern and/or a segment by itself. The bits isFirst and isSecond are used by the AU. Table LT0 only contains the ns and bs fields, tables LT1 to LT2 do not have the tid and control flags fields, and table LT6 does not have the ns and bs fields. LT0 is directly indexed by the input byte, and no comparator circuit is required. For the other stages, the address for accessing the local table is obtained by the direct indexing plus bit-select (DIBS) approach [7]. The selected bits extracted from the input byte form the offset which is added to the current state value to obtain the address for accessing the lookup table. If bs = 00 (i.e. no bit is selected), then the offset value is zero. If the input byte matches the (char, cm) pair retrieved from the local table, then the associated ns, bs and tid (if applicable) will be passed to the next pipeline stage or output interface; otherwise a null (zero) value is passed to the next stage or output interface. The output interface is made up of simple ORgates that combine the outputs from the 4 output stages to one final value. Tokens that are detected by the given PACX pipeline should be mutual exclusive, i.e. there will not be multiple matches in the same cycle.

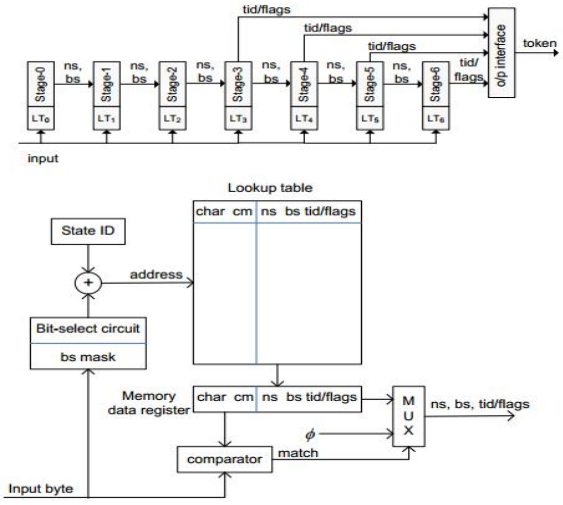


Figure 2. Block diagram for the PACX pipeline for tokens with 4 to 7 bytes, and internal structure of a pipeline stage.

The character mask cm is used to support the matching of wildcard, nibble and alternate bytes. Let c7…c0 represent the stored character char, and m7…m0 represent the cm, and d7…d0 represent the input byte. The output of the comparator is defined by the Boolean equation match=$\overline{\prod_{i=0}^{7}(mi + \overline{ci \oplus di})}$. To match a wildcard byte ??, cm is set to 00. To match the nibble e?, cm is set to F0.

We shall illustrate the organization of the PACX pipeline using an example. Consider the set of tokens shown in Fig. 3. The construction of the lookup tables for the pipeline stages is based on the character-trie of the given set of tokens. The number next to an edge is the transition symbol, and the number inside a state is the base address of the given node in the corresponding lookup table. Setup of the lookup tables of the PACX pipeline for the sample set of tokens is shown in Fig. 7. The table entries corresponding to tokens t1 and t5 are highlighted.

Let's consider the branch point at node A in the character-trie for t1 to t3. The transition symbol for the upper branch A → B is 32, and the transition symbol for the lower branch A → C. We use the bit-select mask bs to differentiate the two groups of transition symbols. A subset of bits is chosen such that the symbols of the two branches can be differentiated. To differentiate 32 , we need to choose the last 3 bits, i.e. bs is set to 07 in entry 01 of LT2. The last 3 bits of 32 are "010", and the last 3 bits of [33-39] are {"011", "100", "101", "110", "111", "000", "001"}. The selected bits extracted from the input byte form the offset value for accessing the table LT3. The transition edge for A → B is stored at address 03 of LT3, and the transition edge for A → C is made up of entries 01 to 02, and 04 to 08 of LT3. The transition edge for C → D is implemented by 2 table entries at addresses 05 and 06 in LT4. The set of 10 symbols can be divided into 2 subgroups and by setting bs to 08 (selecting the fourth bit from the right). For the subgroup, cm is set to F8 (the rightmost 3 bits are don't care). For the subgroup, cm is set to FE (the rightmost bit is don't care). An example to illustrate the operation of the PACX pipeline is given in the appendix.

Token	Value
t_1	2822253236(35|36|37|38|39)66
t_2	28222532(37|38|39)(30|31|32|33|34|35|36|37|38|39)66
t_3	282225(33|34|35|36|37|38|39)(30|31|32|33|34|35|36|37|38|39)(30|31|32|33|34|35|36|37|38|39)66
t_4	76388be?5?e9
t_5	83????6a0068

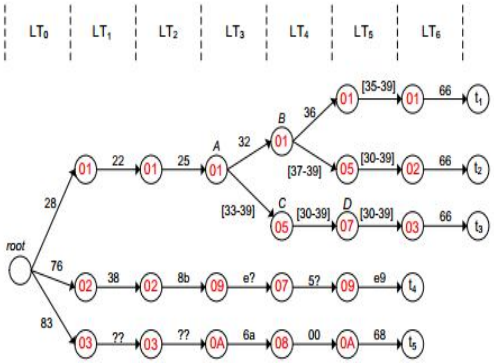


Figure 3. Sample set of tokens and the character-trie.

MX-NFA token detection unit:

The MX-NFA regex detection method was initially developed to support intrusion detection. In principle MXNFA can also be used to detect the full virus patterns but the cost can be very high for long patterns and patterns with large exact-count and range-count. In this study, the virus signatures are divided into tokens, and the MXNFA is used to process tokens that cannot be detected by other methods. The design of the MX-NFA detection unit is refined for virus detection.

In the Snort regex, the counting block corresponds to the repetition of a symbol by the given number of times. For example the Snort-style regex r1 = "number\d{8,16}" represents a prefix string "number" followed by 8 to 16 decimal digits. When the hardware sees the prefix string in the input, it starts the counting. During the counting process, the hardware needs to monitor if the input character fulfills the counting criterion, i.e. the input character is a digit. If the input character does not meet the counting requirement, the counting process is aborted. In this regex, the prefix string is not compatible with the counting criterion, i.e. if the input character is the letter 'n', the counting process will be aborted because the character is not a digit.

In virus detection, the counting block represents a displacement, and the input byte value is don't care. Hence, the hardware needs not monitor the value of the input byte during the counting process. The prefix string in front of the counting block is usually very short, e.g. a few bytes. As a result, we shall be facing the recurrent count problem for checking exact-count and range-count. Consider an example token with an exact-count 1122{n}3344. When the hardware sees the 2-byte prefix string 1122, it starts a counter to count for n cycles. However, during the counting process if the prefix string 1122 is seen again, the hardware needs to start another counter. We call this phenomenon recurrent-counting. If recurrent-counting is possible, the exact-count cannot be checked by one (or some fixed number of) hardware counter. Fortunately, atmost count is the most popular type of variable displacement in virus signatures, and it can be detected by a single hardware counter. Consider an example token with an at-most count 5566{-n}7788. When the hardware sees the prefix string 5566, it starts a counter to count-down from n+1 (value of the at-most count plus 1). The hardware for the detection of the suffix string 7788 is enabled while the value of the counter is non-zero. If during the counting process the prefix string is seen again, then the counter is simply reloaded with the initial count value.

The MX-NFA method is based on NFA. The transition rules of the underlying NFA are stored in an active rule table, where each rule can be enabled or disabled dynamically by the embedded hardware circuits. The block diagram of a MX-NFA unit with a count module is shown in Fig. 8. The 8-bit input symbol is used as the address to retrieve a 36-bit word from the memory array. The size of the memory array 256×36 is chosen due to the characteristics of the FPGA block RAM. Multiple match units can be cascaded to detect regexes requiring more than 36 transition rules. The i-th bit of the memory word represents whether the input symbol matches the transition symbol of the i-th entry (transition rule). A column of 256 bits in the memory array is used to encode an arbitrary character subclass for a transition rule. To encode a wildcard byte ??, all the 256 bits in the column are set to 1. To encode a null entry, all the 256 bits in the column are set to 0. The behavior of a transition rule is defined by the associated control flags listed in Fig. 9. The E-bit of a transition rule changes dynamically, while the other control flags are static. The forward activation signal aj of the i-th entry is connected to the input pj of the (i+j)-th entry.

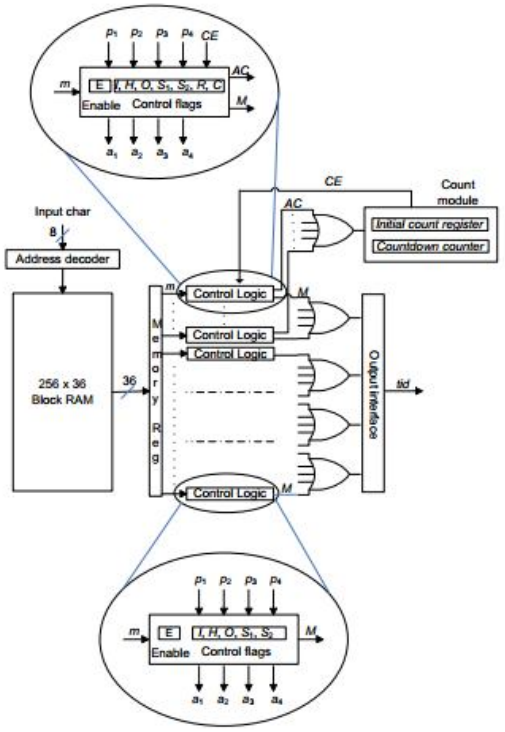


Figure 4. Block diagram of the MX-NFA match unit with a count-module.

The count module is used to support the checking of at-most count. It contains an initial count register and a count-down counter. The 36 transition rules are divided into 4 quadrants. The rules in the top quadrant can interact with the count module, and these entries have 2 additional control flags R and C. When an activate count signal AC is sent to the count module, the initial count value is loaded to the counter, and the counter starts the count down. When the counter counts down to zero, i.e. makes a 1 to 0 transition, a count-event signal CE is generated and the counter stops counting. For exact count, the counting block will be unrolled. A range-count {n-m} is equivalent to an exact-count plus an at-most count, i.e. {n}{-(m-n)}. The exact-count is unrolled to occupy n match entries in the MX-NFA unit, and the at-most count is checked by a count module.

The setting of the E-bit, and generation of the forward activation signals a1 to a4, and match signal M are defined by the following Boolean equations. Let Et denote the value of the E-bit in

cycle t. The symbol Reset represents the system reset signal.

$$E^{t+1} = I \cdot Reset + \overline{Reset} \cdot (E^t \cdot H + p_1^t + p_2^t + p_3^t + p_4^t)$$
$$a_1^t = E^t \cdot m^t$$
$$a_2^t = E^t \cdot m^t \cdot (S_2 + S_1)$$
$$a_3^t = E^t \cdot m^t \cdot S_2$$
$$a_4^t = E^t \cdot m^t \cdot S_2 \cdot S_1$$
$$M^t = E^t \cdot m^t \cdot O$$

For the transition rules associated with the count module, the Boolean equation for the setting of the E-bit and the generation of the signal AC are shown below.

$$E^{t+1} = I \cdot Reset + \overline{Reset} \cdot (E^t \cdot H \cdot (R \cdot CE^t) + p_1^t + p_2^t + p_3^t + p_4^t)$$
$$AC^t = E^t \cdot m^t \cdot C$$

	Control flag	Description
All match entries	Enable (*E*)	The rule is active if *E*=1. By default the *E* bit is reset automatically at the end of the clock cycle.
	Initial (*I*)	If *I*=1, the rule is active after initialization (system reset).
	Hold (*H*)	If *H*=1, the *E* bit will not be reset automatically.
	Sequential activation (S_2S_1)	The two control bits specify the number of sequential rules to be activated if the rule is fired. Let the current rule be rule *i*. S_2S_1=00 : activate rule *i*+1 S_2S_1=01 : activate rules *i*+1 to *i*+2 S_2S_1=10 : activate rules *i*+1 to *i*+3 S_2S_1=11 : activate rules *i*+1 to *i*+4
	Output (*O*)	If *O*=1, an output signal is generated when the rule is fired.
Match entries associated with count module	Respond (*R*)	If *R*=1, responds to count-event signal *CE* and clear *E*-bit.
	Activate Counter (*C*)	If *C*=1, sends an activate count signal *AC* to the count module when the rule is fired.

Figure 5. Control flags of a MX-NFA match entry.

Forward activation of multiple rules can be used to support at-most count of 1 to 3. Consider an example token equals to 1122{-2}3344{-8}5566. The setup of the MXNFA rule table is depicted in Fig. 10. The at-most count {- 2} is handled by introducing 2 wildcard entries (e3 and e4), and the forward activation control of e2 is set to activate the next 3 rules e3 to e5 when it is fired. The at-most count {-8} is checked by the counter module. The bottom part of the figure shows the changes of the E-bit of the rules for processing the input 11228833449988556677.

After system reset, E-bit of e1 is set to 1, and it will remain active with H=1. When e1 is fired in cycle 1, e2 is activated in cycle 2. When e2 is fired in cycle 2, it activates e3 to e5 in cycle 3. The wildcard symbol of e3 and e4 is representing by having all the 256 bits in the given column set to 1. Both e3 and e4 fire in cycle 3, and as a result e4 and e5 remain active in cycle 4. When e6 fires in cycle 5, it activates e7 and also starts the counter. The H-bit of e7 is equal to 1, hence e7 remains active until its E-bit is cleared by the count-event expected to be generated by the counter in cycle 14. While e7 remains active, it will continue to check for the first byte of the suffix string. Entry e7 is fired in cycle 8 when the input byte matches the symbol of e7. When e8 is fired in cycle 9, the local_tid of the token is output. Rule e9 with a null symbol is used as a delimiter, i.e. all the 256 bits in the given column of the memory array are set to 0 and the rule will never fire.

MX-NFA rule table							
		Control flags					
Rule	Symbol	*I*	*H*	*O*	S_2S_1	*R*	C
e_1	11	1	1	0	00	0	0
e_2	22	0	0	0	10	0	0
e_3	??	0	0	0	00	0	0
e_4	??	0	0	0	00	0	0
e_5	33	0	0	0	00	0	0
e_6	44	0	0	0	00	0	1
e_7	55	0	1	0	00	1	0
e_8	66	0	0	1	00	0	0
e_9	null	0	0	0	00	0	0

	Changes of the *E*-bit when process the inputs "11228833449988556677"										
	t=0	*t*=1	*t*=2	*t*=3	*t*=4	*t*=5	*t*=6	*t*=7	*t*=8	*t*=9	*t*=10
Rule	reset	11	22	88	33	44	99	88	55	66	77
e_1	1	1^F	1	1	1	1	1	1	1	1	1
e_2	0	0	1^F	0	0	0	0	0	0	0	0
e_3	0	0	0	1^F	0	0	0	0	0	0	0
e_4	0	0	0	1^F	1^F	0	0	0	0	0	0
e_5	0	0	0	1	1^F	1	0	0	0	0	0
e_6	0	0	0	0	0	1^F	0	0	0	0	0
e_7	0	0	0	0	0	0	1	1	1^F	1	1
e_8	0	0	0	0	0	0	0	0	0	1^M	0
e_9	0	0	0	0	0	0	0	0	0	0	1

Figure 6. Setup of the MX-NFA rule table for the token 1122{-2}3344{-8}5566, and the changes of the E-bit when processing the input. The superscript F represents the rule is fired, and the superscript M represents a match output is generated.

A 9-bit register is associated with each MX-NFA match unit. The output interface produces an 11-bit local_tid value. The leftmost 9 bits of the local_tid are obtained from the register, and the rightmost 2 bits are generated by the interface circuit. The rightmost 2 bits represent the quadrant number of the matching entry. A valid local_tid should be non-zero. If the output local_tid is zero, then it means no token is found. When multiple tokens are mapped to the MX-NFA, it is required that a quadrant cannot

contain output entry of more than 1 token. Also, the tokens mapped to the 36-entry MX-NFA unit cannot produce concurrent match outputs. The local_tid value is later converted to the actual token ID together with the associated control flags via a token descriptor table.

Token refinements, ID assignment and design of the AU:

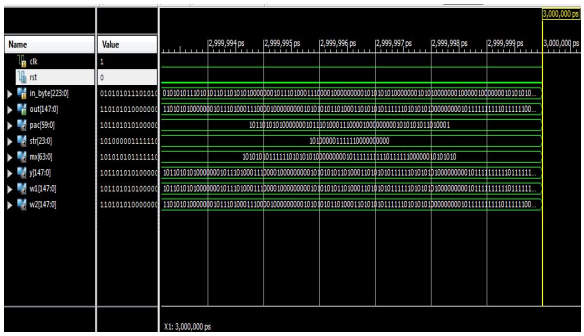
Out of the 6059 regex signatures 5250 patterns contain only 1 segment and 809 patterns are composed of 2 or more segments. The total number of segments is 7925, and the number of distinct segments is 7568. A segment may be shared by multiple patterns. This usually happens when the patterns concerned are variants of the same virus. Out of the 7568 distinct segments, 829 segments are not subdivided, 6739 segments are divided into 2 or more tokens. The AU is used to reassemble the detected tokens to determine if any multi-token segment can be found. We shall first explain the general principle and design of the AU, and then discuss the token set refinement and token ID assignment policy.

The AU is implemented as a NFA with conventional transition rule tables and embedded processing logic. A state transition graph is constructed for the set of subdivided segments. Four-bit tid is used in the example. The chosen example segments will also help to explain the necessary refinements of the token definitions and token ID assignment policy. The transition edges from the root node to the level-1 nodes are labeled with the tid only. The first token of a segment can appear anywhere in the input, and it has no location constraint. The transition edges from other nodes contain 4 fields (tid/wc, minD, maxD), where wc represents the number of wildcard bits on the right hand side of the tid field (bits that are ignored in the comparison), minD and maxD represent the displacement constraint. The NFA will make a transition from the current state to the next state if the input token matches the tid/wc and the displacement constraint specified by minD and maxD.

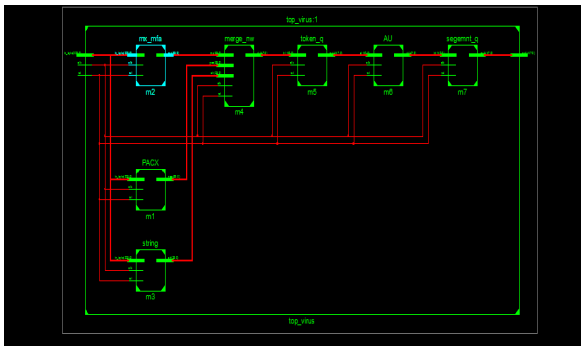
The second token of segment Sa can have variable length. Hence, the minimum and maximum displacement of the second token of Sa from the first token are 22 and 29 bytes, respectively. The value of minD and maxD for the transition edge from node 1 to the output node for Sa are set to 22 and 29, respectively. For segment Sb, the rangecount {100-250} of the second token is replaced by an atmost count {-250} to reduce the cost of the hardware detection unit. The minD value of the transition edge from node 2 to the output node for Sb is set to 102 based on the lower bound of the range-count in the original token. Hence, the verification of the lower-bound displacement is done by the AU, instead of by the token detection unit.

IV. RESULTS

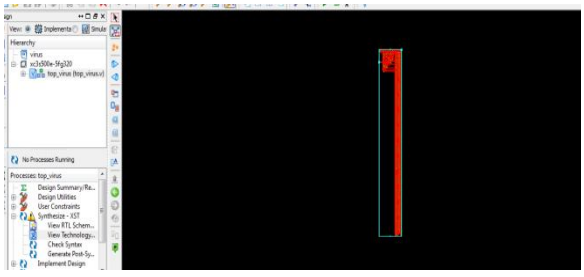
Simulation .



RTL SCHEMATIC:



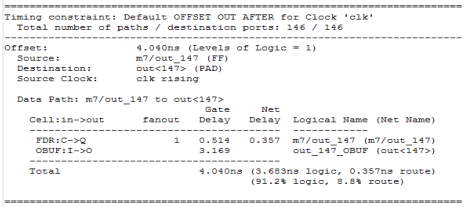
Technology Schematic.



Design Summary.

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slices	187	4656	4%
Number of Slice Flip Flops	296	9312	3%
Number of 4 input LUTs	327	9312	3%
Number of bonded IOBs	293	232	126%
Number of GCLKs	1	24	4%

Timing Diagram.



```
Timing constraint: Default OFFSET OUT AFTER for Clock 'clk'
  Total number of paths / destination ports: 146 / 146
-------------------------------------------------------------------------
Offset:              4.040ns (Levels of Logic = 1)
  Source:            m7/out_147 (FF)
  Destination:       out<147> (PAD)
  Source Clock:      clk rising

  Data Path: m7/out_147 to out<147>
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDR:C->Q             1    0.514   0.357  m7/out_147 (m7/out_147)
     OBUF:I->O                 3.169          out_147_OBUF (out<147>)
    ----------------------------------------
    Total                      4.040ns (3.683ns logic, 0.357ns route)
                               (91.2% logic, 8.8% route)
```

V. CONCLUSION

In this paper, we have presented memory-based hardware architectures to detect 6K ClamAV virus signatures with restricted regex features. Our method is based on the information reduction approach by transforming the byte-oriented matching problem to a token-based matching problem. Three types of tokens, namely string token, PACX token and MX-NFA token, are defined. The detected tokens are then processed by the aggregation unit (AU), which is implemented based on conventional NFA approach. Outputs of the AU may be further processed by the scoreboard to determine if any multi-segment pattern can be found. Hence, it is possible to put together our designs of the detection engines for the 88.9K strings and 9.6K regexes in one FPGA or ASIC. How to obtain a better solution to this problem is left to the future work. Another direction for future research is regarding the improvement of the processing speed.

REFERENCES

[1] ClamAV anti-virus system, http://www.clamav.net

[2] PCRE – Perl Compatible Regular Expressions, http://perldoc.perl.org/perlre.html

[3] D. Pao, X. Wang, X. Wang, C. Cao and Y. Zhu, “String searching engine for virus scanning”, IEEE Trans. Comput., Vol. 60, pp. 1596–1609, 2011.

[4] D. Pao, X. Wang, “Multi-stride string searching for high-speed content inspection”, The Comput. J., Vol. 55, pp. 1216-1231, 2012.

[5] X. Wang, N.L. Or, Z. Lu and D. Pao, “Hardware accelerator to detect multi-segment virus patterns”, The Comput. J., 2015 (accepted for publication), doi:10.1093/comjnl/bxu079

[6] D. Pao, W. Lin, B. Liu, “Pipelined architecture for multi-string matching”, IEEE Computer Arch. Letters, Vol. 7, pp. 33-36, 2008.

[7] D. Pao, W. Lin, B. Liu, “A memory efficient pipelined implementation of the Aho–Corasick string matching algorithm”, ACM Trans. Archit. Code Optim., Vol. 7, Article 10, 2010.

[8] D. Pao, N.L. Or, R.C.C. Cheung, “A memory-based NFA regular expression match engine for signature-based intrusion detection”, Computer Communications, Vol. 36, pp. 1255-1267, 2013.

[9] D. Pao, X. Wang and Z. Lu, “Design of a near-minimal dynamic perfect hash function on embedded device”, IEEE ICACT, pp.457-462, 2013.

[10] D. Pao, Y.K. Li and P. Zhou, “Efficient packet classification using TCAMs”, Comput. Netw., Vol. 50, pp. 3523–3535, 2006.

[11] Snort intrusion detection system, http://www.snort.org

[12] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley and J. Turner, “Algorithms to accelerate multiple regular expressions matching for deep packet inspection”, ACM SIGCOMM Computer Communication Review, Vol. 36, pp. 339–350, 2006.

[13] J. van Lunteren, “High-performance pattern-matching for intrusion detection”. IEEE INFOCOM, pp.1–13, 2006.

[14] T. Liu, Y. Yang, Y. Liu, Y. Sun and L. Guo, “An efficient regular expressions compression algorithm from a new perspective”, IEEE INFOCOM, pp. 2129-2137, 2011.

[15] J. Patel, A. X. Liu, E. Torng, “Bypassing space explosion in highspeed regular expression matching”, IEEE/ACM Trans. Networking, pp. 1701-1714, 2014.

[16] K. Wang, Z. Fu, X. Hu and J. Li, “Practical regular expression matching free of scalability and performance barriers”, Computer Communications, Vol. 54, 97-119, 2014.

[17] N. Tuck, T. Sherwood, B. Calder, G. Varghese, “Deterministic memory-efficient string matching algorithms for intrusion detection”, IEEE INFOCOM, pp. 2628-2639, 2004.

[18] C.R. Meiners, J. Patel, E. Norige, A.X. Liu and E. Torng, “Fast regular expression matching using small TCAM”, IEEE/ACM Trans. Networking, Vol. 22, pp. 94-109, 2014.

[19] L. Tan, B. Brotherton, T. Sherwood, “Bit-split string-matching engines for intrusion detection and prevention”, ACM Trans. Archit. Code Optim., Vol. 3, pp. 3-34, 2006.

[20] R. Dixon, O. Egecioglu, T. Sherwood, "Automata-theoretic analysis of bit-split languages for packet scanning", LNCS 5148, pp. 141-150, 2008.
[21] M. Becchi, Regular Expression Processor, http://regex.wustl.edu/index.php/Main_Page
[22] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection", ACM ANCS, pp. 93-102, 2006. [23] J. van Lunteren and A. Guanella, "Hardware-accelerated regular expression matching at multiple tens of Gb/s", IEEE INFOCOM, pp.1737–1745, 2012.
[24] S. Kumar, B. Chandrasekaran, J. Turner and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia", ACM/IEEE ANCS, pp.155–164, 2007.
[25] R. Smith, C. Estan and S. Jha, "XFA: Faster signature matching with extended automaton", IEEE Symposium on Security and Privacy, pp.187–201, 2008.
[26] R. Smith, C. Estan, S. Jha and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata", ACM SIGCOMM Computer Communication Review, Vol. 38, pp. 207–218, 2008.
[27] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection", ACM Conf. on Emerging Network Experiment and Technology (CoNEXT), pp. 1–12, 2007.
[28] Y.H.E. Yang, V.K. Prasanna, "Space-time tradeoff in regular expression matching with semi-deterministic finite automata", IEEE INFOCOM, pp.1853-1861, 2011.
[29] C. Liu, J. Wu, "Fast deep packet inspection with a dual finite automata", IEEE Trans. Comput., Vol. 62, pp. 310-321, 2013.
[30] M. Bando, N.S. Artan, H.J. Chao, "Scalable lookahead regular expression detection system for deep packet inspection", IEEE/ACM Trans. on Networking, Vol. 20, pp. 699–714, 2012.
[31] I. Sourdis, S. Vassiliadis, J. Bispo and J.M.P. Cardoso, "Regular expression matching in reconfigurable hardware", J. of Signal Processing Systems, Vol. 51, pp. 99 –121, 2008.
[32] I. Sourdis, D.N. Pnevmatikatos and S. Vassiliadis, "Scalable multigigabit pattern matching for packet inspection", IEEE Trans. VLSI Systems, Vol. 16, pp. 156-166, 2008.
[33] Y. H. E. Yang, V. K. Prasanna, "High-performance and compact architecture for regular expression matching on FPGA", IEEE Trans. Comput., Vol. 61, pp. 1013-1025, 2012.
[34] Y.-D. Lin, P.-C. Lin, Y.-C. Lai and T.-Y. Liu, "Hardwaresoftware codesign for high-speed signature-based virus scanning", IEEE Micro, Vol. 29, Issue 5, pp. 56-65, 2009.
[35] T. H. Lee, "Generalized Aho-Corasick algorithm for signature based anti-virus applications", IEEE Int. Conf. Comuter, Communications and Networks, pp. 792-797, 2007.
[36] H. Nakahara, T. Sasao, M. Matsuura and Y. Kawamura, "A virus scanning engine using a parallel finite-input memory machine and MPUs", IEEE Int. Conf. Field Programmable Logic and Applications, pp. 635-639, 2009.
[37] W. Zhang, Y. Xue, D. Wang, T. Song, "A multiple simple regular expression matching architecture and coprocessor for deep packet inspection", IEEE Asia-Pacific Computer System Architecture Conf., 2008.
[38] J. W. Lockwood, J. Moscola, D. Reddick, M. Kulig, T. Brooks, "Application of hardware accelerated extensible network nodes for Internet worm and virus protection", LNCS 2982, pp. 44-57, 2004.
[39] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors", Comm. of ACM, Vol. 13, no. 7, pp. 422-426, 1970.
[40] J.T.L. Ho and G.G.F. Lemieux, "PERG: A scalable FPGA-based pattern-matching engine with consolidated Bloomier filters", Int. Conf. on Field-Programmable Technology, pp.73–80, 2008.
[41] J.T.L. Ho and G.G.F. Lemieux, "PERG-Rx: A hardware pattern matching engine supporting limited regular expressions", ACM FPGA'09, pp. 257-260, 2009.
[42] T. N. Thinh, T. T. Hieu, H. Ishii, S. Tomiyama, "Memory efficient signature matching for ClamAV on FPGA", IEEE Int. Conf. Communications and Electronics, pp. 358-363, 2014.

Mr. T.RAGHUVARAN was born in Kandukuru, AP on December 6, 1991. He graduated from the Jawaharlal Nehru Technological University, Kakinada. His special fields of interest included VLSI DESIGN. Presently He is studying M.Tech in Eluru College of Engineering and Technology, Duggirala.

Mr. K.VEERANNABABU was born in Doddanapudi, AP, on December 3, 1990. He post graduated from the Jawaharlal Nehru Technological University, Kakinada. Presently He is working as an Asst Prof in Eluru College of Engineering and Technology, Duggirala. So far he is having 4 Years of Teaching Experience in various reputed engineering colleges. His special fields of interest included Microprocessors and microcontrollers, Embedded Systems, Digital Signal Processing & communication Systems.