FPGA Based Accelerator for Pattern Matching in YARA Framework

Shreyas G Singapura, Yi-Hua E. Yang,

Anand Panangadan, Tamas Nemeth and Viktor K. Prasanna

Computer Engineering Technical Report Number CENG-2015-05

Ming Hsieh Department of Electrical Engineering – Systems
University of Southern California
Los Angeles, California 90089-2562

September 2015

FPGA Based Accelerator for Pattern Matching in YARA Framework

Shreyas G Singapura*, Yi-Hua E. Yang†, Anand Panangadan*, Tamas Nemeth‡ and Viktor K. Prasanna*
*Ming-Hsieh Dept. of Electrical Engineering, University of Southern California
{singapur, anandvp and prasanna}@usc.edu
†yeyang@google.com, Google Inc., Mountain View, CA
‡Tamas.Nemeth@chevron.com, Chevron

***Abstract*—Pattern Matching is an integral part of intrusion detection systems to detect potential threats and is becoming a bottleneck due to the complexity and scale of patterns. YARA is a pattern matching framework which helps in the identification of malicious content by defining complex patterns and signatures. Software implementation on general purpose processors (CPU) do not meet the throughput requirements of core networks. In this paper, we present an FPGA based hardware accelerator to boost the performance of pattern matching in YARA framework. The proposed streaming architecture consists of pattern matching engines organized as two-dimensional stages and pipelines. Each YARA rule is logically mapped to a stage in the architecture resulting in a modular architecture. This permits easy update to the architecture as the rules change. Several optimizations such as multi-character matching and BRAM based character classification are employed to obtain a high performance. We implemented rulesets of sizes varying from 8 to 200 with the total number of patterns from 128 to 6000. Post place-and-route results demonstrate that the proposed design achieves throughput ranging from 12.85 Gbps to 21.8 Gbps. This is an improvement of 8.8× to 14.5× in comparison to the throughput of 1.45 Gbps for a software only implementation on a state of the art multi-core platform.**

***Keywords*-Pattern Matching; FPGA; BRAM; Intrusion Detection; Finite State Machine; NFA; YARA**

I. Introduction

Intrusion detection systems [1], [2] in applications such as network security, content filtering and data mining use pattern matching is a key kernel used by . The Internet of Things (IoT) has brought along with it an ever increasing bandwidth of network traffic and a new set of vulnerabilities. One such vulnerability is the large number of malware patterns and their complexity due to the large number of devices.

YARA [3] has emerged as a widely used tool [1], [2] which help users to create signatures and patterns specific to malware patterns and perform traffic analysis to detect these patterns. YARA is gaining popularity due to the wide variety of patterns that can be defined and its ease of usage by malware analysts. It is being used by various vendors [3], [4] and in research tools such as Cuckoo [5] and VirusTotal [6]. YARA offers significant capabilities to perform malware analysis and the focus has been on the software implementation on general purpose processor (CPU). CPU based implementations are not able to meet the demands of the internet for higher throughput and lower latency. In the future, the complexity of pattern matching activities such as deep packet inspection will further increase and it is very difficult for CPU based solutions to keep up with the requirements of the next generation network processing systems [7], [8].

In recent years, interest in hardware accelerators such as GPUs and FPGAs for real time network processing systems has increased [9]–[11]. This is due to their ability to perform application specific computations faster than software based implementation on general purpose processors. In this paper, we focus on FPGA as the hardware accelerator. State of the art FPGA devices [12] provide large amount of parallel logic to create deep pipelines, large amount of high bandwidth on-chip memory. Additionally, since FPGAs are programmable, they can be optimized to match the application requirements.

Our work focuses on implementation of pattern matching in the YARA framework on FPGAs and we develop optimizations for high performance architecture focusing on throughout as the performance metric. To the best of our knowledge, this is the first work to accelerate the pattern matching in YARA framework using an FPGA platform. Specifically, the main contributions of this paper are:

- *Modular implementation of YARA on FPGA*: We develop a deeply pipelined two dimensional architecture with a rule of YARA mapped to a stage in the architecture on FPGA. The rule to stage mapping allows for easy replacement and modification of YARA rules on FPGA.
- Our architecture on FPGA achieves a peak throughput of 21.8 Gbps for a ruleset of 1600 patterns distributed among 100 rules with each pattern having 50 characters. For larger rulesets with 200 YARA rules consisting of 6000 strings of 100 characters in each string, our architecture achieves throughput of 12.85 Gbps. In comparison to the peak performance of 1.45 Gbps for an implementation on CPU, we demonstrate 8.8× to 14.5× improvement in performance.

The rest of the paper is organized as follows: Section II provides background information about the YARA tool and pattern matching techniques while Section III describes prior

This material is based in part upon work supported by the National Science Foundation under Grant Number CCF-1116781

work. Section IV discusses the pattern matching architecture in detail. In Section V, various patterns, metrics used in performance evaluation. The experimental results and our analysis is provided in Section VI. Section VII concludes the paper along with directions for future work.

II. Background

A. YARA Framework

Rules represent the granularity of result of pattern matching in YARA. Analysis of a stream of characters for a set of rules returns match or no match as the result for each rule in the ruleset. A representative format of YARA rules is shown in Definition 1.

Definition 1

rule yara_example
{
  meta:
    description = "This is an example of YARA rule"
  strings:
    $a = "ZINGAWI2"
    $b = {6A 40 68 00 30 00 00 6A 14 8D 91}
    $c = /md5: [0-9a-zA-Z]32/
  condition:
    $a or $b and $c
}

Each YARA rule contains pattern(s) to be matched and a condition which is used to determine the final result. A YARA rule consists of the keyword "rule" followed the rule name and 3 components:

- *Meta*: This section of the rule is used to store the metadata such as description, date of creation, references etc. In the above definition, statement: *description = "This is an example of YARA rule"* represents the Meta section of the rule. This is not used in the matching process and is for information purposes only.
- *Strings*: The pattern(s) to be matched is defined in this part of the rule. 3 types of strings can be defined: Text, Hexadecimal, Regular expression. Each of these strings has a specific syntax as shown in the example rule. *"ZINGAWI2"* represents the text, *6A 40 68 00 30 00 00 6A 14 8D 91* represents hexadecimal numbers and *md5: [0-9a-zA-Z]32* represents the regular expression in the above example.
- *Condition*: This part of the rule determines the logic to combine the results of pattern matching of individual strings. It is defined as a Boolean expression. As shown in the above example: $a or $b and $c, it can be also be a combination of operators.

In addition to definition of rules, YARA framework can be used to perform pattern matching as well. When using YARA as a pattern matching tool, it accepts two files as input: one file containing data to be processed and another file containing rules to be matched. At the end of processing, the tool outputs the name of matching rules which are present in the data file.

B. Pattern Matching Techniques

There are two classes of pattern matching techniques: string matching (SM) and regular expression matching (REM). As the names suggest, SM is used to match a set of strings against a stream of incoming characters and REM refers to regular expression matching which are regular languages constructed using character classes over a fixed alphabet. Although SM and REM can have different matching circuits, since strings are a special case of regular expressions, we apply the REM approach to perform pattern matching of both strings and regular expressions in this paper. A pattern matching engine can be implemented as a finite state machine for each individual pattern. There are two types of finite state machines: Deterministic Finite Automata (DFA) and Nondeterministic Finite Automata (NFA).

In the DFA approach [13], all the patterns are combined into a single DFA and a single state is active at any point of time. This single active state is obtained by expanding the different states and state transition lookup table is used to determine the next state. In the NFA class of pattern matching [14], [15], each pattern is converted to a NFA and is processed in parallel independent of each other. Each character of the data input is sent to all the states in each matching engine. The outputs from the matching engines are combined to produce the final output. In this approach, multiple states are in active condition.

Both NFA and DFA techniques have advantages and drawbacks. DFA is efficient for simple regular expressions, but for complex regular expressions, the lookup table can be very large resulting in exponential memory explosion [13]. On the other hand, NFA has multiple state transitions for every character and result in higher throughput due to the parallel processing. The drawback of NFA is that they require more hardware resources [14], [16]. In this study, we focus on the design of NFA based pattern matching engines and their architecture on FPGA.

III. Related Work

Many research works have used YARA as a malware analysis framework. In [17], the authors use YARA as one of the detection methods to develop an analysis engine for malware identification. The paper focused on malicious websites and used other anti-virus engines for malware analysis. The authors in [18] develop methodology to protect against script based cyber attacks and YARA is used as a pattern matching tool in the process. Although [19] focuses on performance of malware detection tools, YARA is not considered and the target metric is to measure the effectiveness and accuracy of the tools. To the best of our knowledge, performance of YARA has not studied and the focus has been on using an implementation of YARA framework on general purpose processor to perform malware detection.

Pattern matching has been a topic of interest in the academia for many years. Large scale string matching on FPGA was designed in [20]. Although the authors utilize the parallel and deep pipeline capabilities of FPGA, the experiments is limited to small problem size (200 patterns). Authors study FPGA and GPU based pattern matching architectures in [21] on streams of spatio-temporal data from devices such as GPS and RFIDs. A memory efficient and modular architecture on FPGA is proposed in [22] for large scale string matching.

Regular expression matching architectures on FPGA have been studied before. NFA based regular expression matching on hardware was studied by Floyd and Ullman [23]. They used $O(n)$ circuit area to implement an n-state NFA. Sidhu and Prasanna [15] proposed an algorithm to translate a regular expression onto FPGA which is used by several other implementations [14], [16]. Shift register lookup tables for single character repetitions was implemented in [14]. Other optimizations such as centralized classification and common prefix extraction were developed. However, the architecture does not support multi character matching. Hardware accelerators for pattern matching in YARA framework have not been studied before and to the best of our knowledge, this is the first work to provide an FPGA implementation of pattern matching in the YARA framework.

IV. Pattern Matching Architecture on FPGA

In this paper, we extend the previous work of [24], [25] to modify the architecture in the context of pattern matching in YARA framework. Here, we present a brief overview of the architecture developed in [24], [25].

A. NFA Construction

Authors in [24] construct NFA for pattern matching in two steps. First, the patterns are parsed to a generate a token list. In this step, each pattern is transformed into a token list data structure which is a mutli-level linked list and then, multiple token lists are chained together by the fields in the data structure. In the second step, they modify McNaughton-Yamada Algorithm to generate the NFA. The modified algorithm used by the authors does not produce extra states and hence redundant ϵ-transitions are not generated. The output NFA of this step is used for implementation of pattern matching engines on FPGA.

B. Pattern Matching Engine

Each pattern is mapped to a pattern matching constructed based on the NFA of the pattern. A matching engine accepts data to be analyzed and the result of character classification as inputs and produces a single bit output representing a match or no match.

C. Stage

As described earlier, a YARA rule contains multiple patterns and a governing condition deciding the final outcome of processing that rule. Each pattern requires a matching engine and all the patterns in the same rule require additional logic to implement the condition aspect of the rule. In our architecture, we group all the pattern matching engines belonging to the same rule and the logic to implement the condition to form a *stage*. This structure of our design wherein a rule is logically mapped to a stage provides modularity and makes it easier to update the existing rules or add new rules in the future.

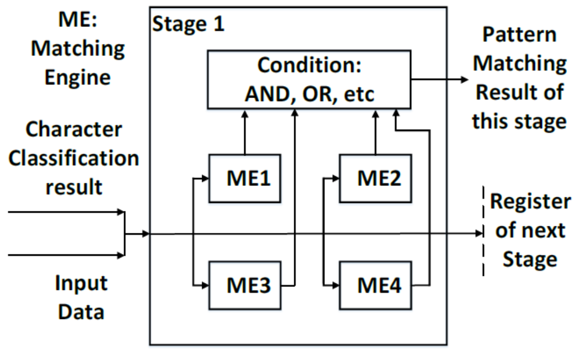


Fig. 1: Architecture of a Stage

A stage consisting of 4 matching engines is shown in Figure 1. The condition logic can be either “all of them”, “any of them” or expressions such as “2 of them”, “3 of them”. When mapped onto a FPGA, all these conditions are implemented using LUTs by the EDA software. Therefore, all these conditions will have the same effect on the clock frequency and resource usage irrespective of which condition is employed in a rule. Within a stage, the matching results of each engine is stored in a register. Later, all these results are combined as per the condition of the rule to obtain the final matching output.

D. Pipeline

Although staging helps in keeping connections local, the character classification outputs from BRAM needs to be used by all the stages as it is not practical to reserve one character classification for each stage and hence sharing of BRAMs is inevitable. Each pipeline has a group of stages which share a centralized character classification. In a given clock cycle, the incoming character is passed as input to all the matching engines in the first stage of the first pipeline (S1 in P1) along with the results of character classification. Both the input characters and results of character classification are buffered. In the next clock cycle, the buffered signals are passed as inputs to the second stage in the first pipeline (S2 in P1) and the first stage of second pipeline (S1 in P2) as well. Meanwhile, the first stage of the first pipeline (S1 in P1) accepts the second set of input characters. In this way, a deep pipeline is created with local interconnects resulting in a high throughput design. An architecture designed using 2 pipelines made of 4 stages in each pipeline is illustrated in Figure 2.

E. Overall Architecture

The performance of a design on FPGA depends on the maximum achievable clock frequency which is heavily influenced by the interconnects' routing of the architecture. With the complexity and size of the patterns the number of

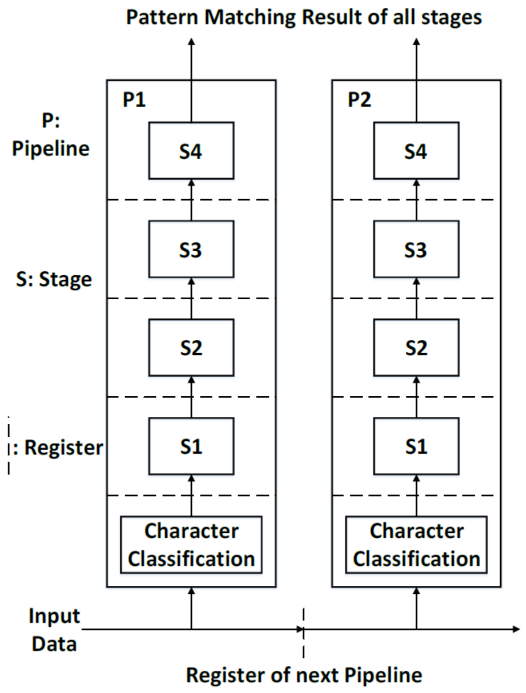


Fig. 2: Overall Architecture

states and logic delay also increases. In order to keep the connections local and short, the architecture is divided into a set of pipelines with each pipeline composed of multiple stages. Stages sharing the BRAMs for character classification are grouped together to form a pipeline. We group the pattern matching engines belonging to a YARA rule to form a stage. This structure consisting of stages and pipelines can be clocked at a higher frequency and results in high throughput due to the short interconnects between stages. A representative overall architecture consisting of 2 pipelines with 4 stages each is illustrated in Figure 2.

Performance of pattern matching on FPGA faces many challenges including: large number of patterns to be matched in parallel, efficient usage of on-chip resources, long interconnects due to thousands of states etc. Authors in [24], [25] have developed several optimizations to achieve high clock rate while still maintaining the capacity of patterns that fit on FPGA. We describe these optimizations below.

F. BRAM based Character Classification

Character class or character sets are used in regular expressions to match any character from a set of characters. This is a critical part of the architecture of matching engines and efficient implementation of character classification is necessary to obtain a high throughput architecture. For a character class of size n, pattern matching using single characters rather than a character class results in the number of states in the NFA to increase by a factor of n. For a character of 8 bits, any arbitrary character classification can be represented using 256 bits. A BRAM of size $256 \times s$, where s represents the number of unique states is used to perform character classification. States having a common character class share the output of character classification. BRAMs allow the matching of multiple character classes in one clock cycle which if implemented by logic would result in long clock cycle and more resources. An example of character classification using BRAMs is shown in Figure 3.

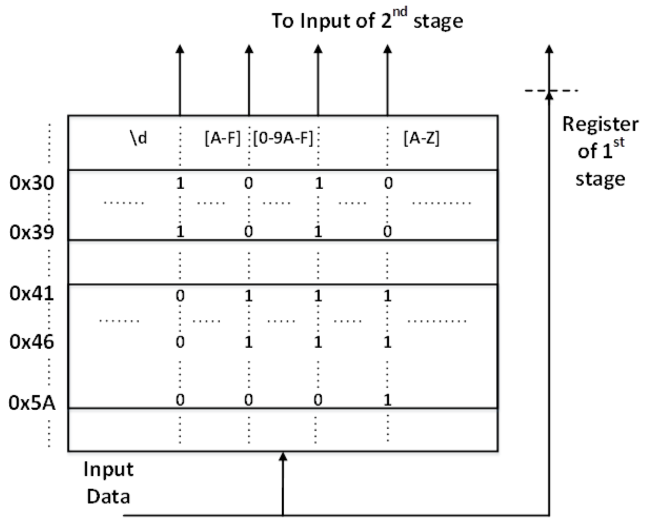


Fig. 3: Character Classification using BRAM [24], [25]

G. Multi-Character Matching

In [24], [25], multi-character matching circuits are designed using circuit level spatial stacking. Each state of an NFA in the resultant matching engine accepts multiple characters as inputs in every clock cycle. The m-character matching circuit having the same number of states and state transitions is constructed by merging m 1-character matching circuits. Although the multi-character matching circuit results in a lower clock frequency, the tradeoff for multiple characters being processed in each cycle pays off resulting in significant increase in throughput. An example of multi-character matching circuit for $/b + c\{15\}(ab|a[ac]) + d/$ with 2 characters per cycle is depicted in Figure 4.

V. Performance Evaluation

A. Our Focus

YARA framework allows the users to describe broad range of conditions. Even though the set of strings is limited, the condition which combines these strings can be complex. For example, a condition can contain:

- AND, OR and arbitrary combination of strings
- matching a string at a particular offset in the input
- iteration over string occurrences
- references to other rules

We studied the public repository of YARA rules [26], and among them, the majority of the conditions were either "all of them" or "any of them". In a sample size of approximately 1500 rules, "all of them", "any of them" and conditions such as "2 of them", "3 of them" constitute 900 rules, i.e., 60%

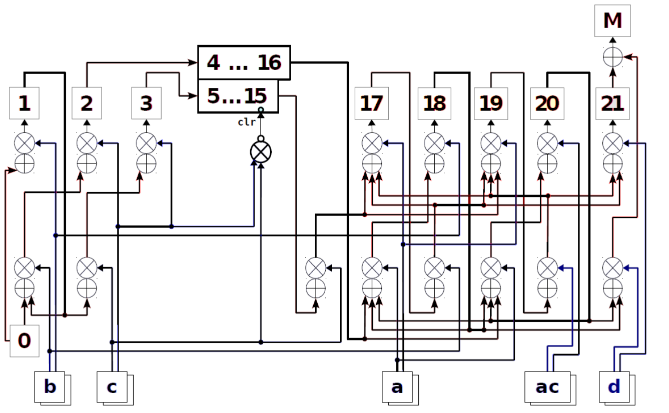


Fig. 4: Spatial Stacking: Multi-character Matching circuit for $/b + c\{15\}(ab|a[ac]) + d/$ [24], [25]

of the total number of rules. Therefore, in this paper, we limit our focus to these types of conditions.

B. Synthetic Patterns and Rulesets

The objective of this paper is to present an FPGA architecture for the implementation of YARA rules. We evaluate our architecture for different parameter sweeps of the rules and compare its performance with the implementation of YARA on general purpose processors. We studied a public repository of YARA rules [26] and it contained approximately 6000 patterns. We build synthetic rulesets of scale similar to the public ruleset with the following parameters: number of rules, number of patterns per rule, and the number of characters in a pattern. Our rulesets are classified based on the composition of patterns in a rule. A rule may contain a combination of the following: strings and regular expressions.

The characteristics of the synthetic rulesets containing strings as patterns are tabulated in Table I. Number of Patterns in the rulesets used in our experiments vary from 128 to 6000 patterns with the total number of characters ranging from 6400 characters to 300,000 characters.

Parameter	Range of Values
No. of Rules	8 - 192
Strings per Rule	16, 64
Characters per String	50, 100
Unique Characters per String	62
Condition	“all” or “any” or “some”

TABLE I: Ruleset Characteristics: Strings

In the public repository of YARA rules [26] we analyzed, regular expressions accounted for fewer than 250 patterns ($\leq 4\%$) out of approximately 6000 patterns. Therefore, we adopt regular expressions from Snort rules [2] repository. Regular expressions in Snort rulesets are PCRE but YARA does not implement all the features in PCRE [27]. Therefore, we limit the experiments with rulesets containing regular expressions to our architecture. The characteristics of the regular expressions used in our experiments are tabulated in Table II. Since the complexity of regular expressions cannot be derived using the simple parameter of number of characters, we present the statistics of regular expressions in terms of number of states in the NFA and number of states in the NFA with incoming transitions greater than 1 (denoted by *Trans-k, k>1* in Table II). The detailed experiments are presented in Section VI-A.

Parameter	Range of Values
No. of Rules	16 - 125
Regular expressions per Rule	16, 32
Condition	“all” or “any” or “some”
States	>100,000
Trans-k, k>1	≥20%

TABLE II: Ruleset Characteristics: Regular Expressions

C. Evaluation Metrics

We use Throughput as the performance metric for our experimental analysis. **Throughput** is defined as the number of bits processed in one unit of time. It is measured in Gigabits per second (Gbps).

The architecture of FPGA implementation and the CPU implementation of YARA framework are different. Pattern matching in the CPU implementation of YARA framework accepts files as data inputs and produces matching rules as the outpur result. Therefore, throughput is measured as the ratio of the size of the input file and the total execution time to process the file. In the context of FPGA, Throughput of an architecture is measured as the ratio of size of data accepted as input and the clock period of the design. Each character in the input data sample is presumed to be in ASCII encoding format in both implementations.

D. Experimental Setup

We chose the implementation of YARA framework on general purpose processors as our baseline implementation. We conducted experiments to determine the performance for various rulesets as described in Section V-B. We observed that the performance is independent of file sizes if the size is of the order of MBs. We performed experiments for file size equal to 20 MB as it resulted in minimal I/O overhead. The performance of software implementation was measured using the YARA module in Python library [28]. The profiling tool “cprofile” was used to measure the execution time of the program and to ensure that the throughput is not limited by the file system calls, we measure the time to process large number of files to find the average throughput. For example, instead of measuring the processing time of 1 file, we measure the time taken to process 10000 files of the same size and divide the total processing time by 10000 to obtain the execution time of a single file. We performed experiments on state-of-the art AMD Opteron 6278 processor. The target platform consists of 16 cores, each running at 2.4 GHz. Each core contains 16 KB L1 data cache, 64 KB L1 instruction cache and 2 MB L2 cache. L3 cache of 60 MB is shared among the 16 cores.

In our FPGA implementation, we used Virtex-7 FPGA device (xc7v2000tfhg1761-2) from Xilinx. All our results are post place and route (PAR) obtained after synthesis and implementation is performed using VHDL as the programming language and Xilinx Vivado 2014.2 as the EDA tool for simulation experiments. Our design accepts 8 input characters each cycle for the multi-character matching circuit. All our input characters are assumed to be in ASCII encoding format.

VI. Experimental Results

A. *Performance Comparison*

We evaluate the implementation of YARA framework on FPGA and compare its performance with that of software implementation on CPU. In our experiments, we first vary the number of rules (R) in Section VI-A1a and in Section VI-A1b, the number of characters in a pattern (C) is varied as a parameter keeping other parameters constant. Finally, we evaluate the effect of number of patterns in a rule (P) on performance in Section VI-A1c.

1) Strings as Patterns: We compare the performance of our architecture with the implementation on CPU for rulesets containing strings as patterns.

a) Variation in number of rules (R): The number of rules has a significant effect on the performance of pattern matching with larger number of rules resulting in higher execution time and hence, lower performance. We present the experimental results for two cases: Figure 5(a) depicts the effect of variation in number of rules on performance with $C = 100$ and $P = 100$ whereas, the parameters are set to $C = 50$, $P = 64$ and the number of rules as a variable in Figure 5(b).

Both the software implementation and the FPGA implementation show degradation in performance with the increase in number of rules. The degradation is marginal in the case of software implementation and the performance remains stable at approximately 1.45 Gbps. In general, with higher number of rules, more patterns need to processed by the matching engines and this translates to higher execution time and lower throughput. In the FPGA implementation, with the increase in number of rules, the number of stages and the number of pipelines increase. Therefore, the length of inter-stage connections increases and becomes the critical path. These might be signals such as *reset* at the input of a stage which has to go to all the matching engines in a stage and also to the global reset of the next stage as well. In the FPGA implementation, we observed the performance decreases when the resource consumption approaches 50% of the device being used. This is degradation in performance is expected as with higher amount of resources consumed by the design, lower amount of resources are available for routing and this causes long interconnects due to reduced flexibility.

b) Variation in number of characters in a pattern (C): The performance comparison is demonstrated for two sets of parameters. In Figure 5(c), $P = 16$ while varying $C = 50$ and $C = 100$ for different rulesets. We fix $P = 64$ in Figure 5(d) with C taking values of 50 and 100 for various rulesets. Software implementation maintains a performance in the range of 1.35 Gbps to 1.45 Gbps. As mentioned in Section [**?**], the throughput of the implementation on FPGA decreases when the resource consumption by the design approaches 50%. The number of characters in a pattern affects the resource consumption and higher the number of characters, larger the matching engine and consequently, larger stage for the pattern. Therefore, with higher number of characters the critical path increases leading to lower clock frequency. Even with the increase in number of characters from 50 to 100, our architecture maintains a frequency of 200 MHz for the largest ruleset of 192 rules with 16 patterns in each rule.

c) Variation in number of patterns per rule (P): The last parameter we vary in our experiments is the number of patterns in a rule. In each rule to be processed, all the patterns in the rule must be analyzed by the matching engine. Therefore, with higher number of patterns in a rule, each rule consumes more processing time. In this set of experiments, we evaluate the implementations for the number of patterns in a rule set to $P = 16$ and $P = 64$ and keeping the number of characters per pattern constant at $C = 100$ or $C = 100$.

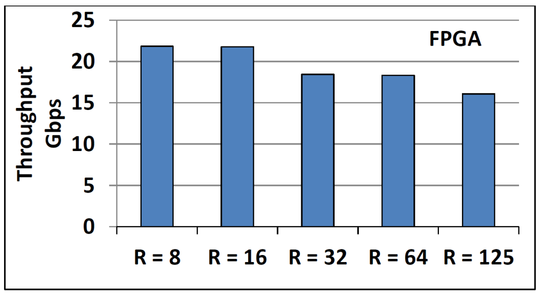
With the increase in number of patterns in a rule, the intra stage connections become the bottleneck in performance. These connections include output signals from BRAMs which have to be routed all the matching engines in a stage and later on, these signals need to be passed onto the inputs of the second stage. The experimental results are demonstrated in Figure 5(e) and Figure 5(f). We observe that the effect of varying number of patterns in a rule is equivalent to increasing the number of rules. Time to analyze a rule containing 64 patterns is analogous to analyzing 4 rules with 16 patterns with conditions as mentioned previously (Table I). This is evident from the performance comparison of 32_100_64 and 128_100_16. The number of patterns is reduced by a factor of 4, the number of rules is increase by the same factor and their performance is approximately equal.

2) Regular Expressions as Patterns: As mentioned earlier, the number of regular expressions available in the public repository is limited and therefore, we limit the experiments on the CPU implementation to these 240 regular expressions. On the FPGA implementation, in addition to the public rules, we perform experiments with regular expressions from Snort rulesets as well.

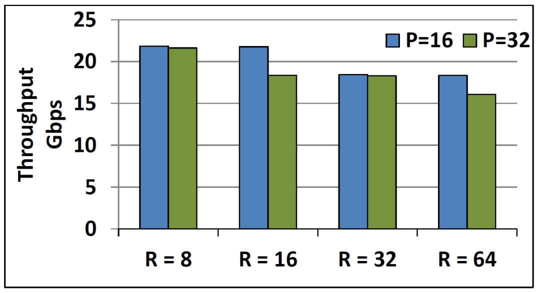
a) Public Rules: We group 240 regular expressions into 15 rules of 16 patterns per rule. The condition "all of them" is used in the rules. The performance of the CPU implementation for regular expression matching is significantly lower in comparison to string matching. The degradation in performance is attributed to the fact that regular expression matching is more complex in comparison to exact string matching. On our FPGA architecture, although the performance deteriorates, our optimizations for regular expression matching help in achieving a throughput which is significantly higher than the implementation on CPU. The peak performance of the implementation on CPU is 0.160

Gbps in comparison to a peak performance of 18 Gbps on FPGA.

b) Variation in number of rules (R): The performance comparison for rulesets containing regular expressions as patterns is illustrated in Figure 6(a). It can be observed that the performance in Figure 6(a) is marginally lower in comparison with that of the performance shown in Figure 5(a). For example, in the context of string matching, the peak throughput for 32 rules with 16 strings in each rule is 21.6 Gbps whereas, for 32 rules with 16 regular expressions in each rule, the performance is 18.4 Gbps. Strings and regular expressions cannot be compared based on length or number, and the performance is dependent on the structure and operators in the regular expressions. Operators such as Kleene closure ($*$) and Union ($|$) operators result in large number of states and feedback connections with long wires. When these operators are combined with repetition operator (m, n), the number of states in a NFA increases significantly. Higher number of states translates to higher consumption of logic resources and results in complex interconnect routing. This translates to a long critical path and consequently low clock frequency.



(a) Variation in number of rules (R) with P = 16



(b) Variation in number of regexes per rule (P)

Fig. 6: Throughput Comparison of FPGA based Regular Expression Matching: varying R and P

c) Variation in number of patterns (P): We now vary the number of regular expressions in a rule. Figure 6(b) depicts the performance comparison between rulesets containing 16 and 32 regexes per rule. Analysis of experiments for variation in number of strings per rule VI-A1c can be extended to the variation in number of regular expressions per rule as well. For a given ruleset, keeping the number of rules constant, as the number of regular expressions in a rule increases, the throughput decreases. This is expected as the amount of logic resources increases significantly with the number of patterns in a rule. As mentioned in the previous section, the clock frequency is further lowered due to the routing of long interconnects.

B. Evaluation of Software Implementation

In this section, we evaluate the software implementation of YARA framework on CPU for rulesets larger than the rules evaluated in Section VI-A. The total number of patterns in these rulesets range from 5000 to 200000.

1) Variation in number of rules (R): We fix the parameters C and P at a specific value and vary the number of rules R. We repeat this experiment for different values of C and P as illustrated in Figure 7(a) and Figure 7(b). We observe that the software implementation of YARA framework sustains a performance in the vicinity of 1.4 Gbps. The throughput does not appear to decrease even if R is increased to a value of 1000. Although this is encouraging, the performance does not improve for a smaller ruleset and it maintains the throughput of $\approx$1.45 Gbps.

2) Variation in number of characters in a pattern (C): We vary the number of characters in a pattern C from 50 to 250 and present the results for rulesets of size $R = 100$ and $R = 1000$ in Figure 7(c) and Figure 7(d) respectively. For a ruleset with R and P as constants, the variation in C does not appear to have a significant impact on the performance.

3) Variation in number of patterns in a rule (P): Finally, we evaluate the effect of variation in P on the performance. We fix the values of R to 100 and 1000 in Figure 7(e) and Figure 7(f). In both the figures, we perform experiments with a fixed value of C and varying P. Different lines indicate different values of C. The increase in number of patterns in a rule causes a marginal decrease in performance and the throughput decreases by 15% from 1.4 Gbps to reach a value of 1.2 Gbps even if the number of patterns are varied from 5 to 250.

C. Scalability Analysis

The size of a ruleset that can be implemented on one FPGA device is constrained by the amount of resources available on an FPGA. As mentioned in Section VI-A1a, the performance of the FPGA implementation is also dependent on the amount of resources available on FPGA. A ruleset composed of 200 rules with 16 pattern per rule and 100 characters per pattern is equivalent to rulesets with larger number of rules (>200) with lower number of patterns per rule (<16) or lower number of characters per pattern (<100). The performance of both these rulesets was seen to be similar because the amount of resources consumed by both the rulesets is approximately equal. Figure 5(e) illustrates an example of this scenario as we observe the performance of 48_50_64 and 96_50_16.

Although we limit the scope of this paper to all patterns implemented on a single FPGA, if the total number of patterns to be matched is greater than our ruleset in terms of size of a pattern and the number of patterns, they can be distributed among multiple FPGAs which can operate in parallel. This does not deteriorate the clock frequency and hence the architecture can maintain the throughput without degradation in performance.

We should also note that with multiple CPU cores running in parallel, it is possible to increase throughput of CPU based implementation. For string matching, the performance tends to be memory and I/O bound. In general, FPGA is well suited for accelerating streaming problems that are memory and I/O bound as well. However, a detailed multi-core scalability analysis for this kind of problems is out of the scope of this paper.

VII. Conclusion and Future Work

In this paper, we presented a high performance architecture on FPGA to accelerate pattern matching in the YARA framework. We built a modular architecture with a one to one mapping between a YARA rule and a stage in the FPGA architecture. We presented a parameter sweep of YARA rules on our architecture and compare our performance with the implementation on CPU through extensive experiments with rulesets composed of strings and regular expressions. Our FPGA architecture achieves a throughput of up to 21.8 Gbps with 14.5× improvement in performance in comparison with the throughput of 1.45 Gbps for the implementation on CPU.

In the future, we plan to implement the complete YARA ruleset using a heterogeneous architecture consisting of CPU and FPGA as the accelerator. Conditions such as backreferences to other rules, iteration over string occurrences and matching a string at a particular offset can be implemented using software running on CPU, whereas the simple conditions such as logical combination of strings can be implemented on FPGA to achieve high throughput. We plan to study the various tradeoffs in the above architecture to achieve high performance for the complicated conditions.

References

[1] Bro. Intrusion Dectection System. http://bro-ids.org.

[2] Snort. Intrusion Dectection System. http://www.snort.org/.

[3] YARA. Patter Matching Tool. http://plusvic.github.io/yara/.

[4] Symantec. Using YARA in Symantec. http://www.symantec.com/connect/blogs/performing-incident-response-using-yara.

[5] Cuckoo Module in YARA. http://yara.readthedocs.org/en/v3.4.0/modules/cuckoo.html.

[6] VirusTotal. https://www.virustotal.com/.

[7] Y.H. Cho and W.H. Mangione-Smith. Deep packet filter with dedicated logic and read only memories. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 125–134, April 2004.

[8] Nicholas Weaver, Vern Paxson, and Jose M Gonzalez. The shunt: an FPGA-based accelerator for network intrusion prevention. In *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 199–206. ACM, 2007.

[9] Gajanan S Jedhe, Arun Ramamoorthy, and Kuruvilla Varghese. A scalable high throughput firewall in FPGA. In *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*, pages 43–52. Ieee, 2008.

[10] Antonis Nikitakis and Loannis Papaefstathiou. A memory-efficient FPGA-based classification engine. In *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*, pages 53–62. IEEE, 2008.

[11] Haoyu Song and John W Lockwood. Efficient packet classification for network intrusion detection using FPGA. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 238–245. ACM, 2005.

[12] Xilinx. Virtex 7 FPGA. http://www.xilinx.com/support/documentation/data_sheets/ds183_Virtex_7_Data_Sheet.pdf.

[13] Fang Yu, Zhifeng Chen, Yanlei Diao, TV Lakshman, and Randy H Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102. ACM, 2006.

[14] Joao Bispo, Ioannis Sourdis, Joao MP Cardoso, and Stamatis Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 119–126. IEEE, 2006.

[15] Reetinder Sidhu and Viktor K Prasanna. Fast regular expression matching using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*, pages 227–238. IEEE, 2001.

[16] Christopher R Clark and David E Schimmel. Scalable pattern matching for high speed networks. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 249–257. IEEE, 2004.

[17] Masood Mansoori, Ian Welch, and Qiang Fu. YALIH, yet another low interaction honeyclient. In *Proceedings of the Twelfth Australasian Information Security Conference-Volume 149*, pages 7–15. Australian Computer Society, Inc., 2014.

[18] Jong-Hun Jung, Hwan-Kuk Kim, Hyun-lock Choo, and Lim ByungUk. The Protection Technology of Script-Based Cyber Attack. *Journal of Communication and Computer*, 12:91–99, 2015.

[19] Sudhir Kumar Pandey and BM Mehtre. Performance of malware detection tools: A comparison. In *Advanced Communication Control and Computing Technologies (ICACCCT), 2014 International Conference on*, pages 1811–1817. IEEE, 2014.

[20] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system. In *Field Programmable Logic and Application*, pages 880–889. Springer, 2003.

[21] Roger Moussalli, Ildar Absalyamov, Marcos R Vieira, Walid Najjar, and Vassilis J Tsotras. High performance FPGA and GPU complex pattern matching over spatio-temporal streams. *GeoInformatica*, 19(2):405–434, 2014.

[22] Hoang Le and Viktor K Prasanna. A memory-efficient and modular approach for large-scale string pattern matching. *Computers, IEEE Transactions on*, 62(5):844–857, 2013.

[23] Robert W Floyd and Jeffrey D Ullman. The compilation of regular expressions into integrated circuits. *Journal of the ACM (JACM)*, 29(3):603–622, 1982.

[24] Yi-Hua Edward Yang and Viktor K Prasanna. High-Performance and Compact Architecture for Regular Expression Matching on Fpga. *Computers, IEEE Transactions on*, 61(7):1013–1025, 2012.

[25] Yi-Hua E Yang, Weirong Jiang, and Viktor K Prasanna. Compact architecture for high-throughput regular expression matching on FPGA. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 30–39. ACM, 2008.

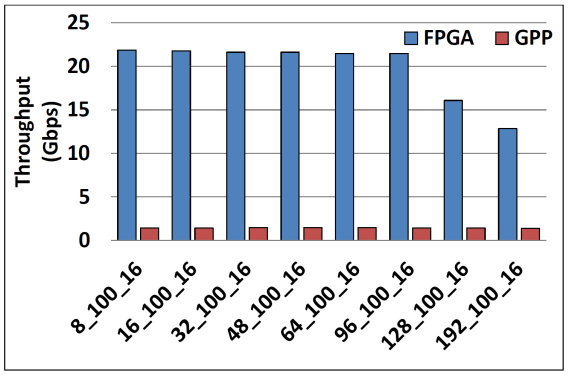
[26] Public Repository of YARA Rules. https://github.com/Yara-Rules/rules.

[27] Regular Expression in YARA. http://yara.readthedocs.org/en/latest/writingrules.html.

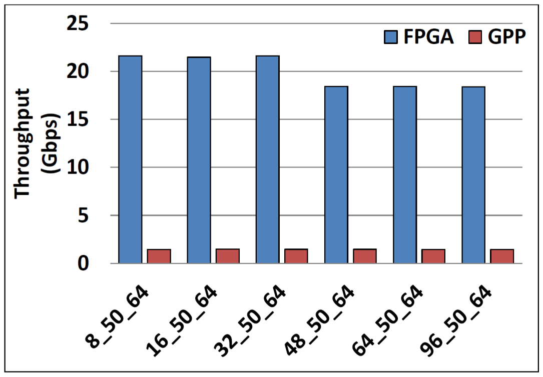
[28] Python Library for YARA. http://yara.readthedocs.org/en/latest/yarapython.html.

[29] Young Han Choi, Byoung Jin Han, Byung Chul Bae, Hyung Geun Oh, and Ki Wook Sohn. Toward extracting malware features for classification using static and dynamic analysis. In *Computing and Networking Technology (ICCNT), 2012 8th International Conference on*, pages 126–129. IEEE, 2012.

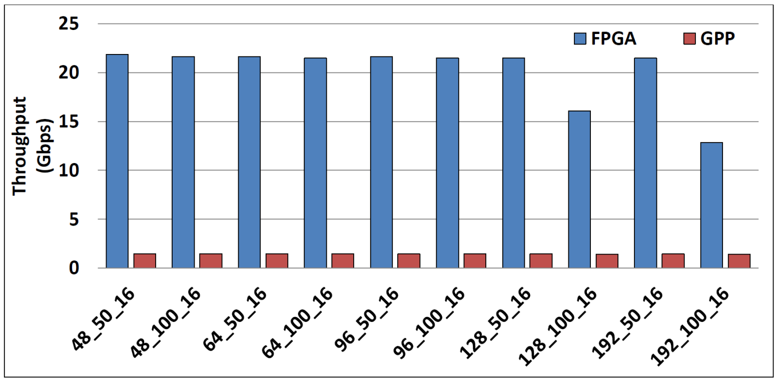
[30] Jeff Gennari and David French. Defining malware families based on analyst insights. In *Technologies for Homeland Security (HST), 2011 IEEE International Conference on*, pages 396–401. IEEE, 2011.



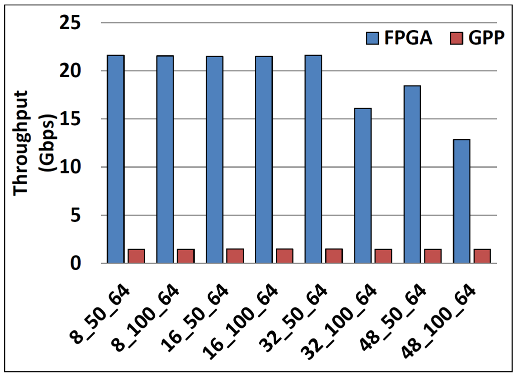
(a) Variation in number of rules (R) with C = 100, P = 16



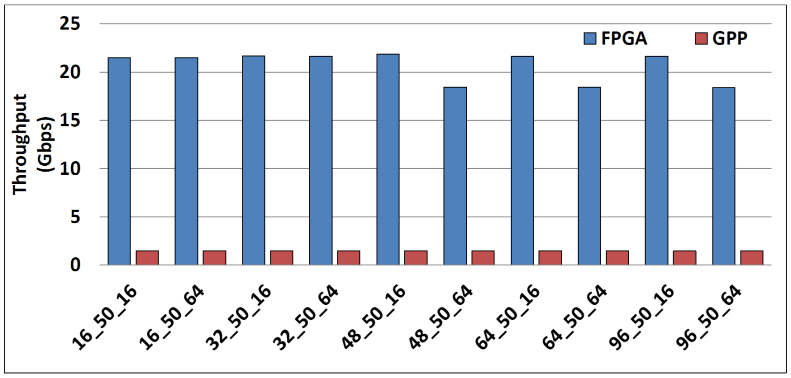
(b) Variation in number of rules (R) with C = 50, P = 64



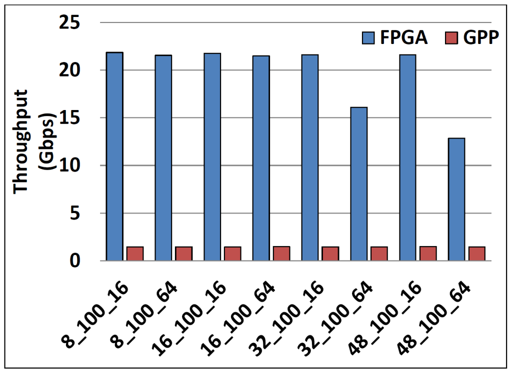
(c) Variation in number of characters (C) with P = 16



(d) Variation in number of characters (C) with P = 64

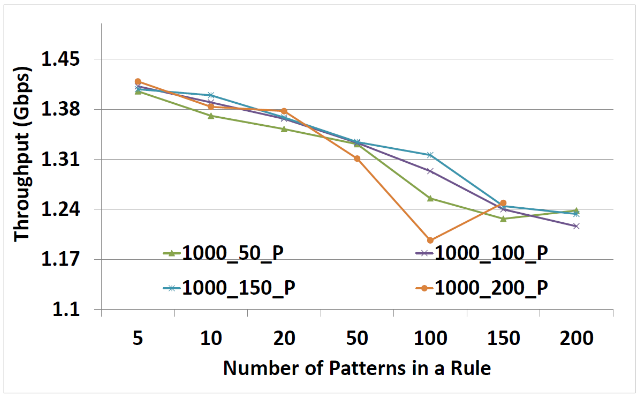


(e) Variation in number of patterns (P) with C = 50

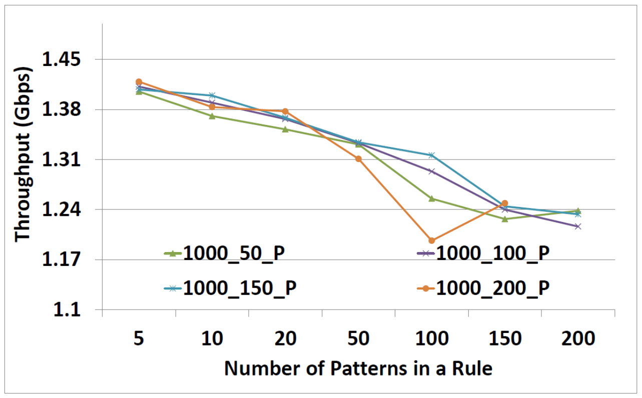


(f) Variation in number of patterns (P) with C = 100

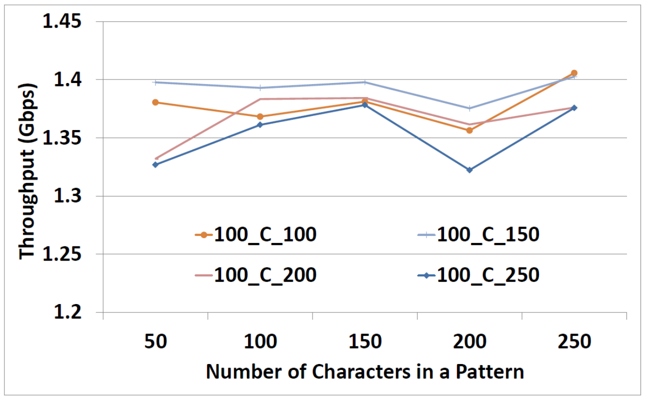
Fig. 5: Throughput comparison of FPGA and CPU based String Matching with various (R, C, P) configurations



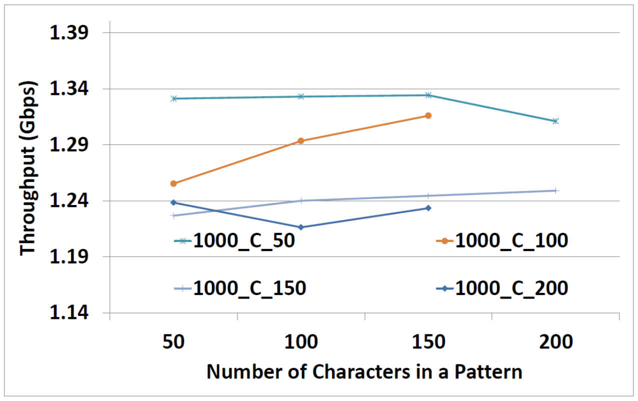
(a) Variation in number of rules (R)



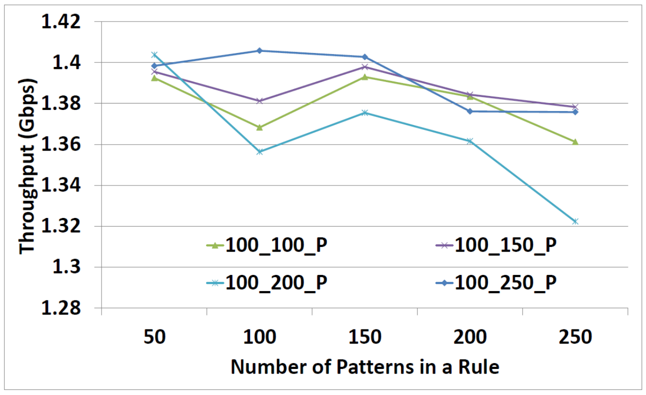
(b) Variation in number of rules (R)



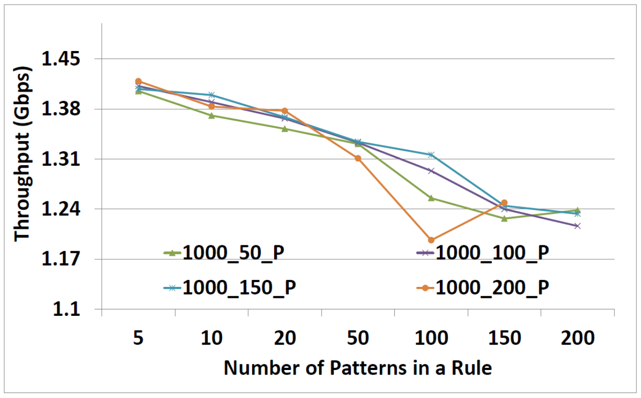
(c) Variation in number of characters (C) with R=100



(d) Variation in number of characters (C) with R=1000



(e) Variation in number of patterns (P) with R=100



(f) Variation in number of patterns (P) with R=1000

Fig. 7: Throughput comparison of CPU based String Matching with various (R, C, P) configurations