# FPGA-based Acceleration of Pattern Matching in YARA

Shreyas G. Singapura[1], Yi-Hua E. Yang[2], Anand Panangadan[3], Tamas Nemeth[4], Peter Ng[4], and Viktor K. Prasanna[1⋆]

[1] Ming-Hsieh Dept. of Electrical Engineering, University of Southern California, USA, {singapur and prasanna}@usc.edu,
[2] Google Inc., Mountain View, CA, yeyang@google.com
[3] California State University, Fullerton, USA, apanangadan@fullerton.edu
[4] Chevron, Tamas.Nemeth@chevron.com, PeterN@chevron.com,

**Abstract.** String and regular expression pattern matching is an integral part of intrusion detection systems to detect potential threats. YARA is a pattern matching framework to identify malicious content by defining complex patterns and signatures. Software implementations of YARA on CPU do not meet the throughput requirements of core networks. We present a FPGA based hardware accelerator to boost the performance of pattern matching in YARA framework. The proposed architecture consists of pattern matching engines organized as two-dimensional stages and pipelines. We implemented rulesets of sizes varying from 8 to 200 rules with total number of patterns ranging 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 with the throughput of 1.45 Gbps for a software implementation on a state of the art multi-core platform.

## 1 Introduction

Intrusion detection systems [1,2] in applications such as network security, content filtering and data mining use pattern matching as a key kernel. The number of malware patterns and their complexity is increasing due to the large number and diversity of devices. YARA [3] has emerged as a widely used tool [1,2] which helps users to create signatures and patterns specific to malware patterns and perform traffic analysis to detect these patterns. It is also used in research tools such as Cuckoo [3] and VirusTotal [4]. With regards to the implementation of YARA, the focus has been on its software implementation on general purpose processors (CPU). CPU-based implementations are however 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 CPU-based solutions may be unable to keep up with the requirements of the next generation network processing systems [5].

⋆ This work is supported by Chevron U.S.A. at the University of Southern California.

In recent years, interest in hardware accelerators such as GPUs and FPGAs for real time network processing systems has increased [6]. This is due to their ability to perform application specific computations faster than software-based implementations on CPU. In this paper, we focus on FPGA as the hardware accelerator. State of the art FPGA devices [7] provide large amount of parallel logic to create deep pipelines and large amount of high bandwidth on-chip memory. FPGAs are also programmable and can be optimized to match the application requirements.

Our work focuses on implementation of pattern matching algorithms on FPGA in order to accelerate the YARA tool. We extend the work of [8] and modify the architecture to perform pattern matching in YARA. To the best of our knowledge, this is the first work to accelerate the pattern matching in YARA framework using an FPGA platform. Main contributions of this paper are:

- *Modular implementation of YARA on FPGA*: We develop a mapping of every multi-pattern rule to a stage in a 2-D pipeline on FPGA.
- Our architecture achieves a throughput in the range of 12.8 Gbps - 21.8 Gbps for rulesets consisting of 6000 and 1600 patterns respectively. In comparison with the implementation on CPU, we demonstrate 8.8× to 14.5× improvement in performance.

The rest of the paper is organized as follows: Section 2 provides background information about the YARA tool and pattern matching techniques while Section 3 describes prior work. Section 4 discusses the pattern matching architecture in detail. In Section 5, various patterns and metrics used in performance evaluation are explained. The experimental results and our analysis is provided in Section 6. Section 7 concludes the paper with directions for future work.

## 2 Background

### 2.1 YARA Framework

In the YARA framework, input data is analyzed for a set of rules and match or no match is output 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 by 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. *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. *Condition:* This part of the rule determines the logic to combine the results of pattern matching of individual strings.

YARA framework can also be used to perform pattern matching. While 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. The tool outputs the name of matching rules present in the data file.

### 2.2 Pattern Matching Techniques

There are two classes of pattern matching techniques: string matching (SM) and regular expression matching (REM). 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. 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. In this study, we focus on the design of Nondeterministic Finite Automata (NFA) based pattern matching engines wherein each pattern is converted to a NFA and is processed in parallel independent of each other.

## 3 Related Work

Many research works have used YARA as a malware analysis framework. In [9], the authors use YARA as one of the detection methods to develop an analysis engine for malware identification. The authors in [10] develop methodology to protect against script based cyber attacks and YARA is used as a pattern matching tool in the process. To the best of our knowledge, performance of YARA has not been 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 [11]. Although the authors utilize FPGA for implementation, the experiments is limited to small problem size (200 patterns). A memory efficient and modular architecture on FPGA is proposed in [12] 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 [13]. They used $\boldsymbol{O(n)}$ circuit area to implement an $\boldsymbol{n}$-state NFA. Sidhu and Prasanna [14] proposed an algorithm to translate a regular expression

onto FPGA which is used by several other implementations [15]. Shift register lookup tables for single character repetitions was implemented in [15]. 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.

## 4 Pattern Matching Architecture on FPGA

In this paper, we extend the previous work of [8] and modify the architecture in the context of pattern matching in YARA framework. Here, we present a brief overview of the architecture developed in [8] and the modification required for implementing YARA rules.

***NFA Construction and Pattern Matching Engine:*** The patterns are transformed into a token list data structure which is a mutli-level linked list and multiple token lists are chained together to generate the NFA. Each pattern is mapped to a pattern matching engine 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 outputs a match or no match result.

***Stage:*** Each pattern in a YARA rule 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*. A stage consisting of 4 matching engines is shown in Fig. 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. The matching results are combined as per the condition of the rule to obtain the final matching output.

***Pipeline:*** Each pipeline has a group of stages sharing a centralized BRAM which is used for 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. 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. An architecture using 2 pipelines made of 4 stages is illustrated in Fig. 2.

***Mapping YARA Ruleset to Our Architecture:*** We map each rule in a given YARA ruleset to a stage in our architecture. Each stage consists of matching engines equal in number to the number of patterns in the rule and a LUT to

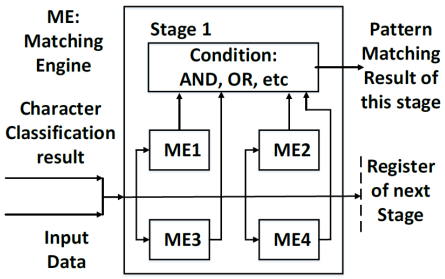



Fig. 1: Architecture of a Stage

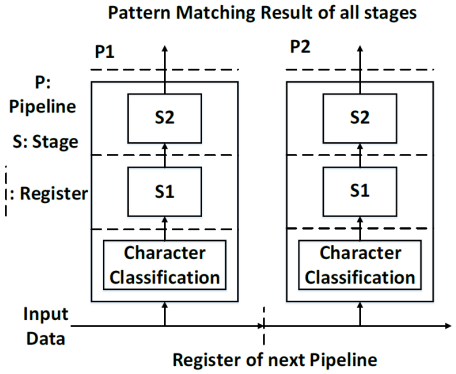



Fig. 2: Overall Architecture

implement condition aspect of the rule. Each stage produces a single bit output corresponding to match/no match for that particular rule. Our architecture consists of multiple pipelines with multiple stages in each pipeline and the output of all the stages across all the pipelines cumulatively form the result of pattern matching of the input for the ruleset.

## 5 Performance Evaluation

### 5.1 Rulesets

We use two types of rulesets for our performance evaluation: publicly available real-world YARA rules [16], plus a larger number of synthetic rules.

**Real World Rules**: The public repository of real-world YARA rules [16] contains about 6000 patterns distributed among 1500 rules. Most of the patterns in the public YARA rules were strings ($\geq$ **95%**); regexes accounted for fewer than 250 ($\leq$ **4%**) patterns.

**Synthetic Rules**: To increase the number and diversity of rules, two types of synthetic rules are constructed: *string-based rulesets* contain rules that match strings only and *regex-based rulesets* contain rules that match regexes.

For the string-based rulesets, each string is a sequence of random alphanumeric characters. For the regex-based rulesets, we borrowed the PCRE (Perl-Compatible Regex Expression) patterns from a Snortrules [2] repository. Fig. 3 describes the characteristics of different rulesets.

Although in principle, a YARA rule can have complex conditions such as backreferences and iteration over string occurrences; in the public repository [16] of 1500 rules, "all of them", "any of them" and conditions such as "2 of them", "3 of them" constitute 900 rules, i.e., 60% of the total number of rules. Therefore, for both string-based and regex-based rulesets, we use similar conditions in the rules.

### 5.2 System Configurations

The proposed YARA acceleration architecture (YARA-FPGA) is evaluated against an official YARA software (YARA-CPU) as the baseline. With YARA-CPU, experiments were conducted through the Python interface in YARA software [3] and Python's "cProfile" library. The experiments were performed on a platform with an AMD Opteron 6278 processor and 64 GB DDR3 memory. The processor consists of 16 cores, running at 2.4 GHz with 16 KB L1 data 2 MB L2 and 8 MB L3 cache memories. Only 1 core is utilized by the YARA software as we are comparing the per-stream throughput, rather than the aggregated throughput.

With YARA-FPGA, the acceleration engine was implemented on a Xilinx Virtex-7 FPGA device (xc7v2000tfhg1761-2). The design was written in VHDL and configured to match 8 input characters per clock cycle. A character is assumed to have 8 bits. The performance was estimated from post place and route (PAR) resource and timing results reported by Xilinx Vivado 2014.2.

### 5.3 Evaluation Metrics

We use throughput as the primary metric for performance evaluation. *Throughput* is defined as the amount of input data processed (matched) by the YARA implementation per unit time. It is measured in Gigabits per second (Gbps).

With YARA-CPU, throughput is calculated as the *input file* size divided by the run time of the pattern matching routine *over* that file (as reported by cProfile). With YARA-FPGA, throughput is estimated as the input bus width (8 input characters per cycle) divided by the post place-and-route clock period.

## 6 Experimental Results

We evaluate the implementation of YARA on FPGA and compare its performance with that of software implementation on CPU for various values of number of rules ($\boldsymbol{R}$), number of patterns ($\boldsymbol{P}$) and number of characters per pattern($\boldsymbol{C}$).

### 6.1 String-based Rulesets

The performance comparison of YARA-CPU and YARA-FPGA is illustrated in Fig. 4(a). The parameters $\boldsymbol{R}$, $\boldsymbol{P}$ and $\boldsymbol{C}$ do not appear to affect the string

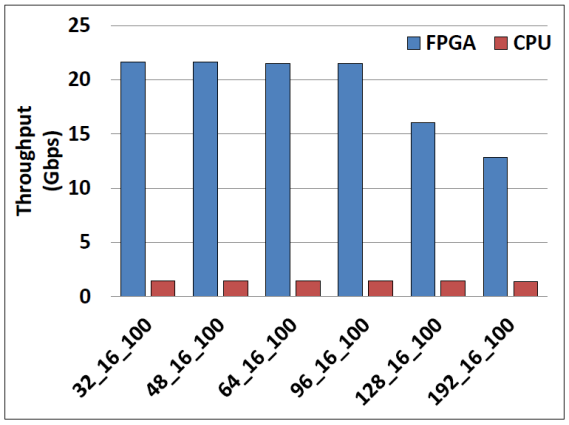
| Parameter | Range of Values |
|---|---|
| No. of Rules | 8 - 192 |
| Strings per Rule | 16, 64 |
| Characters per String | 50, 100 |
| Condition | "all"/"any"/"some" |

(a) String based Rulesets

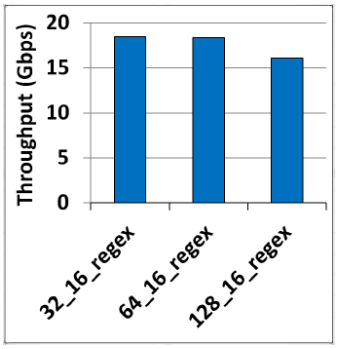
| Parameter | Range of Values |
|---|---|
| No. of Rules | 16 - 125 |
| Regexes per Rule | 16, 32 |
| States | >100,000 |
| Condition | "all"/"any"/"some" |

(b) Regex based Rulesets

Fig. 3: Characteristics of Rulesets




(a) String-based Rulesets: R_P_C




(b) Regex-based Rulesets: R_P_regex

Fig. 4: Throughput comparison of FPGA and CPU based Pattern Matching

matching performance of YARA-CPU significantly. Performance of the software implementation remains stable at approximately 1.45 Gbps. On the other hand, performance of YARA-FPGA implementation shows performance degradation due to decreasing clock frequency with increasing $\boldsymbol{R}$, $\boldsymbol{P}$ and $\boldsymbol{C}$. 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. Increasing $\boldsymbol{R}$, $\boldsymbol{P}$ and $\boldsymbol{C}$ all have the net effect of increasing the circuit size, which increases both logic depth and interconnect length.

### 6.2 Regex-based Rulesets

We grouped the 240 regular expressions obtained from Real World Rules (Section 5.1) into 15 rules of 16 patterns per rule. The condition "all of them" is used in the rules. The peak performance of the implementation on CPU is 0.160 Gbps. The degradation in performance is attributed to the fact that regular expression matching is more complex in comparison to exact string matching. YARA-FPGA achieves a throughput of 18.5 Gbps, an improvement of over 115×, therefore, we do not perform experiments for regex-based rulesets with Snort rules on YARA-CPU and focus on the FPGA implementation in this section.

The performance comparison for rulesets containing regular expressions as patterns is illustrated in Fig. 4(b). The performance of regex-based rulesets depends on the structure and operators in the regular expressions. Operations such as Kleene closure ($*$) and Union ($|$) operators result in large number of state transitions and longer wires.

## 7 Conclusion and Future Work

In this paper, we presented a high performance modular architecture on FPGA to accelerate pattern matching in the YARA framework. We compared the performance of FPGA implementation with the software implementation on CPU. Our FPGA architecture achieves a throughput of up to 21.8 Gbps with 14.5× improvement in performance in comparison with the throughput of 1.47 Gbps for the implementation on CPU. In the future, we plan to implement the YARA framework using a heterogeneous architecture consisting of CPU and FPGA for complex conditions such as backreferences and iteration over string occurrences.

## 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. VirusTotal, https://www.virustotal.com/.
5. N. Weaver, V. Paxson, and J. 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.* ACM, 2007, pp. 199–206.
6. A. Nikitakis and L. Papaefstathiou, "A Memory-efficient FPGA-based Classification Engine," in *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on.* IEEE, 2008, pp. 53–62.
7. Xilinx. Virtex 7 FPGA. http://www.xilinx.com/support/documentation/data_sheets/ds183_Virtex_7_Data_Sheet.pdf.
8. Y.-H. E. Yang and V. K. Prasanna, "High-Performance and Compact Architecture for Regular Expression Matching on Fpga," *Computers, IEEE Transactions on*, vol. 61, no. 7, pp. 1013–1025, 2012.
9. M. Mansoori, I. Welch, and Q. Fu, "YALIH, yet another low interaction honeyclient," in *Proceedings of the Twelfth Australasian Information Security Conference-Volume 149.* Australian Computer Society, Inc., 2014, pp. 7–15.
10. J.-H. Jung, H.-K. Kim, H.-l. Choo, and L. ByungUk, "The Protection Technology of Script-Based Cyber Attack," *Journal of Communication and Computer*, vol. 12, pp. 91–99, 2015.
11. I. Sourdis and D. Pnevmatikatos, "Fast, Large-scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System," in *Field Programmable Logic and Application.* Springer, 2003, pp. 880–889.
12. H. Le and V. K. Prasanna, "A Memory-efficient and Modular Lpproach for Large-scale String Pattern Matching," *Computers, IEEE Transactions on*, vol. 62, no. 5, pp. 844–857, 2013.
13. R. W. Floyd and J. D. Ullman, "The Compilation of Regular xpressions into Integrated Circuits," *Journal of the ACM (JACM)*, vol. 29, no. 3, pp. 603–622, 1982.
14. R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," in *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on.* IEEE, 2001, pp. 227–238.
15. J. Bispo, I. Sourdis, J. M. Cardoso, and S. Vassiliadis, "Regular Expression Matching for Reconfigurable Packet Inspection," in *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on.* IEEE, 2006, pp. 119–126.
16. "Public Repository of YARA Rules," https://github.com/Yara-Rules/rules.