New Regular Expressions on Old Accelerators

Jackson Woodruff
University of Edinburgh
Edinburgh, Scotland
J.C.Woodruff@sms.ed.ac.uk

Michael F.P. O'Boyle
University of Edinburgh
Edinburgh, Scotland
mob@inf.ed.ac.uk

Regular expressions (regexes) play a key role in a wide range of systems including network intrusion detection. FPGA accelerators can provide power savings over CPUs by exploiting MISD parallelism inherent in regex processing. However, FPGA solutions are brittle, requiring hours to reprogram when rulesets change, while real-world security threats evolve rapidly.

We present RXPSC (Regular eXPression Structural Compiler), a compiler designed to compile new regexes to existing regex accelerators. We use input-stream translation to enable fixed FPGA accelerators to accelerate new patterns with minimal overhead and little update delay. Compared to a solution where new regexes run on a CPU, RXPSC reduces CPU load by more than a factor of ten for 84% of unseen regexes in ANMLZoo benchmarks.

***Index Terms*—regular expressions, accelerator, compiler**

I. Introduction

Regex processing is critical in a wide range of fields, from genome processing to network intrusion detection [1]. This range of applications, and the applicability of the computational model to a MISD processor model, has driven the development of a number of hardware accelerators. IP companies such as Grovf [2] currently offer automata accelerators, and Micron offers a physical automata processing board [3]. FPGAs are also able to take advantage of the parallelism available in regex processors [4], offering an alternative to high ASIC design costs, latencies, required expertise and low design utilization [5]. In particular, the network intrusion detection setting has seen significant support for reconfigurable regex acceleration, from P4 programmable architectures [6] to FPGA-based accelerators [7]. However, FPGA-based automata accelerators face problems with the dynamic nature of network intrusion detection rules. As Xu et al. [8] state, "FPGAs do not support fast dynamic updates, so are not applicable in network security applications where signature rules are altered frequently".

Existing work addressing pattern update times focuses either on better utilizing FPGA toolchains [9] which still leaves pattern update times orders of magnitude too long, or on introducing architectures that make reconfigurability easier [10], [11]. Of these architectures, generalized FPGA overlays such as NAPOLY [11] reduce compile time drastically, but have far less throughput and capacity than single-level reconfigurable designs [4]. There have been attempts at fast reprogrammability [10] but they lack compiler support, and only support regex *replacement* rather than regex *addition*.

We present a methodology for supporting acceleration of new regexes on existing accelerators. We provide a compiler, RXPSC which supports dynamically injecting regexes. Our methodology is *independent* of the implementation of the underlying accelerator, and allows the new and old accelerator to coexist with negligible latency and no throughput penalty. We compare to an existing automatic compilation technique that can be applied to this problem, prefix merging, as discussed by Wadden et al. [1] and see that prefix merging is limited by its requirements for complete equality between prefixes, an assumption that often does not hold.

In our methodology, regexes are efficiently *translated* to each other using stateless translators, which convert the input-stream character-by-character to a new input-stream and pass the new input stream into an existing accelerator. Compared to a hybrid solution where new regexes are run on the CPU, this translation can reduce the number of bytes that must be checked on the CPU by more than a factor of ten in 84% of cases across ANMLZoo [1]. Stateless translation allows regexes to share underlying accelerators *regardless of their implementation* and provides easy scalability for higher performance. We expect our work to be particularly useful in cases where not all regexes must be run over every input — for example anti-virus programs with different rules for different file types, protein-search operations with different proteins or network intrusion detection systems where different rules apply to different protocols/ports. This allows us to find accelerators that would not otherwise be in use to reuse for new patterns. RXPSC finds accelerators for 97% of ANMLZoo patterns among the regex family benchmarks, Brill, ClamAV, Dotstar, PowerEN, Protomata and Snort. We examine the network intrusion detection use case in more detail, where RXPSC finds accelerators for 96% of patterns in the registered Snort rules.

We make the following contributions:

- A model for accelerator re-use that is scalable and independent of the underlying accelerator implementation.
- RXPSC, a compiler for finding structural similarities between regexes and generating stateless translators.

Our model is capable of reducing the CPU load of unseen accelerators by more than a factor of ten in 84% of cases across the ANMLZoo benchmark suite.

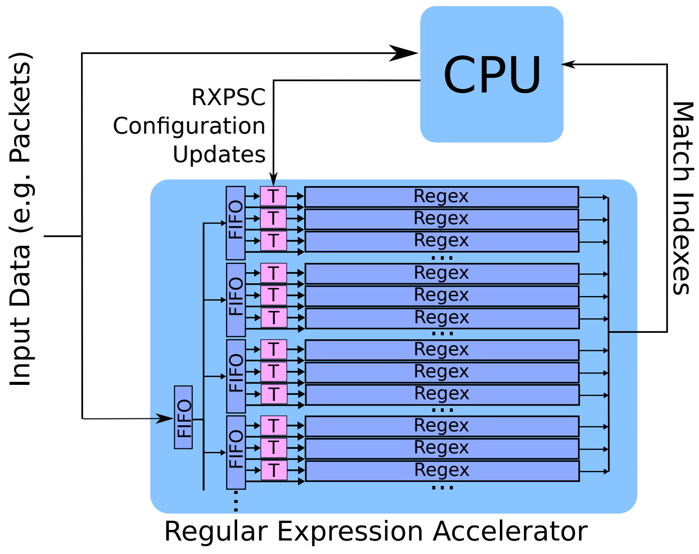
CPU
RXPSC Configuration Updates
Input Data (e.g. Packets)
Match Indexes
FIFO
T
Regex
Regular Expression Accelerator

Fig. 1. Data is read into the accelerator, which sends match indexes to the CPU. Translators (T) are used to change the behavior of accelerators to enable acceleration of new regexes.

`PASS\s*\n`	`PASS\s[^\n]*%[^\n]*%`	`PASS\s[^\n]50`

TABLE I
THREE PREFIX MERGABLE PATTERNS FROM THE SNORT RULESET.

II. Input Stream Translation

The model we present for translating input streams is *stateless-translation*. We use a character lookup that applied to every input character. We present this model because it is easy to implement as a lookup on FPGA, *easy to parallelize* for higher throughput as there are no stateful dependencies, *easy to enable* or disable in a fine-grained manner and *independent of the underlying accelerator* implementation.

A diagram of how this integrates into a Grapefruit [4] accelerator is shown in Figure 1. Input data is streamed into the accelerator, which distributes it between multiple regexes. These report to the CPU when they match. Our translator sits between the regex and the input, and can optionally be enabled for certain types of input. It behaves as a lookup table, translating characters byte-by-byte. The CPU can also read input data, and may have to for further processing of matches. The CPU can be used to inject new patterns on to the accelerator by updating the stateless translators without a full reconfiguration.

A. Motivating Example

Regexes are matching rules used in numerous domains, from network intrusion detection (indicating malicious packets based on text matches) to bioinformatics (indicating genome sequences).

Input	Output	Input	Output	Input	Output
F	U	:	‘ ’	`\n`	`\r`
r	S	`\r`	‘ ’	‘ ’	%
o	E	;	′	<	`\r`
m	R	,	′	"	`\r`

TABLE II
A STATELESS TRANSLATION TABLE TO CONVERT FROM `From: +[^\r\n"<]*[;',]` TO `USER *[^\r]+%'`.

Input	Output	Input	Output	Input	Output
c	a	d	b	*	x

Fig. 2. A stateless translator converting `cd*`to `ab*`. `x` is an arbitrarily selected character to avoid false-positives.

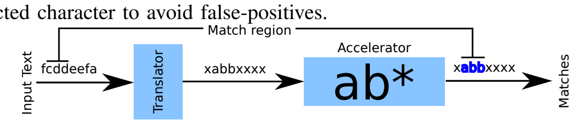

Fig. 3. A stateless translator acting on an input stream.

Regexes include character ranges (`[a-z]`), optional substrings (`a?`) and repeated substrings (`(ab)*`). As an example, the regex `ab*` matches the strings `a, ab, abb, ....`

Regexes are usually implemented using either deterministic (DFAs) or non-deterministic automata (NFAs). Grapefruit implements regexes using NFAs, which take advantage of the parallelism available on an FPGA.

1) Prefix Merging: Prefix merging is a well-known automata compression technique [1]. As an example, suppose we have accelerators for the regexes `abx` and `aby`. These share a common prefix of `ab`, which can be extracted at compile time and accelerated. If we add a regex, `abz`, this common prefix can be used to reduce the computational load of accelerating this third regex without recompiling the FPGA accelerator. A real world example from the Snort rules is shown in Table I.

2) Limits of Prefix Merging: However, prefix merging fails in many cases. Suppose we have an accelerator for the regex `ab*`, and we wish to accelerate the regex `cd*`. Despite significant similarity between the regexes, prefix merging fails to provide even a partial accelerator, as the two regexes do not share a prefix.

Stateless translation can generate the character lookup table shown in Figure 2. If we translate the input stream through the stateless translation table, a match for the regex `ab*` means that the (pre-translation) input stream contained the string `cd*`. Figure 3 shows this translator acting on an input stream. We will explore how our algorithm generates this translation as a running example in Section III.

In general, regexes are more complex, and deriving translators is challenging as stateless translators must satisfy larger sets of constraints. Table II shows one such real-world example of a translator for the Snort rules, again in a case where prefix matching would fail.

III. Compilation Overview

RXPSC takes as input a set of regexes that already have accelerators implementations, and a number of new regexes to be accelerated. It then translates each new regex to a different accelerator, enabling acceleration of the new regex.

A. Groups

RXPSC exploits groupings of regexes that do not have to evaluated at the same time. Groups are application-dependent; for example, network intrusion detection provides groups in the form of protocol, port numbers and IP address ranges. These can be distinguished rapidly in hardware [12], to activate a group of accelerators. RXPSC takes advantage of the otherwise unutilized accelerators.

B. Accelerator Assignment

When adding a regex, RXPSC will calculate the similarity of that regex to *all* of the existing accelerators. Once RXPSC has all the potential translations, it picks the best, operating in a greedy manner. Accelerators may not accelerate the whole regex, so RXPSC identifies accelerators that can accelerate the remainder. This set of accelerators, and corresponding stateless translators, is the output of RXPSC. The core technique of determining similarity is described in the next section.

C. Over Approximation

To find accelerators for a wider range of regexes, RXPSC can find over approximations, meaning that they always accept strings that the original regex would have accepted, but they also accept additional strings. These additional matches can be filtered on the CPU. To distinguish between accelerators that over approximate by various degrees internally, we use an over approximation factor, which is the fraction of edges in the accelerator that are spuriously activated by input symbols. This is a simplification of more complex error representations [13], but allows RXPSC to make informed decisions about which accelerators are likely to be useful. The result of this over approximation is discussed in terms of the extra bytes it involves sending to the CPU in Section V.

IV. Regular Expression Similarity

RXPSC addresses structural similarity and symbol similarity individually. Computing structural similarity generates pairs of corresponding terms (e.g. Figure IV-A2). Symbol similarity unifies these assignments into a stateless translator or rejects the compilation.

A. Structural Similarity

Structural similarity matches a new regex to parts of existing accelerators, while abstracting away any symbol. Symbol similarity (Section IV-B) subsequently determines whether a translator is feasible once symbols are taken into account. We approach structural similarity by compiling regexes to an intermediate language, the accepting path algebra, that abstracts structure from each regex. We then use this algebra to determine which accelerators can structurally support a regex.

1) Accepting Path Algebra: We propose the accepting path algebra, which can be created from regexes. The accepting path algebra enables comparison between different regexes to determine which regexes are similar to each other. The elements of the algebra are:

$n \in \{0, 1\}$	This means n symbols are read from the input.
a	This means there is an accept.
e	Means that this branch of the regex continues no further.
$x + y$	Where x and y are accepting path algebra terms. This means y follows x. It is not commutative. We collapse long sums for readability (e.g. writing $1 + 1 + 1$ as 3), but our algorithms work on unary digits.
$x*$	Where x is an accepting path algebra term. This means that there are loops, one represented by x.
$\{x_0, \ldots, x_n\}$	This means that there is a branch and each arm of the branch is one element within the set. Each of the x_i are accepting path algebra terms.

2) Example: In practice, we generate the accepting path algebra from NFAs. We will use regexes in these examples for simplicity. Below, we show the accepting path algebras for various simple regexes:

Regex	Accepting Path Algebra
`ab*`	$1 + (1) * + a + e$
`cd*`	$1 + (1) * + a + e$
`a(b|c)de`	$1 + \{1, 1\} + 2 + a + e$

3) Determining Structural Support: We describe the algorithm used to determine whether an accelerator can structurally support a different regex. We use the notation $x \leq y$ (read as y structurally supports x) and define the algorithm by cases on the structure of the accepting path algebra. Algorithm 1 shows a simplified boolean version of this algorithm — our implementation keeps track of which terms are related, and which terms are disabled. Given two accepting path algebras A and B, we apply this algorithm with a recursive walk through the algebra structures.

Algorithm 1 Simplified structural support algorithm, defined using terms in the accepting path algebra (Section IV-A1)

```
1: procedure A ≤ B
2:     $e \leq x$: if $x \neq a + ...$ return True                 ▷ (trim)
3:     $a \leq a + x$: return True                               ▷ (dropadd)
4:     $0 \leq y*$: return True                                  ▷ (dropmul)
5:     $m \leq n$ $(m, n \in \mathbb{N})$ return $m == n$        ▷ (inteq)
6:     $a \leq a$: return True                                   ▷ (accepteq)
7:     $e \leq e$: return True                                   ▷ (endeq)
8:     $x* \leq y*$: return $x \leq y$                           ▷ (muleq)
9:     $x_0 + \cdots + x_n \leq y_0 + \cdots + y_m$: return True if:   ▷ (plus)
10:        $\exists i.$ with $x_0 + \cdots + x_i \leq y_0 + \cdots + y_m$
11:        Or, $\exists .i$ with $x_0 + \cdots + x_n \leq y_0 + \cdots + y_i$
12:        Or, $\exists$
13:            XGroup = $[x_0 + \cdots + x_i, x_{i+1} + \cdots + x_j, \ldots]$
14:            YGroup = $[y_0 + \cdots + y_a, y_{a+1} + \cdots y_b, \ldots]$.
15:            Such that: $\forall m.$ XGroup$[m]$ $\leq$ YGroup$[m]$
16:    $\{x_0, \ldots, x_n\} \leq \{y_0, \ldots, y_m\}$: return True if:   ▷ (set)
17:        $\forall x_i . \exists y_j . x_i \leq y_j$
18:    Otherwise: return False
19: end procedure
```

4) Example: Consider the regexes `ab*` and `cd*`. As we saw above, these both have accepting path algebras $1 + (1) * + a + e$. That structural support exists here is clear since the algebras are the same, and is shown in Figure 4. A non-trivial example of structural support is shown in Figure 5, showing that the regex `h(i|jk|l)m*` structurally supports the regex `a(b|c)`.

Structural support is a necessary, but not sufficient, condition for the existence of a stateless translator. Finding an

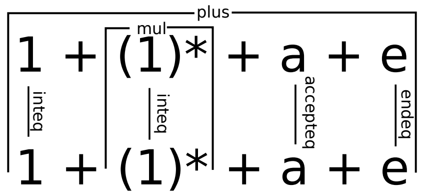

Fig. 4. Rules from Algorithm 1 applied to show how the accepting path algebra for `ab*` structurally supports the algebra for `cd*`.

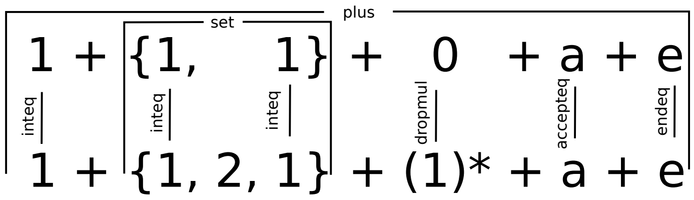

Fig. 5. Rules from Algorithm 1 applied to show how the accepting path algebra for `h(i|jk|l)m*` structurally supports the algebra for `a(b|c)`.

accelerator that structurally supports a new regex shows that acceleration is plausible; but to be useful, we must have sufficient symbol set similarity.

B. Symbol Similarity

Symbol similarity receives, as input, pairs of symbol sets (one virtual symbol set and one accelerator symbol set per algebra term) and produces, as output, a stateless translation. Symbol similarity operates in two steps: symbol completeness then symbol correctness. **Symbol completeness** means our translator does not miss any regexes, and **symbol correctness** means our translator does not accept any patterns that should not be accepted. These steps can fail, in which case we either reject the translation, or accept it as an over approximation.

C. Terminology

Symbol Set A range of symbols, e.g. {`a`} or {`[0-9]`}.

Symbol Set Pairs The pairings produced by the structural support algorithm, for example in Figure 4.

Virtual Symbols Symbols in the regex to accelerate.

Physical Symbols Symbols in the existing accelerator.

Disable Symbols Symbols that activate terms in the accelerator that should not be activated. For example, the term `jk` in Figure 5.

Algorithm 2 Algorithm for generating symbol complete translators.

```
1: procedure SymbolComplete(SymbolSetPairs)
2:     ∀x. Translator(x) = U                        ▷ Universal set
3:     for (VirtSymbs, AccSymbs) in SymbolSetPairs do
4:         for Each symbol in VirtSymbs, v do
5:             Translator(v) = Translator(v)∩AccSymbs
6:         end for
7:     end for
8: end procedure
```

1) Symbol Complete Translator: Symbol completeness (Algorithm 2) begins with the pairwise assignment of accepting path algebra terms produced by the structural support algorithm. It produces a translator that accepts all strings the added regex accepts. The output of this step is passed to symbol-correctness to produce a stateless translator.

Algorithm 3 Algorithm for generating correct translators from a regex R to an accelerator A. X^c denotes the set compliment.

```
1: procedure SymbolCorrect(CompleteTranslator, ToDisable)
2:     ∀x. ActiveSet(x) = {T | T is a term in A with x in its symbol set}
3:     ∀x. MustBeActive(x) = {T | T is a term in A paired to T′ in R where x is in the symbol set of T′ }
4:     for Each Symbol, x do
5:         ApproxSymbs = Active(x)∩MustBeActive(x)^c
6:         Trans(x) = CompleteTrans ∩ ApproxSymbs^c
7:     end for
8:     InactiveTerms = {x | ActiveSet(x) = ∅}
9:     ToDisable = ToDisable ∩ InactiveTerms
10:    if ToDisable = ∅, fail.
11:    if Translator(x) = ∅ for any x, fail
12:    Select an arbitrary element from each Translator(x).
13: end procedure
```

2) Symbol-Correct Translation: Symbol-correctness (Algorithm 3) begins with a symbol-complete translator. The process for arriving at a symbol-correct translator ensures that terms are not spuriously activated in the accelerator, which would cause acceptance of strings that should be rejected. For example, suppose we have the accelerator `ab*` and we wish to accelerate the regex `cd*`. We ensure that `a` and `b` are translated to characters that are neither `a` or `b` to avoid incorrectly accepting strings such as `ab, abb, ...`. We often omit or partially execute the correctness phase of stateless translator generation. This results in accelerators that over approximate and leave some extra computation for the CPU. We discuss this over approximation more in Section III-C.

V. Evaluation

We examine RXPSC's performance on the ANMLZoo [1] benchmarks in the regex family: Brill, Snort, Protomata, PowerEN, Dotstar and ClamAV. We also consider a network intrusion detection setting.

A. Setup

Experiments are run by taking each benchmark set, and removing a regexes. The remaining regexes are compiled. We then add the new regex, by computing whether some prefix of the new regex can be represented using the other regexes. In each experiment, we generate simulators for each converted regex and run the 1 MB ANMLZoo inputs into each simulator. We compare these to the baseline (unmodified) accelerator to compute the over approximation rate — the rate at which the accelerator generates spurious accepts. Using the average

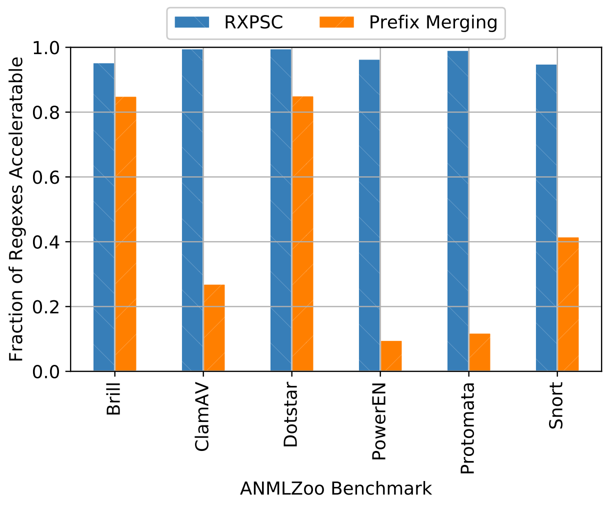

Fig. 6. Fraction of patterns that can run on the accelerators for the other patterns in each benchmark.

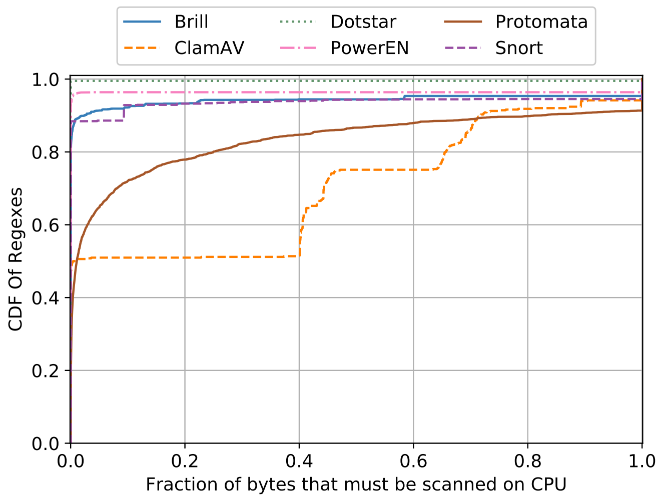

Fig. 7. Fraction of bytes executed on the CPU using RXPSC.

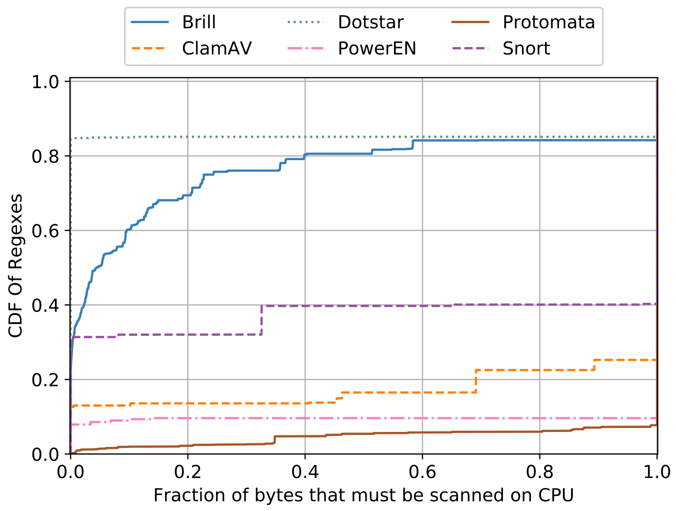

Fig. 8. Fraction of bytes executed on the CPU using prefix merging.

accepting length as a proxy for the number of bytes that the CPU must then check, we can compute the fraction reduction in bytes that the CPU must scan. We run these experiments for every regex within each benchmark (~2,500 regexes each).

RXPSC is particularly useful within a network intrusion detection setting as it provides easy-to-form groups of regexes and requires pattern updates without excessive compile time. We consider Snort [14], a set of network intrusion detection rules from Cisco and look at two different rule sets, the unregistered rule set (1,497 regexes) We split rules by their protocol, port numbers and IP addresses. This results in a number of different groups of regexes, that can each be run on different packets and creates a less general, but more realistic, situation than that used for ANMLZoo.

For the Snort experiment, we remove one *class* of regexes from the set of all regexes. A class of regexes is some set of regexes that must be run for a particular protocol/port/IP address combination. We then compile each of these regexes to the accelerators presented by the remaining regexes (of different classes).

B. Results

Figure 6 shows what fraction of unseen regexes RXPSC and prefix merging can find *any* accelerator for. We can see that RXPSC finds accelerators for 97% of supplied regexes on average, performing particularly well on ClamAV, Dotstar and PowerEN in this metric, where RXPSC finds accelerators for more than 99% of regexes. In contrast, prefix merging only finds accelerators for 43% of regexes.

As discussed above, finding an accelerator does not tell the full story, as for some regexes RXPSC only achieves partial offload from the CPU. We show the number of additional bytes that must be scanned on a CPU as a fraction of the total data in Figure 7 for RXPSC, and Figure 8 for prefix merging.

We see that for some benchmarks, PowerEN, Dotstar, Snort, and Brill, RXPSC is able to almost entirely remove the need for the CPU with new patterns. For others, ClamAV and Protomata, we see that the generated accelerators are only able to reduce the CPU in a smaller fraction of cases, reducing CPU by more than a factor of ten in 51% and 72% of cases respectively. We can also see that RXPSC outperforms prefix merging by a significant margin on all benchmarks, where only Brill and Dotstar can be run without complete reliance on the CPU for more than 50% of regexes.

RXPSC generates assignments reducing the quantity of data that must be scanned by the CPU by more than a factor of ten in 84% of cases, performing particularly well on Dotstar, PowerEN and Snort where RXPSC achieves this benchmark in 99.5%, 96% and 93% of cases respectively. Prefix merging reaches this threshold for 85%, 9% and 32% respectively.

The differences between the accelerator generation graph (Figure 6) and the bytes requiring CPU graph (Figure 7) are down to two key features: first, we do not require an entire regex match; and second, translators often over approximate. For some benchmarks, such as ClamAV, regexes often begin with large series of single-character symbol sets (e.g.

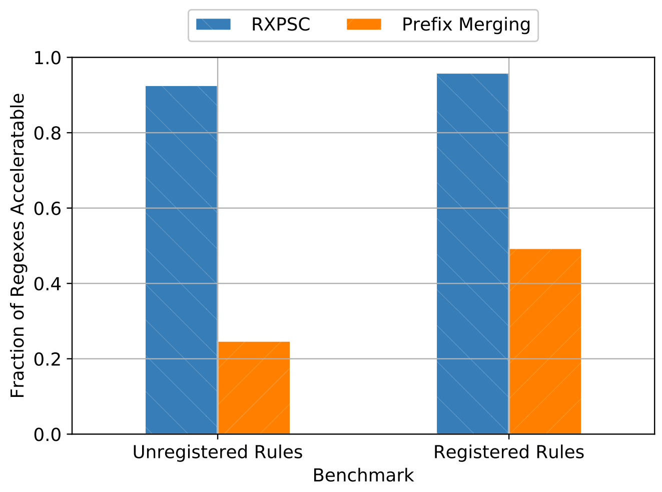

Fig. 9. Fraction of regexes that can be supported by existing accelerators.

	REAPR [9]	Optimized REAPR [9]	RXPSC	Prefix Merging
Brill	21168 s	15864 s	1.8 s	0.2 s
ClamAV	17100 s	13020 s	0.9 s	0.1 s
Dotstar	Unreported	Unreported	0.5 s	0.1 s
PowerEN	Unreported	Unreported	0.7 s	0.1 s
Protomata	23388 s	17130 s	12.4 s	0.1 s
Snort	25020 s	25020 s	0.9 s	0.03 s

Fig. 10. Time required to add additional regexes. We compare to REAPR [15], a prior version of Grapefruit, as numbers for Grapefruit are unreported.

`\x00\x00\x00`). RXPSC is capable of finding accelerator matches for these, which under our own heuristics perform well, but these do not distinguish test data significantly. Similarly, for Protomata, a very reduced dictionary of 16 characters is used, again resulting in poor choices of accelerator to use. We expect both benchmarks could see improved performance with more appropriate heuristics for their particularities.

1) Network Intrusion Detection: Figure 9 shows that 96% of regexes can be accelerated using existing accelerators in the registered rules and 93% of regexes can be accelerated in the unregistered rules. The difference is down to the number of rules — the larger set of rules provides more accelerators to choose from. Prefix merging finds alternatives for only 49% and 25% of each set of rules, again showing the benefit of having more accelerators to choose from. In this real-world situation where we must load a new pattern onto an accelerator quickly, RXPSC does so in the vast majority of cases, reducing the volume of data the CPU must process, and freeing CPU cycles.

C. Compile Time

RXPSC is capable of compiling new regexes to existing accelerators in a number of seconds. In this experiment, we explore compile times for additional regexes using RXPSC, and compare them to the reported compile times for the ANMLZoo benchmark suite [9] on REAPR [15] (a prior version of Grapefruit), showing both the default compile times and compile times improved with compilation toolchain optimizations. Figure 10. RXPSC's compile times that are almost all a factor of 10,000 less than those reported for REAPR.

VI. Conclusion

We present RXPSC, a regex compilation tool capable of compiling new regexes to existing accelerators. RXPSC finds structural similarity between regexes and generates stateless translators that allow in-place accelerator updates without recompiling an FPGA accelerator. We find that we can reduce CPU workload by more than a factor of ten for 84% of unseen regexes across the ANMLZoo benchmarks, outperforming prefix merging, which only reaches this benchmark for 34% of unseen regexes in ANMLZoo. We demonstrate a use case in network intrusion detection, where new rules must be implemented quickly in response to new threats, and again show that RXPSC achieves significant improvement over prefix merging.

References

[1] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron, "ANMLZoo: A benchmark suite for exploring bottlenecks in automata processing engines and architectures," *IISWC*, 2016.
[2] Grovf, "GRegeX." https://grovf.com/products/gregex.
[3] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE PDS*, vol. 25, pp. 3088–3098, 12 2014.
[4] R. Rahimi, E. Sadredini, M. Stan, and K. Skadron, "Grapefruit: An open-source, full-stack, and customizable automata processing on FPGAs," *FCCM*, 2020.
[5] T. Nowatzki, V. Gangadhar, K. Sankaralingam, and G. Wright, "Domain specialization is generally unnecessary for accelerators," *IEEE Micro*, vol. 37, pp. 40–50, 2017.
[6] T. Jepsen, D. Alvarez, N. Foster, C. Kim, J. Lee, M. Moshref, and R. Soulé, "Fast string searching on PISA," in *SOSR*, ACM, 4 2019.
[7] M. Ceska, V. Havlena, L. Holík, J. Korenek, O. Lengál, D. Matousek, J. Matousek, J. Semric, and T. Vojnar, "Deep packet inspection in FPGAs via approximate nondeterministic automata," *CoRR*, 2019.
[8] C. Xu, S. Chen, J. Su, S. M. Yiu, and L. C. K. Hui, "A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms," *IEEE Communications Surveys & Tutorials*, vol. 18, pp. 2991–3029, 2016.
[9] C. Bo, *Automata Processing: from Application Acceleration to Hardware Design*. PhD thesis, University of Virginia, 2019. Chapter 6.
[10] C. Bo, V. Dang, T. Xie, J. Wadden, M. Stan, and K. Skadron, "Automata processing in reconfigurable architectures," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 12, pp. 1–25, 6 2019.
[11] R. Karakchi, C. Daniels, and J. Bakos, "An overlay architecture for pattern matching," in *ASAP*, IEEE, 7 2019.
[12] M. Attig and G. Brebner, "400 Gb/s programmable packet parsing on a single FPGA," in *ANCS*, IEEE, 2011.
[13] A. H. N. Sabet, J. Qiu, Z. Zhao, and S. Krishnamoorthy, "Reliability analysis for unreliable FSM computations," *ACM Transactions on Architecture and Code Optimization*, vol. 17, pp. 1–23, 5 2020.
[14] The Snort Project, "SNORT users manual: 2.9.16." http://manual-snort-org.s3-website-us-east-1.amazonaws.com/snort_manual.html, 2020.
[15] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan, "REAPR: Reconfigurable engine for automata processing," in *FPL*, IEEE, 9 2017.