Journal of Network and Computer Applications 208 (2022) 103507

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Reconfigurable regular expression matching architecture for real-time pattern update and payload inspection

Jaehyun Nam [a], Seung Ho Na [b], Seungwon Shin [b], Taejune Park [c,*]

[a] Dankook University, 152, Jukjeon-ro, Suji-gu, Yongin-si, Gyeonggi-do, 16890, Republic of Korea
[b] KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon, 34141, Republic of Korea
[c] Chonnam National University, 77 Yongbong-ro, Buk-gu, Gwangju 61186, Republic of Korea

ARTICLE INFO

Keywords:
Deep packet inspection (DPI)
Regular expression (regex)
Real-time pattern update
Pattern matching
FPGA hardware

ABSTRACT

Regular expression (regex) matching is an integral part of deep packet inspection (DPI), but its efficiency becomes a question due to low performance. For regex matching (REM) acceleration, FPGA-based solutions have emerged to maximize parallelism by processing multiple regex patterns concurrently. However, even though they significantly accelerate the performance, they have a critical problem that they do not support dynamic regex pattern updates in run time, which is the key functionality along with frequently altered signatures to cover newly identified vulnerabilities. Hence, we present Reinhardt, a new reconfigurable hardware architecture for REM. Reinhardt introduces new FPGA blocks, called reconfigurable cells, that form regex patterns in hardware, enabling real-time regex pattern update and match in run time while providing high performance. With the prototype of Reinhardt on NetFPGA-SUME, our evaluation shows that Reinhardt updates hundreds of regex patterns within a second and performs REM at up to 10 Gbps throughput (max. hardware bandwidth) with the constant latency. Our case studies also show that Reinhardt can operate in multiple modes (e.g., as a standalone NIDS/NIPS or as the REM accelerator for them).

1. Introduction

As network traffic has become sophisticated with time, payload analysis has also become an essential operation in network protection. In that sense, deep packet inspection (DPI), which analyzes packet payloads, plays a central role in network intrusion and prevention systems (NIDS/IPS) (Snort, 2021; Suricata, 2021; Zeek (Bro), 2021). However, while DPI in modern networks should satisfy *high performance* and *dynamic updatability* to deal with a large amount of traffic and rapidly changing networks (Xu et al., 2016; Van Lunteren, 2006; Atasu et al., 2013; AbuHmed et al., 2008; Tupakula et al., 2011; Fernandes et al., 2019), its key functionality, regular expression matching (REM), is considered the major bottleneck. Thus, prior researchers have attempted to accelerate the performance of REM using hardware, mainly based on Field-Programmable Gate Arrays (FPGA), by matching multiple regex patterns in parallel (Sidhu and Prasanna, 2001; Hieu et al., 2013; Hutchings et al., 2002; Sourdis et al., 2008; Lin et al., 2007; Yang et al., 2008; Mitra et al., 2007).

However, even though FPGA-based REM significantly accelerates the overall performance, it also raises three critical problems due to the lack of dynamic updatability. First, *updating regex patterns* in FPGA takes a significant amount of time (e.g., at least a few hours) to compile the patterns into hardware circuits (synthesis, map, placement, and routing). Second, *service interruption* is required for the initialization to apply newly compiled patterns into FPGA, exposing a network in an unprotected state for a while. Third, the update has to perform in an *all-or-nothing* fashion, which means that even a tiny pattern change requires the entire compilation process and service interruption. For these reasons, the hardware-based REM solutions have been less adopted for NIDS/IPS so far. Also, while these problems in FPGA-based REM have been pointed out for years, they still remain as significant but unsolved limitations (Kumar, 2007; Vasiliadis et al., 2008; Chen et al., 2010; Xu et al., 2016).

To achieve dynamic updatability in FPGA-based REM, we propose Reinhardt, a new reconfigurable hardware architecture for real-time regex pattern update and match in run time. The Reinhardt design sits on the opposite spectrum of prior FPGA-based REM (from a circuit level to a logic level) to better support dynamic updatability. For this, Reinhardt introduces new FPGA blocks, called reconfigurable cells, that can change their connections to neighbor cells in run time. In this work, reconfigurable cells represent regex patterns in hardware. The combination of the cells implements Finite-State Machines (FSM) for given regex patterns using our conversion algorithm and is deployed

* Corresponding author.
E-mail addresses: jaehyun.nam@dankook.ac.kr (J. Nam), harry.na@kaist.ac.kr (S.H. Na), claude@kaist.ac.kr (S. Shin), taejune.park@jnu.ac.kr (T. Park).

https://doi.org/10.1016/j.jnca.2022.103507
Received 20 March 2022; Received in revised form 8 August 2022; Accepted 23 August 2022
Available online 21 September 2022

into hardware. Indeed, Reinhardt enables dynamic pattern updates instantly without service interruptions while maximizing the natural parallelism of hardware. Moreover, Reinhardt maintains all the information of cell connections in FPGA memory; thus, Reinhardt can process a large number of regex patterns that even exceed the physical space of Reinhardt by dynamically fetching them (swapping pattern sets in processing), which means that it allows a packet to be inspected with multiple patterns continuously (called resubmitting). Lastly, Reinhardt provides application programming interfaces (APIs) for better applicability, allowing existing NIDS/IPS solutions to adopt Reinhardt for REM acceleration as well.

We implement a Reinhardt prototype using NetFPGA-SUME (NetFPGA, 2014; Zilberman et al., 2014). Our evaluation shows that Reinhardt can update 1300 patterns in 0.96 s with zero downtime, which is overwhelmingly faster than prior solutions (1–5 h). Reinhardt shows 1.4–10 Gbps throughput with 800–160 regex patterns, respectively. It also performs with the constant latency (2 μs) no matter how many patterns are installed, which means that Reinhardt enables deterministic processing. In fact, Reinhardt has competitive benefits in providing stable performance, compared with DPDK-Hyperscan (Intel, 2021; Wang et al., 2019). Our case studies demonstrate the unique strengths of Reinhardt in REM. In particular, Reinhardt NIDS/IPS covers 87% of signatures in 2.9.7 default rules (6411 signatures), and the hardware acceleration using Reinhardt improves the overall throughput up to 65 times compared to the original performance of the vanilla Snort IDS.

In sum, this paper makes the following contributions.

- **Reconfigurable REM architecture:** We introduce Reinhardt, a novel reconfigurable REM architecture, which directly implements given regex patterns as the state machine logic with the combinations of the reconfigurable cells without service interruption.
- **Prototype and evaluation:** We implement a prototype of Reinhardt on the FPGA hardware and our evaluation shows that Reinhardt can update regex patterns within a second and reach 10 Gbps throughput with a negligible latency overhead.
- **Practical Deployment:** We present the practical deployments of Reinhardt: Reinhardt as an NIDS/NIPS, REM acceleration in Snort IDS, and SDN integration, showing that it can replace today's NIDS/NIPS and accelerate the performance of Snort IDS up to 65 times.

The rest of the paper is organized as follows. Section 2 provides the background and motivation for FPGA-based REM. Sections 3 and 4 describe the design and implementation of the Reinhardt architecture. Sections 5 and 6 summarize the results of our performance evaluation and practical use cases. Section 7 relates Reinhardt to prior work and Section 8 provides a conclusion.

2. Background and motivation

In this section, we provide the background of regular expression matching (REM) and explain the performance degradation issue in DPI due to REM. Then, we discuss the challenges of previous efforts for accelerating the REM performance by using FPGA.

2.1. Regular Expression Matching (REM)

Regular expressions (regex) are helpful to structurize a string that contains a set of specific patterns (e.g., attack signatures) with *metacharacters*, which have special meanings as described in Table 1. Hence, regular expression matching (REM) has been widely adopted in deep packet inspection (DPI) to search for one or more matches of specific patterns in an observation string (e.g., packet payload), enabling network intrusion detection and prevention systems (NIDS/IPS) to detect a variety of network attacks (Kreibich et al., 2001).

Table 1
Common metacharacters for regular expressions.

Syntax	Description
.	Match any character
[ ]	Match a single character contained in brackets
^	Match the starting position in a string
[^]	Match a single character not contained in brackets
*	Match the ending position in a string
+	Match the preceding element one or more times
?	Match the preceding element zero or one time
{*m*, *n*}	Match the preceding element from *m* to *n* times
|	Match an expression before or after an operator
( )	Defines a marked subexpression

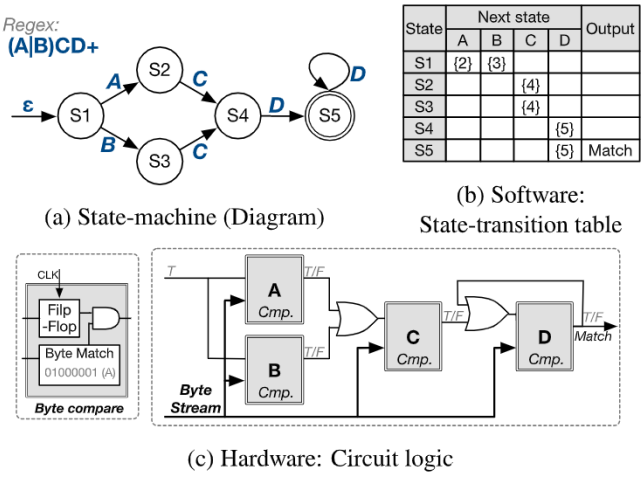


Fig. 1. Implementations for Regex '(A|B)CD+'.

REM begins by generating an equivalent finite-state machine (FSM) for a given regex pattern and drives the state machine on an observation string. For example, a regex pattern '(A|B)CD+', which means {ACD, BCD, ACDD, BCDDD, . . . }, is first converted to a state machine as shown in Fig. 1(a). If a character for each state is sequentially given from a payload (i.e., A or B→C→D→D...), the state machine transits active states from S1 to S4, and eventually reaches S5 (the final state), which is called 'Accept', indicating that the state machine has found a matched pattern from the payload.

2.2. Performance degradation in REM

In general, finite-state machines for regex patterns are implemented into a state-transition table as shown in Fig. 1(b). Then, payload inspection is done by inquiring state transitions over its state table on given payloads byte by byte. Unfortunately, it is time-consuming and memory-intensive to traverse an FSM because it has to continuously read each byte of given payloads and keep following the state transitions over the state graph. In addition, it can get much worse if the complexity and number of regex patterns increase, leading to significant performance degradation.

Here, we conduct a microbenchmark using the PCRE engine (Hazel, 2005) of Snort IDS 2.9.7 (Snort, 2021), which is one of the most popular open-source DPI engines, to show how REM severely impacts the performance. This evaluation is performed on Intel Xeon E5-2630, and the input traffic is a burst of 64-to-1514-byte packets generated by Intel DPDK-Pktgen (Intel, 2021). Also, we randomly select two groups of regex patterns (specified in the pcre option) from the default Snort ruleset, and those regex patterns contain 1.6 (noted as Simple) and 7.6 (noted as Complex) metacharacters on average.

Table 2 presents the performance variations with different sizes of packets and Fig. 2 illustrates the performance variations with the

Table 2
PCRE throughput measurements with different-size of packets (Simple and complex rules contain 1.6 and 7.6 metacharacters on average, and the above throughputs are measured in Mbps.).

# of rules	64B	128B	512B	1024B	1514B
No rule	453.5	1280.5	3440.5	5723.0	7650.5
1/Simple	344.7	1024.4	2776.2	5086.8	7236.0
1/Complex	344.7	1024.4	2772.9	5089.7	6855.3
50/Simple	113.4	128.1	174.8	195.4	153.4
50/Complex	45.4	64.0	81.7	88.9	67.2
100/Simple	56.7	67.4	75.3	80.9	49.0
100/Complex	34.9	44.2	61.4	70.0	0.6

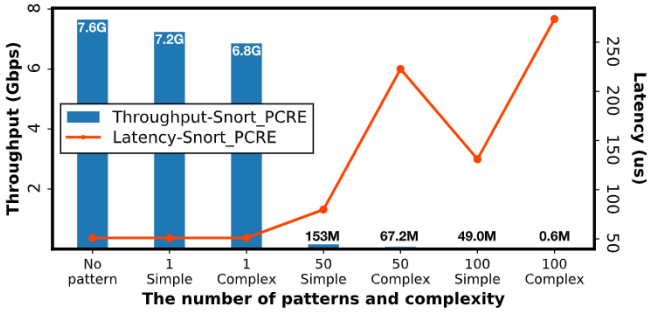


Fig. 2. PCRE throughput and latency measurements with the increasing number of regex patterns (Simple and complex rules contain 1.6 and 7.6 metacharacters on average.).

increasing number of the patterns. Compared to the baseline (from 0.45 to 7.65 Gbps), we see noticeable throughput degradation (31.5% for 64-byte packets and 5.7% for 1514-byte packets) with both simple and complex patterns. However, as the number of regex patterns increases, the overall throughputs are dramatically dropped; with 50 simple and complex patterns, the throughputs become 153 and 67 Mbps, respectively, dropping up to 97.9% and 99.1% compared to the baseline (no pattern). With 100 patterns, the throughputs are no longer viable. Similarly, the latency also increases from 50 μs to 273 μs, 5.5 times higher than the baseline with the increasing number and complexity of the patterns.

The performance degradation mostly happens due to frequent state transitions along input strings (packet payloads) and metacharacter operations, incurring heavy memory accesses. In addition, as the number of patterns increases, the overhead caused by the state transition becomes serious. In sum, REM is a major bottleneck point in DPI and should be improved for practical deployments.

2.3. Hardware-based REM acceleration

To improve the performance of REM, prior studies have proposed hardware-based acceleration using Graphics Processing Units (GPU) (Jamshed et al., 2012; Vasiliadis et al., 2008; Zu et al., 2012; Yu and Becchi, 2013; Smith et al., 2009) and FPGA (Sidhu and Prasanna, 2001; Hutchings et al., 2002; Sourdis et al., 2008; Lin et al., 2007; Yang et al., 2008; Mitra et al., 2007; Hieu et al., 2013).[1]

GPU-based studies utilize a large number of GPU cores to process multiple regex patterns and improve the performance of REM. However, they require multiple copies of packet payloads (from a network interface to CPU and from CPU to GPU). Thus, GPU-based solutions transfer incoming packet payloads to GPU in batch to minimize the overhead, which means that GPU-based solutions are not suitable for real-time REM, especially in mission-critical networks.

In contrast, FPGA-based solutions situate the state machines of regex patterns at a circuit level (no memory access) as depicted in Fig. 1(c), maximize the natural parallelism of hardware, and directly connect to network interfaces (no data transfer delay due to bump-in-the-wire). Thus, they enable REM in a constant time regardless of the number of patterns, which means that they can guarantee *deterministic performance* (Thomas et al., 2009; Che et al., 2008; Cope et al., 2010; Fowers et al., 2012). FPGA-based solutions are more widely used for mission critical networks such as latency-sensitive networks than GPU-based solutions (Firestone et al., 2018; Gupta, 2016; Putnam et al., 2014; Li et al., 2016). For this reason, in this work, we will focus on FPGA.

Table 3
Compilation time measurements with prior FPGA-based REM.

Regex engine	# of patterns	Time (H:M:S)
Sourdis et al. (2008)	1504	4:53:50
	310	0:45:49
Bispo et al. (2007)	310	1:47:00
Johnson and Mackenzie (2001)	200	1:38:57
Ganegedara et al. (2010)	760	1:52:00

2.4. Challenges in FPGA-based REM

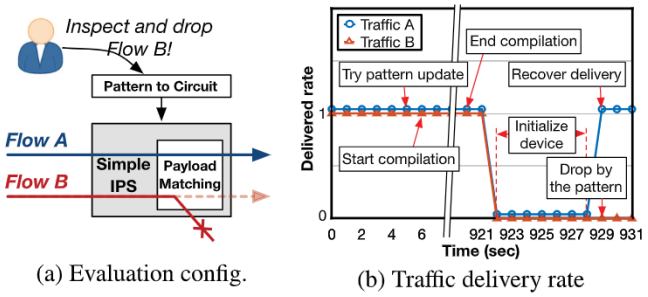
FPGA-based solutions mostly adopt non-deterministic finite automata (NFA) rather than deterministic finite automata (DFA) (Sidhu and Prasanna, 2001; Xu et al., 2016) for REM because (1) parallel processing in hardware allows concurrent access to multiple states, easily handling non-deterministic states, and (2) DFA requires much larger space than NFA while the space problem is one of the sensitive issues in FPGA due to its limited resource. However, although FPGA-based approaches can definitely improve the performance of REM in DPI, in reality, people often hesitate to adopt them due to the *limited flexibility* of FPGA. Here, we discuss three critical challenges of FPGA-based REM solutions from the pattern update perspective.

(1) Long Compilation Time: Unlike software-based REM, FPGA-based REM requires the compilation process (i.e., synthesis, map, placement, and routing) to update regex patterns at a circuit level. However, it takes at least a couple of hours to conduct this process (Vasiliadis et al., 2008; Lavin et al., 2011, 2013; Love et al., 2013). Also, it can take even a couple of days in the worst case, which means that it is hard to apply certain regex patterns into FPGA hardware immediately. Table 3 shows the magnitude of the long compilation time of existing solutions (Sourdis et al., 2008; Bispo et al., 2007; Johnson and Mackenzie, 2001; Ganegedara et al., 2010). While it is difficult to fairly compare the update times with each other due to different goals, production methods, and the complexity of target regex patterns, we confirm that updating regex patterns is a time-consuming task, allowing malicious users to have enough time to exploit new attacks (e.g., zero-day attacks) to compromise a system until such attack patterns are updated into DPI engines.

(2) Inevitable Service Interruption: After the compilation process, FPGA-based solutions require a system halt due to the initialization with updated regex patterns. It may not take such a long time as much as the compilation process (from several seconds to a half of a minute). However, in terms of serviceability, networks are temporarily exposed to potential threats, which increases an operational burden.

(3) All-or-Nothing Update Operation: Current FPGA-based solutions need to update regex patterns in an all-or-nothing fashion since all regex patterns deployed in FPGA hardware are statically deployed (fixed) at a circuit level (Vasiliadis et al., 2008; Lavin et al., 2011, 2013; Love et al., 2013). For this reason, even a small change in

[1] In this paper, FPGA-based approaches mean circuit-based approaches only, not including memory-based approaches (Baker et al., 2006; Brodie et al., 2006; Tang et al., 2014; Mellanox, 2021) because the memory-based one features sequential processing, it does not fully support massively parallel processing (Bispo et al., 2007).



(a) Evaluation config. (b) Traffic delivery rate

Fig. 3. Regex pattern update and function recovery time measurements of a simple hardware-based IPS.

regex patterns requires the entire compilation and initialization process again. Hence, they tend to enforce regex pattern updates on a regular basis (stacking updates and applying them at once) instead of actively applying updates.

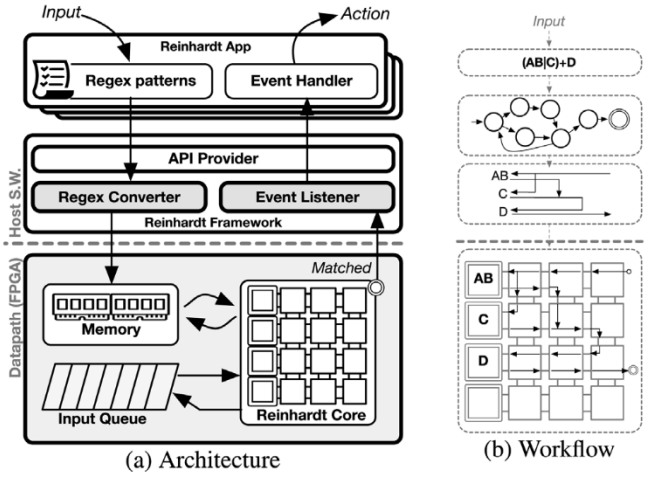
Demonstration in FPGA-based REM: Here, we conduct a simple evaluation to demonstrate those challenges. As illustrated in Fig. 3(a), we generate two traffic Flow A and B toward a simple hardware IPS. Then, we try to install a new signature (regex pattern) that inspects and drops Flow B on the IPS at 5 s and measures the time when Flow B is blocked by the IPS. We use NetFPGA-SUME (NetFPGA, 2014) as the base hardware platform, and the circuit was compiled using Vivado 2016.04 on Xeon E5-2630. Fig. 3(b) shows the result of this evaluation. To compile the *one* pattern only, it takes 15 min in our host. Then, while deploying the newly compiled circuit, the IPS was stopped for about 10 s Finally, all updates are completed after 924 s, and the circuit eventually works by filtering Flow B.

2.5. Need for near real-time rule update in DPI

It had not been a serious concern to update regex patterns in near real-time. We could stop or delay the operation of a hardware/software system for DPI, upload newly compiled patterns to the system, and relaunch the system after the update. This delayed operation had been not a big problem.

However, the rapid increase in malware drives the need for a real-time update in DPI to detect attack patterns (i.e., signatures) in packet payloads. According to the statistics of common vulnerabilities and exposures (CVE) (Mitre, 2021; Detail, 2021), more than 30 new vulnerabilities are registered on their index every day, which means that we should update patterns (signatures) that detect those vulnerabilities immediately to keep security up to date. Indeed, according to the update history of signatures (Emerging Threats, 2021), one or two update(s) occurred every day, and the updates for critical threats sometimes occurred multiple times within hours. For this reason, previous works (Xu et al., 2016; Van Lunteren, 2006; Atasu et al., 2013; AbuHmed et al., 2008; Tupakula et al., 2011; Fernandes et al., 2019) and many eminent security articles (CORSA, 2019; RAPID7, 2017; InfoSecurity, 2019; Symantec, 2002; Check Point Software Technologies Ltd, 2019; FireEye, 2021) address *dynamic updatability* as one of the key concerns in NIDS/IPS.

In addition, virtualized environments such as clouds and 5G core networks have also a significant demand for dynamic update due to their flexibility (Vaquero et al., 2012; Bremler-Barr et al., 2014), and these areas already consider to migrate their facilities to FPGA (Firestone et al., 2018; HPC, 0000; Intel, 0000a; Yu et al., 2011; Fahmy et al., 2015; Ahmadi, 2016; Ricart-Sanchez et al., 2018; Ribeiro and Gameiro, 2017; Intel, 0000b; Chamola et al., 2020) to accelerate the performance of REM in large-scale networked systems. However, it is still conservative to implement DPI using FPGA in environments where patterns should be frequently updated due to the limited updatability. Also, this issue has been a major challenge for years in research on implementing REM with FPGA but still remained unsolved (Kumar, 2007; Vasiliadis et al., 2008; Chen et al., 2010; Xu et al., 2016).



(a) Architecture (b) Workflow

Fig. 4. Overall design of Reinhardt.

3. Design

In this section, we present Reinhardt, a reconfigurable FPGA architecture for REM, and explain how Reinhardt dynamically updates given regex patterns and matches them with the incoming traffic on the FPGA architecture.

3.1. Overview

Fig. 4 shows the design of Reinhardt and its workflow. Reinhardt consists of a software framework on the host side and the Reinhardt datapath in the FPGA hardware. First, the software framework provides APIs that allow operators to inject regex patterns to the datapath and to receive messages (matching events) from the datapath. Then, The Reinhardt datapath processes the regex patterns injected through Reinhardt APIs with the incoming traffic (an observation string). For dynamic pattern update and matching, Reinhardt introduces a new FPGA block, called a *reconfigurable cell*, and the Reinhardt core consists of a set of reconfigurable cells connected in a $w \times h$ grid topology with input/output ports for each direction, *Top-Bottom-Left-Right*. Thus, each cell can dynamically determine the output directions of the input signals according to Reinhardt configurations.

Reinhardt mainly focuses on the dynamic pattern update in real-time without any service interruption. Here, the update includes all procedures required to manage regex patterns, such as (1) deploying new patterns, (2) modifying and removing previously deployed patterns. Hence, Reinhardt stands on the key idea that regex patterns are directly represented in the Reinhardt core as the form of non-deterministic finite-automata (NFA) through a combination of the cell connections. As an example, Figure Fig. 4(b) illustrates how a regex pattern can be converted to the corresponding NFA structures and represented with the reconfigurable cells in the Reinhardt core. The conversion results are stored in the Reinhardt memory, changing the input/output directions of the cells in the core; thus, the equivalent state machines for the regex patterns can be implemented in real-time.

Finally, Reinhardt inspects the incoming traffic (an observation string) by following the NFA logic in the Reinhardt core, and it sends a notification to the event listener on the host side if any matched pattern is found. Then, the event listener notifies Reinhardt applications (specifically, event handler) and the applications will take actions (e.g., blocking suspicious traffic or updating regex patterns to cover newly identified vulnerabilities).

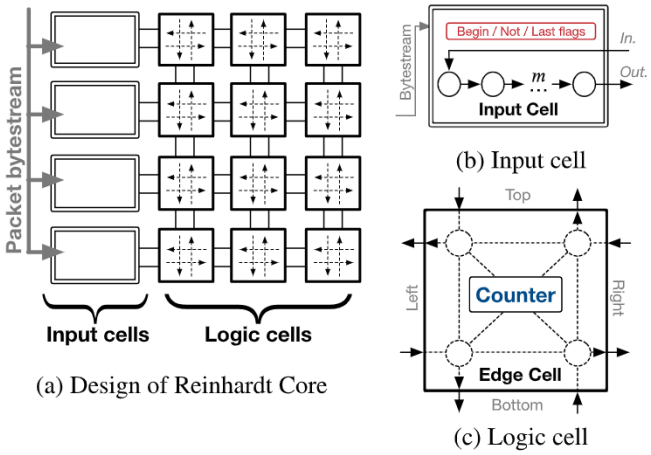


Fig. 5. Design of Reinhardt Core.

3.2. Reinhardt core

Fig. 5 illustrates the overall design of the $w \times h$ Reinhardt core. Reinhardt provides two types of reconfigurable cells: *input cells* and *logic cells*. The first column of the grid is composed of input cells and the rest of the grid is composed of logic cells. Each cell has connections to neighbor cells through input/output ports, and the position of each cell is expressed in (x, y) coordinates.

Input cell (Fig. 5(b)): The input cells represent the states in state machines, and they are activated or transitioned from/to the input/output ports. They internally consist of linear-chained states of length m, which means that one state cell works as a single m string matcher. For example, if input cells are set to match 'http', 'h', 't', 't', and 'p' are assigned to each cell, and the state transition occurs when 'http' is given sequentially in a byte stream. In addition, input cells can compare observation strings on a min–max range basis. For instance, to find a character between 'a' and 'f' (i.e., a bracket expression [a-f]), the min character is set as 'a', and the max character is set as 'f'. The input cell can take flags that indicate the beginning and end of a string and not contain conditions (i.e., ^,$ and [^]) as well.

Logic cell (Fig. 5(c)): The logic cells function as *the directed edges of state machines*. Each cell has input/output ports at each direction, and it routes incoming state transitions to designated directions according to cell configurations. Also, the logic cells have counters to measure how many times state transitions occur in each region, helping logic cells implement interval conditions that have to be met a certain number of times (i.e., {m,n}).

Core I/O: In the Reinhardt core, there are three types of buses: character input bus, ϵ-signal bus, and accept-signal bus. The input cells are connected to the character input bus that delivers each character of an observation string (i.e., the byte stream of an incoming packet) sequentially to all input cells every clock. The logic cells are connected to the ϵ-signal bus and accept-signal bus. The ϵ-signal triggers the initial state transition, and the accept-signal receives the final state transition. Hence, they are connected to the logic cells for the first and last states of state machines.

3.3. Converting regex to Reinhardt cell logic

To deploy regex patterns into Reinhardt, a given regex pattern should be converted to a combination of reconfigurable cells. Here, we explain the overall compilation sequence. For better understanding, Fig. 6 presents the compilation steps for a regex pattern '$(ab|cd)+X\{1,3\}[0-9]^*$'.

(i) State Machine: A regex pattern is first transformed into a typical finite-state machine based on Thompson's algorithm (Thompson,

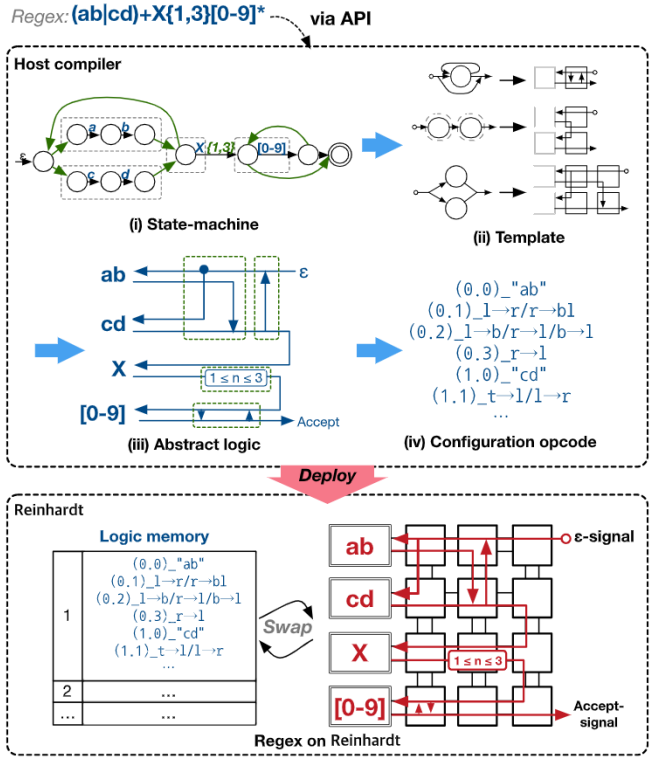


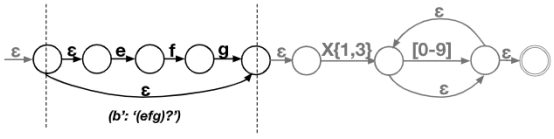
Fig. 6. Compilation and deployment of regex into Reinhardt.

Table 4
Templates for regex pattern to Reinhardt cell logic.

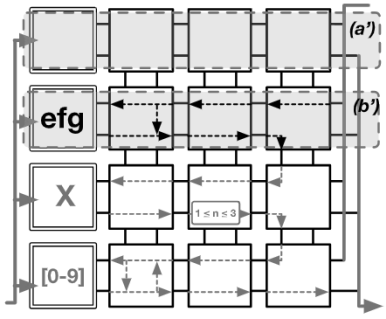
Regex sample	NFA Snippet	Reinhardt logic template
.	any	(Match from null)
[0-9]	0 ~ 9	(Match min/max range)
^A	begin A	(With 'begin' flag)
[^a-z]	a ~ z not	(With 'not' flag)
A$	last A	(With 'last' flag)
A*	A	
A+	A	
A?	A	
A{m,n} (A{m}, A{m,})	m≤A≤n A	(Through a range)
A|B	A B	
(AB)(CD)	AB CD	

Algorithm 1: Regex to Reinhardt Cell Logic

```
Input: regex, Given pattern, start_row, Starting row in the core
row ← start_row
STACK ( in[x, y, d], out[x', y', d'] ) // Stack for IN/OUT
// x, y, d: cell coordinates with its direction
postfix ← Regex_to_postfix (regex)
foreach Character c in postfix do
    switch c do
        case 'Literal' do
            setInput (row, c);
            if c is End of Substring then
                PUSH ( [0, row, 'R'], [0, row, 'R'] );
                row ← row +1;
        case 'Unary_metachar' do
            [in1, out1] ← POP();
            [in', out'] ← setTemplate ( c, [in1, out1]);
            PUSH ([in', out']);
        case 'Binary_metachar' do
            [in2, out2] ← POP(); [in1, out1] ← POP();
            [in', out'] ← setTemplate ( c, [in1, out1], [in2, out2]);
            PUSH ( [in', out']);
[in1, out1] ← POP()
setESignal ( in1)
setAcceptSignal ( out1)
```



(a) NFA for '(efg)?X{1,3}[0-9]*'



(b) NFA Update in the Reinhardt core

Fig. 7. Deployed pattern modification and removal.

1968). The regex pattern is split into sub-expressions, literal characters (ab, cd, X, and [0-9]) and metacharacters (|, +, {1, 3}, and *). Then, the literals are converted to partial states (gray boxes in Fig. 6-(i)), and concatenating them in the relations of the metacharacters completes the state machine (green edges in Fig. 6-(i)).

(ii) Template: Reinhardt provides various kinds of templates as shown in Table 4. With these templates, each sub-expression in the generated state machine is interpreted to one of the templates representing their partial structures into a combination of the cells. The states (i.e., literal characters) are placed into the input cells, and their relations (i.e., metacharacters) are interpreted to the links of the logic cells. For example, the edges corresponding to the alternation operator '|' are replaced to a connection with four logic cells.

(iii) Abstract Logic: Reinhardt formalizes an abstract logic by concatenating the templates and builds up a cell connection topology to be implemented in the Reinhardt core. This abstract logic is completely equivalent to the state machine; the state machine in Fig. 6-(iii) shows the internal representation for metacharacters (i.e., green edges).

(iv) Configuration Opcode: To reflect the abstract logic with the combination of the reconfigurable cells in the Reinhardt core, the logic is compiled into configuration opcodes that control the cells to correspond to the abstract logic. The opcodes instruct reconfigurable cells in (x, y) coordinates and set the behaviors of the cells according to their types. For example, if an 'X' is set to zero (i.e., input cells), its argument needs literal characters for states. Otherwise (i.e., logic cells), the argument needs signal directions (e.g., 't->b', a forwarding signal from the top to the bottom).

(v) Regex Deployment: After all the compilation process is done, the opcodes are stored into the Reinhardt memory. Then, Reinhardt changes cell configurations based on the opcodes in the Reinhardt memory to implement the circuit logic for the state machine of the given pattern.

In-depth Explanation: Algorithm 1 describes the overall compilation and deployment process. It first takes a start row in the Reinhardt core to place a generated NFA and a target regex pattern to deploy and initializes a stack that stores the input/output coordinates and the direction for the last generated partial NFA(s) so far. Then, the given regex pattern is converted to the postfix form to reflect the precedence where the partial NFAs are generated and parsed by reading the postfix sequentially. (1) The sub-strings are placed in the input cell on each row, and whenever the input of one sub-string is completed, the sub-string is considered as one small NFA so that the input/output coordinates of its input cell are stored in the stack. (2) The metacharacter reads the coordinates of the recently generated partial NFAs from the stack by the number of required operands and synthesizes the operand NFA(s) into a bigger NFA with the metacharacter template. Then, the input/output coordinates of the bigger NFA are stored as the new partial NFA. As this process is recursively performed, the entire NFA is completed by connecting the ϵ and accept signals to the last stacked NFA (See Appendix for the detailed steps).

3.4. Modifying and removing deployed patterns

Reinhardt also allows modifying and removing deployed patterns in real-time. When we have a modified pattern, Reinhardt first matches the metacharacters of the original pattern with the modified one and decides what logic cell circuits should be preserved or rewired. This decision is basically made by three steps: (1) expressing each regex pattern as a list of metacharacters, (2) overlapping the two lists with different overlapping sizes and placements, and (3) calculating the Hamming distance (Hamming, 1950) between the two list fragments. Then, Reinhardt finds the overlapping region that has the smallest Hamming distance, and the region will be preserved during pattern modification.

Fig. 7 illustrates how Reinhardt detects the overlapping region that has the smallest hamming distance between '(ab|cd) + X{1,3}[0 − 9]*' and '(efg)?X{1,3}[0 − 9]*' and keeps the logic cells starting from the second row (i.e., *(b′)*) during the pattern modification. For pattern removal, Reinhardt simply initializes the cells occupied by a removed pattern, and this example also includes how Reinhardt removes a deployed regex pattern (see the first row *(a′)* in Fig. 7).

3.5. Reinhardt memory and input queue

The remaining components on the Reinhardt datapath are the Reinhardt memory and input queue, and they assist the Reinhardt core to match an observation string.

Reinhardt Memory: Reinhardt stores multiple NFA logic for the Reinhardt core (i.e., cell configurations) in the Reinhardt memory and fetches the NFA logic to the Reinhardt core dynamically, enabling *seamless* update.

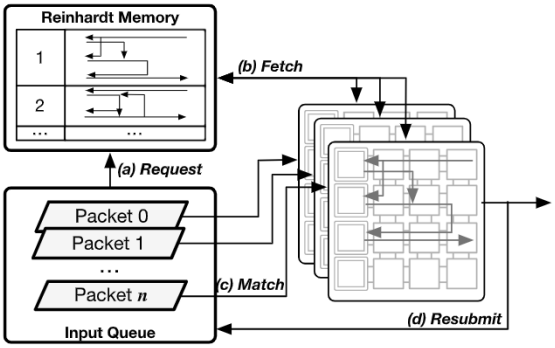


Fig. 8. Reinhardt datapath components and resubmitting.

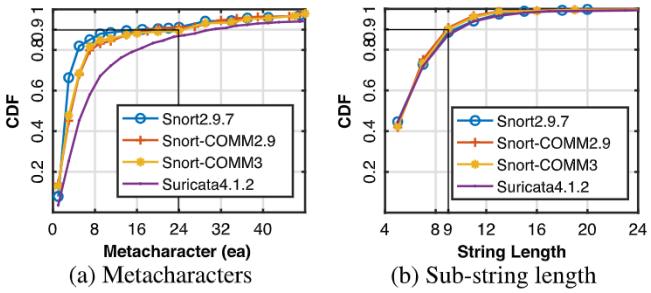


Fig. 9. Frequency statistics of regex patterns.

Reinhardt Input Queues: To match an observation string, the observation string is first enqueued into an input queue, and then each character has to be sequentially entered into the Reinhardt core. This operation is an essential procedure for driving the finite-state machine, but it delays the matches of following observation strings as much as the length of the observation string currently being processed, causing the performance degradation in REM. For this reason, we design the input queue to n input queues and make the Reinhardt core multi-layered, mapping one layer to one queue to maximize parallelism.

When observation strings arrive at the input queue, they are distributed to n input queues in the order of arrival, and all queues keep pushing each character of the observation strings to the Reinhardt core concurrently. Here, the number of the queues n determines the performance of Reinhardt. However, as the number of queues increases, the layers of the Reinhardt core should increase, requiring more FPGA resources as well. Thus, it is important to find the optimal number of queues (we will more cover this in Section 5).

Resubmission: With the Reinhardt memory and input queues, Reinhardt can match an observation string with multiple regex patterns (Park and Shin, 2021) by using Reinhardt's resubmission mechanism, which enables matching a large number of regex patterns beyond the Reinhardt core size.

Fig. 8 shows the steps to resubmit an observation string to match multiple regex patterns. Each regex pattern set in the memory is assigned an ID, and a set contains a list of regex patterns. Step (a) involves the request of an ID set to match. Next, (b) the logic mapped to an ID is fetched and deployed into the Reinhardt core, and (c) the actual matching process with the observation string is done in the Reinhardt core. Then, (d) the observation string is resubmitted into the input queue for the following pattern matching. Then, these steps are repeated until the observation string is matched with all regex patterns in the given set.

3.6. Reinhardt notification events

If any matched pattern is found (i.e., a state transition reaches the accept-signal), the Reinhardt datapath sends a notification message to the event listener in the host. The message contains the row number and active memory ID of the core to indicate which pattern is matched. The received notification is delivered to Reinhardt applications that register its event handler at the event listener. Then, the applications will be able to take actions based on the given message. For example, they can block certain network traffic by enforcing firewall rules or adding more regex patterns to cover newly identified vulnerabilities.

4. Implementation

To validate the efficiency and feasibility of the Reinhardt design, we implemented a prototype using NetFPGA-SUME with Xilinx Virtex-7 XC7V690T and four SFP+ 10 Gbps interfaces (NetFPGA, 2014; Zilberman et al., 2014), which processes packets in chunks of 256-bit at 160 MHz. We also implemented a device driver based on the NetFPGA-SUME reference driver (NetFPGA-SUME, 2020) to create communication channels between the Reinhardt framework and the Reinhardt datapath. Reinhardt APIs internally utilize *ioctl* and *netlink* protocol to update reconfigurable cells.

In terms of the Reinhardt core configuration, there are four constraints to determine the core size: (1) the number of input queues n which is related to the overall throughput, (2) the core width w which determines the complexity of regex patterns, (3) the core height h which indicates the capacity of regex patterns, and (4) the length of input cells m which specifies the maximum length of substrings in regex patterns. While the higher number shows better performance and capacity, we need to carefully determine the constraints of the Reinhardt core within the FPGA hardware limitations.

5. Evaluation

5.1. Reinhardt core constraints

To maximize the performance of Reinhardt, we need to configure its constraints (the width and height of the core, the length of input cells, and the number of input queues). Here, we demonstrate how to determine these constraints.

For this, we collect 2735 regex patterns from Snort 2.9.7 default (648), Snort 2.9 (645) and 3.0 (524) community, and Suricata 4.1.2 default (918) rulesets, and then find the constraints that can express 90% of regex forms and accommodate as many patterns as possible with Reinhardt.

Core width: The width of the core (w) determines how many metacharacters in a single regex pattern Reinhardt can support. Thus, we examine the number of metacharacters in the collected regex patterns, and Fig. 9(a) shows the frequency distribution of the regex patterns. Based on the result, we see that Reinhardt can cover 90% of regex patterns regardless of the ruleset choice when $w = 24$.

Length of input cells: The length of the input cell (m) specifies the maximum length of a substring in a single regex pattern. If the defined length is insufficient to cover each substring, the substrings are concatenated into multiple cells, wasting the core space. To find the optimal length, we examine the distribution of the lengths of substrings in the collected regex patterns. Then, as shown in Fig. 9(b), 90% of the substrings have up to 9 characters (i.e., $m = 9$).

Core height and the number of queues: While the above-mentioned constraints are determined based on the given regex patterns, the height of the Reinhardt core h and the number of queues n are dependent on how many resources (i.e., LUTs) in FPGA are utilized. Thus, we implement *four* Reinhardt datapaths with different core sizes: $24 \times 160 \times 8, 24 \times 300 \times 4, 24 \times 665 \times 2$ *and* $24 \times 1580 \times 1$ (Width w × Height h × Queues n) with $m = 9$ with about 90% of the resources in NetFPGA-SUME. We will address which combination leads to the optimal performance of Reinhardt in the evaluations.

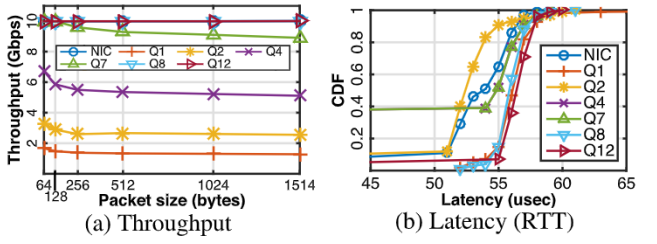


Fig. 10. Throughput and latency variations with the different number of input queues (NIC indicates the baseline without Reinhardt).

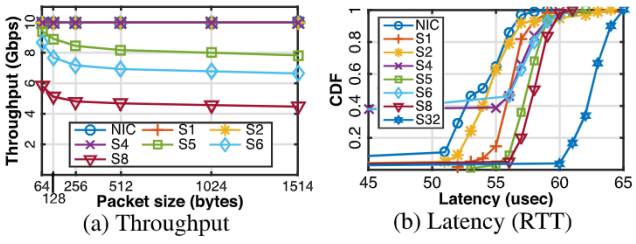


Fig. 11. Performance degradation due to resubmitting.

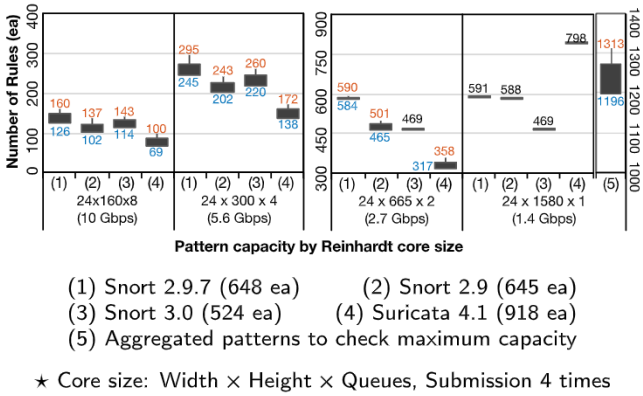


Fig. 12. Pattern capacities with the different core sizes.

5.2. Throughput and latency measurements

We measure the performance variations of Reinhardt to see how many input queues are required to get the line-rate performance (i.e., 10 Gbps) and to see how much overhead Reinhardt impacts on its performance due to the resubmission. For these measurements, we use three machines with an Intel Xeon E5-2630 CPU, 64 GB, and Intel X520 10GbE NICs. We install a NetFPGA-SUME FPGA board on one of them, and the others are used for traffic generation with Intel DPDK-Pktgen (Intel, 2021) and Nping (Nping, 2021). For comparison, we also measure the throughput and the latency between the sender and the receiver by directly wiring them without Reinhardt.

The number of input queues: This required number of queues to achieve 10 Gbps can be first proved arithmetically. We implement Reinhardt to process a packet in chunks of 256-bit (32 characters), resulting in the delay of 32 clocks per chunk, and this delay is always constant regardless of the Reinhardt core size. Since the clock rate of NetFPGA-SUME is 160 MHz, 1 clock takes 6.25 ns; thus, the delay of 32 clocks takes 200 ns. For processing 256-bit chunks at 10 Gbps speed, each chunk must be processed within 25.6 ns. As a result, the required number of queues can be calculated through $200/25.6 = 7.81$ (i.e., 8). Then, we measure the throughput and latency variations with the physical machines. Fig. 10(a) shows the throughput variations of Reinhardt with the different number of input queues. The throughput with a single queue varies from 1.68 to 1.28 Gbps. As the number of the queues increases, the overall throughput increases, achieving the line-rate throughput (10 Gbps) with *eight* queues and above.

One peculiar point is that the throughput of Reinhardt slightly decreases as the packet size increases. This is because Reinhardt operates the queues in a store-and-forward manner and inspects each packet byte-by-byte. Therefore, while small packets are processed with a shorter queue occupancy time and released from the queue earlier, a larger packet should consume a longer processing time and should be held in a queue longer. As a result, overruns on queue capacity occur more frequently on larger packets, resulting in throughput degradation.

As the processing time of Reinhardt takes 32 clocks of delay, the measured latency with Reinhardt should be higher than the baseline. However, as shown in Fig. 10(b), the latency with Reinhardt is very close to the latency of the baseline regardless of the number of the queues since the overhead caused by Reinhardt is negligible (nanosecond level). In conclusion, packet transmission can be guaranteed with a latency of 60 μs, regardless of the number of queues.

Resubmission: We measure the throughput and latency variations of Reinhardt with the different number of resubmissions to see the performance impacts due to resubmissions. For this, we configure Reinhardt with *eight* queues. As shown in Fig. 11(a), the throughputs with up to 4 times submissions are steady. However, as the number of submissions increases over 4 times, 20% of throughput degradation occurs every resubmission. Similar to the throughput degradation, the latency gradually increases with the increasing number of resubmissions, but it is still negligible. There is a delay of about 200 ns per round, which means that the delay is less than 1 μs with 4 resubmissions. As the number of resubmissions increases, the delay gradually accumulates; 32 resubmissions increase the overall latency by about 6–8 μs.

These results are because the dynamic reconfigurability can utilize potential resources in FPGA; NetFPGA-SUME processes a packet with 256-bit chunks per clock, but it takes 25.6 ns to get a 256-bit chunk at the 10 Gbps speed, which is about 4 clocks at 160 MHz. Thus, the chunks of incoming packets are processed every 4 clocks and make the gap of 3 clocks during queue entry. This gap is utilized for the chunks of resubmitted packets, so the submissions up to 4 times will not suffer throughput degradation (Park and Shin, 2021). However, if the number of resubmissions exceeds over 4 times, some of the incoming packets, unfortunately, are dropped due to the limited capability of the FPGA hardware.

5.3. Regex pattern deployment

Pattern capacity: Pattern capacity means how many regex patterns Reinhardt can accommodate at once without performance loss. For this, we randomly select regex patterns among the regex set used to determine the core constraints in Section 4 until the core becomes full (with four resubmissions). The experiment was repeated 100 times. Fig. 12 illustrates the number of deployed regex patterns for the different core sizes into a candlestick graph (the range of standard deviations around the average and its shadow as a min–max). Here, we see that the number of deployed patterns increases up to 1313 as the core size increases.

Pattern configuration time: One of the key contributions in Reinhardt is the dynamic reconfigurability without service interruption. To show its agility, we measure the pattern configuration times with the different core sizes. The configuration assumes that a host software configures all cells in the core, including resubmitting 4 times

Table 5
Reinhardt cell configuration time.

Core size	# of the cells	# of patterns	Time (s)
24 × 160 × 8	15,360	≤160	0.116
24 × 300 × 4	28,800	≤295	0.186
24 × 665 × 2	63,840	≤590	0.403
24 × 1580 × 1	151,680	≤1313	0.965

⋆ The number of the cells to configure considers four submissions
∗ The number of patterns is referenced in Fig. 12.

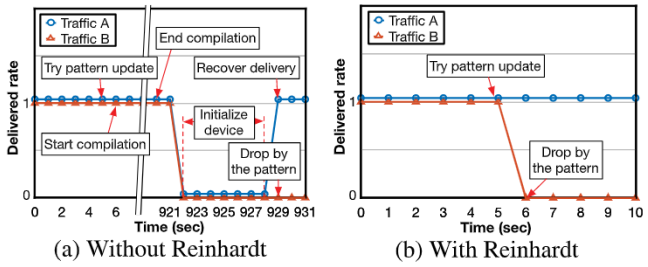


Fig. 13. Update and response times of the Simple IPS with Reinhardt (Graph-(a) is the same with Fig. 3-(b) in Section 2.4).

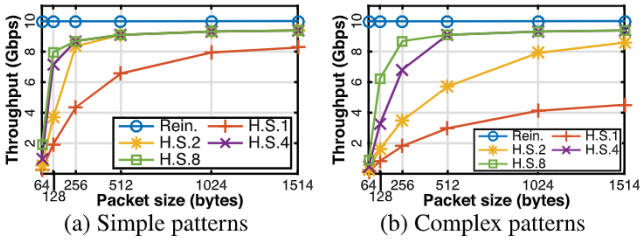
(i.e., worst-case). Table 5 shows the number of cells being configured and their configuration times. While these times are measured under the worst cases, all configurations are completed within a second. It takes 116 ms to configure 160 patterns and 965 ms to configure 798 patterns, which is much faster than the configuration times in existing solutions described in Table 3. Note that the communication between the datapath and the software framework takes most of the configuration time while the actual update in the hardware is instantly done.

Update and response times in NIDS/IPS: To validate how effectively Reinhardt addresses the challenge in FPGA-based DPI, we back to our motivating example of Fig. 3 in Section 2.4 and perform the same evaluation with Reinhardt. Fig. 13-(b) shows its result. As soon as a new pattern is installed to inspect and filter Flow B, the new pattern is used to match the incoming traffic instantly while the device is up and running as ever. Hence, unlike the motivating example, Flow A is delivered continuously, but Flow B is dropped immediately after the pattern update.

5.4. Comparison with DPDK-Hyperscan

Intel DPDK-Hyperscan (Intel, 2021; Wang et al., 2019) is one of the best-of-breed baselines for fast regex matching that utilizes a multi-core CPU. Here, we analyze the advantages and disadvantages of Reinhardt by comparing it with DPDK-Hyperscan. We implement a simple DPDK-Hyperscan application using the Hyperscan open-source (Intel, 2019) and DpdkBridge (PcapPlusPlus, 2020) that receives packets from network interfaces and matches the packets with target patterns and run the application on Intel Xeon E5-2630 (10 cores, Hyper-Threading disabled), 64 GB of RAM and Intel X520 10GbE NICs. During the evaluation, we used 843 patterns compatible with Reinhardt of 847 patterns from Intel's sample data (Snort-PCRE) (Intel 01.org, 2019) as the target patterns.

Regex patterns within the Reinhardt capacity: We randomly select 100 simple and 100 complex regex patterns among the 843 patterns and measure the throughputs with (Reinhardt 24 × 160 × 8) and DPDK-Hyperscan, respectively. We repeat 20 times, and Fig. 14 shows their average values. Here, we observe that Reinhardt constantly achieves 10 Gbps regardless of pattern complexity, but DPDK-Hyperscan shows the performance degradation as the packet size and pattern complexity increase. DPDK-Hyperscan requires four cores for simple patterns and eight cores for complex patterns to get the maximum performance.



⋆ The core size of Reinhardt is 24×160×8
⋆ H.S.n means Hyperscan with n cores

Fig. 14. Throughput for 100 patterns (vs. Hyperscan).

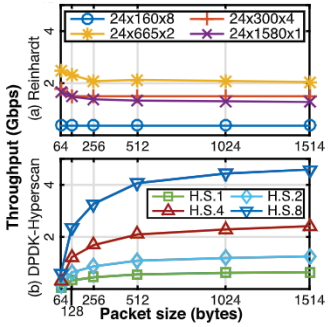


Fig. 15. Throughput for 843 patterns (vs. Hyperscan).

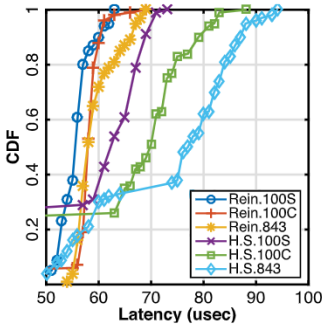


Fig. 16. Latency for 100/843 patterns (vs. Hyperscan).

However, the performance degradation is still observed in the cases of 64–256 bytes packets, and the overall throughput is up to 9.3 Gbps.

Regex patterns over the Reinhardt capacity: To measure the performance with a larger pattern set, we deploy all 843 patterns into the different sizes of Reinhardt, taking into account the excess of the number of resubmissions ensuring maximum speed (i.e., 4 times). The Reinhardt cores (24 × 160 × 8,24 × 300 × 4,24 × 665 × 2 and 24 × 1580 × 1) can accommodate all the patterns by taking total 19, 10, 5 and 2 submission rounds respectively. Fig. 15 presents their throughput averages for 20 iterations. Reinhardt with the core sizes (24 × 665 × 2 and 24 × 1580 × 1) can handle all the patterns within the capacity or with one more submission, they almost preserve the original throughput, but Reinhardt with the core sizes (24 × 160 × 8 and 24 × 300 × 4) suffers significant performance degradation from 10/5.6 Gbps to 0.37/1.5 Gbps respectively due to too many resubmissions. Reinhardt shows the similar performance of Hyperscan with 1–4 cores. However, Hyperscan can perform well with more cores (e.g., Hyperscan approaches 4.5 Gbps with 8 cores).

Latency measurements: We compare the latencies of Reinhardt and DPDK-Hyperscan while handling 100 simple/complex patterns and 843 patterns. As seen in Fig. 16, the latency of Reinhardt is almost similar regardless of the complexity and the number of patterns. In contrast, the latency of DPDK-Hyperscan is much slower to 50% with a

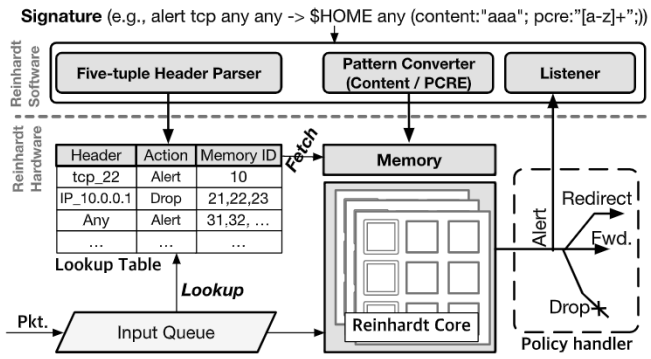


Fig. 17. Reinhardt as an NIDS/NIPS.

wide range of variation (i.e., jitter) as the complexity and the number of patterns increase.

Discussion: This evaluation shows that Reinhardt guarantees the line-rate throughput within the capacity and still provides the stable throughput and the latency regardless of the packets sizes and the pattern complexities even though overloaded. DPDK-Hyperscan also achieves the outstanding throughput, and it can even obtain better performance than Reinhardt when utilizing a large number of CPU cores. However, we see that its throughput and latency are fairly affected by the packets sizes and the pattern complexities.

6. Use cases

In this section, we discuss how Reinhardt can be employed in real-world applications. For this, we introduce three use cases: Reinhardt as an NIDS/NIPS, (2) Replacement of the PCRE engine in Snort IDS, and Reinhardt integration with software-defined networking (SDN). Here, we provide the intuition on how to leverage the unique strengths of Reinhardt through these use cases.

6.1. Reinhardt as an NIDS/NIPS

Reinhardt can be utilized as a stand-alone NIDS/NIPS with two extensions (e.g., a policy table and an output engine). Fig. 17 illustrates the overall workflow of the Reinhardt NIDS/NIPS. Here, attack signatures are split into the headers in the policy table (the 5-tuple lookup table) and the corresponding patterns (i.e., the "content" and the "pcre"), and Reinhardt converts the patterns to the Reinhardt logic in the Reinhardt memory. Here, the pairs of a header and a pattern are placed in the same memory ID for resubmitting, and the IDs of the corresponding area are assigned to each header. When packets arrive at Reinhardt, Reinhardt fetches the matching logic from the memory to the Reinhardt core, and the Reinhardt core performs REM. When Reinhardt finds any matches, the policy handler (i.e., an output engine) follows one of the predefined actions: (1) *alert* (sending an alert message to the host side), (2) *drop* (terminating the packet processing and dropping the matched packet), and (3) *redirect* (forwarding the packet to alternative routes (e.g., honeypot) instead of the original destination).

The performance of the Reinhardt NIDS/NIPS is equal to that of the naive Reinhardt that we measured in Section 5 (i.e., 10 Gbps). In fact, while Reinhardt NIDS/NIPS creates an additional delay to search the lookup table, this delay only takes a few nanoseconds, which are hard to measure at end hosts. As a result, Reinhardt can effectively work as an NIDS/IPS. In particular, the ability of Reinhardt to update regex patterns in real-time allows detecting newly identified attack signatures by dynamically loading them on the Reinhardt core.

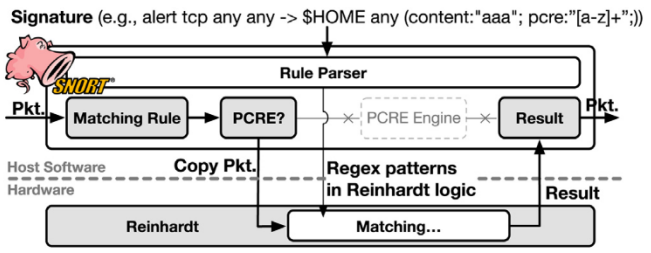


Fig. 18. Snort IDS with Reinhardt-based REM.

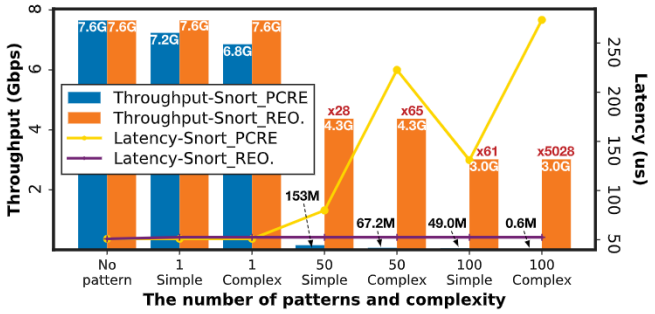


Fig. 19. Performance comparison for Snort IDS acceleration.

6.2. PCRE replacement in Snort IDS

In Section 2.2, we have presented the performance issue of software-based REM through the PCRE engine of Snort IDS and seen the performance drop up to 99% with only dozens of regex patterns. Here, we present Reinhardt as the replacement of the PCRE engine to accelerate the performance. Fig. 18 shows the overall design of Snort IDS with Reinhardt. When the Snort IDS initializes the regex patterns in given rules, Reinhardt takes these regex patterns and deploys them in the core. Then, during the rule option evaluation against incoming packets, the Snort IDS passes the packets to Reinhardt under evaluation and receives the matching results from Reinhardt.

Fig. 19 shows the throughput variations of Snort IDS with Reinhardt. Here, the test environment is the same as that of the PCRE throughput benchmark (described in Section 2.2). One of the conspicuous results is that Reinhardt performs REM regardless of the complexity of regex patterns unlike the PCRE engine in Snort IDS; thus, there is no throughput difference with the same number of simple and complex rules. In terms of performance improvement, there is no significant performance drop with Reinhardt, and Reinhardt could improve the overall throughput with both 50 and 100 rules up to 65 times compared to those with the PCRE engine. Unfortunately, there is gradual throughput degradation as the number of rules increases due to the architectural design of Snort IDS. The throughput degradation mostly comes from the iterations to check the other rule options (e.g., *offset*, *distance*, and *within* options) of each rule in Snort IDS, which means that it is not due to Reinhardt.

6.3. Reinhardt integration with SDN

These days, software-defined networking (SDN) is widely used in various networking environments because of its dynamicity and flexibility. However, its use is highly limited to networking (e.g., flow routing and load-balancing) due to the limited visibility of packet payloads. For this reason, there have been several studies (Yoon et al., 2015) to add security functionalities (i.e., DPI) into the SDN control plane, but they also point out that performing DPI in the control plane causes the overall network slowdown owing to the low performance of DPI (specifically, software-based REM).

Here, Reinhardt can be a good choice to provide such security functions and even improve their performance in SDN environments. Fig. 20

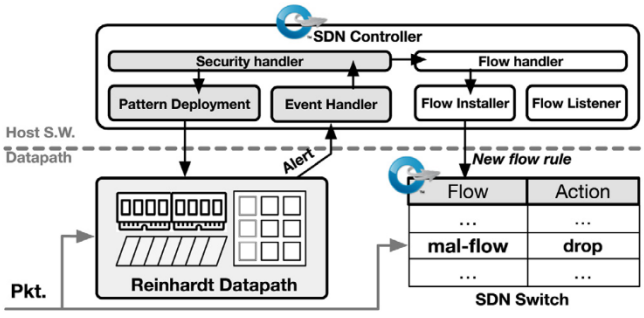


Fig. 20. Reinhardt integration with SDN.

presents the overall design of an SDN environment with Reinhardt. First, a Reinhardt application is integrated into an SDN controller as a security handler, and deploys attack signatures (regex patterns) into the Reinhardt core located in a data plane. When Reinhardt detects any matches with deployed patterns, it alerts the application with a matched pattern, and then the application lets a flow handler deal with malicious flows by installing flow rules (e.g., dropping the flows) into SDN switches. Whenever there is any change in a network or in regex patterns, the application keeps the core up to date.

7. Related work

FPGA-based REM: Sidhu and Prasanna (2001) proposed a one-hot encoding scheme to express NFA with circuit blocks, and its subsequent studies (Hutchings et al., 2002; Lin et al., 2007) inspire Reinhardt. Also, Sert and Bazlamacci (2021) proposed a NFA-based REM based on 2-stride-input-based transitions for high performance and memory efficiency. Some studies (Hieu et al., 2013; Wang et al., 2013; Nakahara et al., 2012) suggested resource-efficient regex circuits. Other studies (Mitra et al., 2007; Nakahara et al., 2010; Yamagaki et al., 2008; Yang and Prasanna, 2012) focused on high-performance FPGA-based REM.

Configurable FPGA-based REM: To do REM in the FPGA hardware, one strategy is generating FPGA source code (i.e. HDL) from regex patterns automatically (Bispo et al., 2006; Mitra et al., 2007; Sourdis et al., 2008) and compiling the generated HDL code to FPGA. They look very similar to Reinhardt at first glance, but they must have a long compilation process (i.e., synthesis, implementation, bitstream generation, and bitstream download) from the automatically generated HDL codes. Hence, they still require a long processing time and suffer inevitable service interruption to update regex patterns, as well as cannot keep pace with fast-changing networking such that they are difficult to be adopted as DPI in real networks. That is, it is far from being able to be reconfigured and updated in real-time. Reinhardt, however, can update regex patterns in real-time without any service interruption, it can respond quickly to any situation while providing high-performance.

Memory-based approaches (Baker et al., 2006; Brodie et al., 2006; Tang et al., 2014) and a CPU-FPGA hybrid approach (Sidler et al., 2017) are also able to give FGPA real-time configurability. However, as they are memory-intensive, their matching should work in sequential processing by DFA so that cannot fully support massive parallel processing. Also, they cannot support *constraint repetitions* (i.e., *, +) from a security perspective, so it is difficult to handle a signature including such NOP sleds often prepended before a shellcode in remote exploit payloads to make an attack more reliable (Bispo et al., 2007; Yang and Prasanna, 2011). In the case of the CPU-FPGA hybrid one, as it is a design pre-implementing possible cases within a fixed form of automaton, the number of states that can be stored is limited (≤ 32), post-processing by CPU is essential, and all expressions are not supported (e.g., interval `{m,n}`, begin/not/and (`^`, `[$]$`) and concatenation). Whereas Reinhardt is fully programmable and free to express any shape of automata, more states are available, and all metacharacters are supported.

Programmable-dataplane-based REM: Programming protocol-independent packet processors (P4) allows a limited syntax in pattern matching. For example, DeepMatch (Hypolite et al., 2020) and Jepsen et al. (2019) proposed a way of pattern matching with P4. However, while supporting string matching and glob patterns, they do not allow frequently used syntax (e.g., `{m,n}`, `[^]`, and `[a-f]`). On the other hand, Reinhardt is specialized for REM, supporting the full regex syntax.

Network packet inspection: Xu et al. (2016) and Imran et al. (2021) summarized various approaches to REM. Kargus (Jamshed et al., 2012), Haetae (Nam et al., 2015), and Hyperscan (Wang et al., 2019) presented accelerated pattern matching using modern CPU techniques, e.g. hybrid multi-core/GPU, many-core, or SIMD operations. DFC (Choi et al., 2016) is a memory-efficient and cache-friendly data structure for fast multi-pattern matching. Also, Irfan et al. (2022) surveyed recent studies utilizing content addressable memory (CAM) for REM. For example, Some (Meiners et al., 2010; Sourdis and Pnevmatikatos, 2004) proposed CAM-based pattern matching approaches, improving matching throughputs. Srinivasavarma et al. (2022) used ternary content addressable memory (TCAM) for low-delay pattern matching. As we can see here, enhancing performance has been a major challenge in DPI.

8. Conclusion

FPGA-based REM satisfies high-performance but its flexibility becomes a major and critical limitation as it involves a time-consuming process to update regex patterns in the hardware. To address this, we have presented Reinhardt, an improved hardware architecture of implementing regex patterns with its reconfigurable cells to support dynamic pattern updates. Our evaluation and case studies demonstrate that Reinhardt updates regex patterns immediately without service interruption and serves as a high-performance NIDS/IPS and hardware acceleration for REM. We believe that Reinhardt can be positioned as an advanced DPI that is adept at responding to frequent changes and can also be implemented as a specialized regex processor (e.g., ASIC) in the future.

CRediT authorship contribution statement

Jaehyun Nam: Conceptualization, Methodology, Validation, Software, Writing – original draft, Writing – review & editing. **Seung Ho Na:** Conceptualization, Methodology, Software, Writing – original draft, Writing – review & editing. **Seungwon Shin:** Project administration, Supervision, Funding acquisition, Writing – original draft, Writing – review & editing. **Taejune Park:** Conceptualization, Formal analysis, Project administration, Supervision, Funding acquisition, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

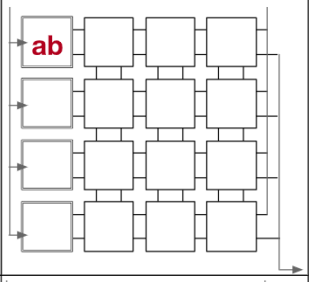
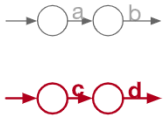
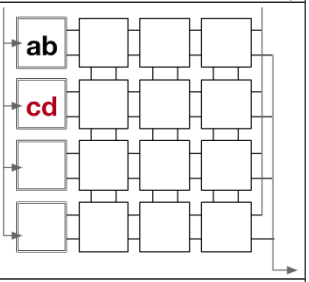
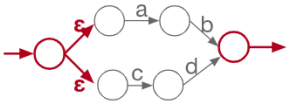
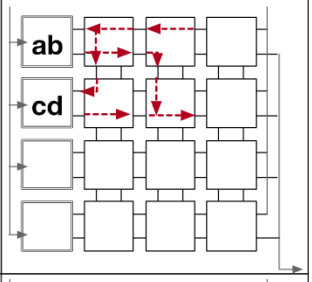
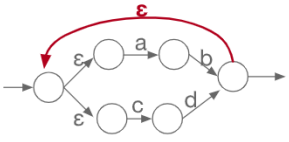
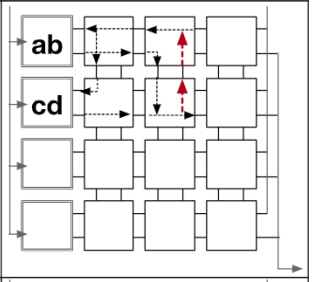
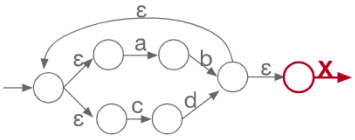
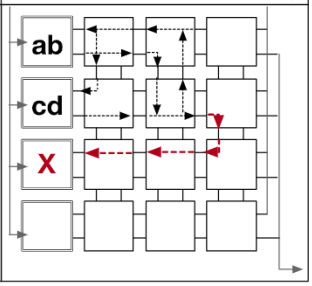
Acknowledgments

This work was supported in part by National Research Foundation of Korea (No. 2022R1C1C1006967).

Appendix. Converting a regex pattern to reinhardt logic

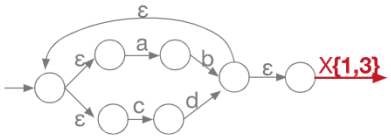
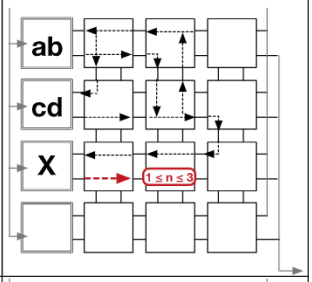
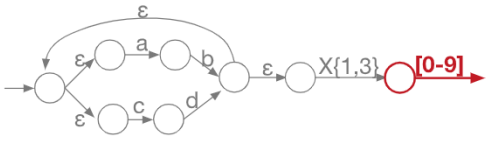
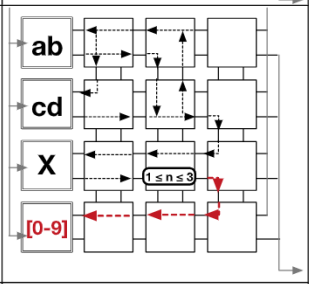
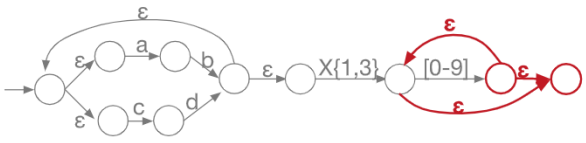
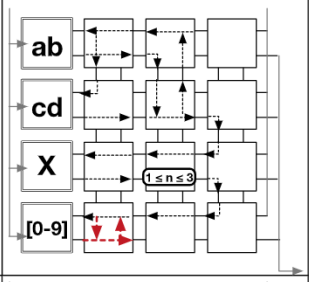
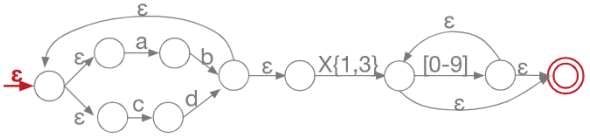
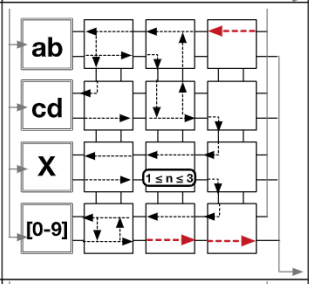
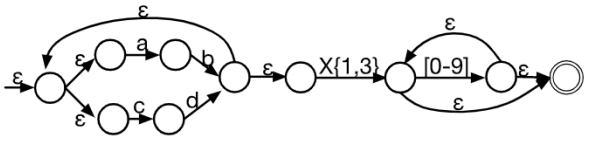
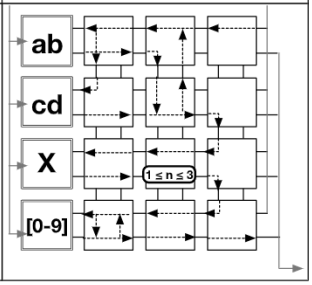
See Table A.1.

Table A.1
Converting a regex pattern '(ab|cd) + X1, 3[0 − 9] ∗' to Reinhardt logic.

#	Regex pattern	NFA	Reinhardt Logic
1	ab		
2	ab cd		
3	ab|cd		
4	(ab|cd)+		
5	(ab|cd)+X		

(continued on next page)

Table A.1 (*continued*).

#	Regex pattern	NFA	Reinhardt Logic
6	(ab|cd)+X**{1,3}**		
7	(ab|cd)+X{1,3}**[0-9]**		
8	(ab|cd)+X{1,3}[0-9]*		
9	(ab|cd)+X{1,3}[0-9]*		
done	**(ab|cd)+X{1,3}[0-9]***		

References

AbuHmed, T., Mohaisen, A., Nyang, D., 2008. A survey on deep packet inspection for intrusion detection systems. arXiv preprint arXiv:0803.0037.

Ahmadi, S., 2016. Toward 5G Xilinx solutions and enablers for next-generation wireless systems. In: Xilinx MPSoCs and FPGAs.

Atasu, K., Polig, R., Rohrer, J., Hagleitner, C., 2013. Exploring the design space of programmable regular expression matching accelerators. J. Syst. Archit. 59 (10), 1184–1196.

Baker, Z.K., Jung, H.-J., Prasanna, V.K., 2006. Regular expression software deceleration for intrusion detection systems. In: 2006 International Conference on Field Programmable Logic and Applications. IEEE, pp. 1–8.

Bispo, J., Sourdis, I., Cardoso, J.M., Vassiliadis, S., 2006. Regular expression matching for reconfigurable packet inspection. In: Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on. IEEE, pp. 119–126.

Bispo, J., Sourdis, I., Cardoso, J.M., Vassiliadis, S., 2007. Synthesis of regular expressions targeting fpgas: Current status and open issues. In: International Workshop on Applied Reconfigurable Computing. Springer, pp. 179–190.

Bremler-Barr, A., Harchol, Y., Hay, D., Koral, Y., 2014. Deep packet inspection as a service. In: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies. ACM, pp. 271–282.

Brodie, B.C., Taylor, D.E., Cytron, R.K., 2006. A scalable architecture for high-throughput regular-expression pattern matching. In: 33rd International Symposium on Computer Architecture (ISCA'06). IEEE, pp. 191–202.

Chamola, V., Patra, S., Kumar, N., Guizani, M., 2020. FPGA for 5G: Re-configurable hardware for next generation communication. IEEE Wirel. Commun..

Che, S., Li, J., Sheaffer, J.W., Skadron, K., Lach, J., 2008. Accelerating compute-intensive applications with GPUs and FPGAs. In: 2008 Symposium on Application Specific Processors. IEEE, pp. 101–107.

Check Point Software Technologies Ltd, 2019. How quick are turn-around times for IPS signature updates addressing newly found vulnerabilities. https://supportcenter.checkpoint.com/supportcenter/portal?eventSubmit_doGoviewsolutiondetails=&solutionid=sk98937.

Chen, H., Chen, Y., Summerville, D.H., 2010. A survey on the application of FPGAs for network infrastructure security. IEEE Commun. Surv. Tutor. 13 (4), 541–561.

Choi, B., Chae, J., Jamshed, M., Park, K., Han, D., 2016. DFC: Accelerating string pattern matching for network applications. In: NSDI. pp. 551–565.

Cope, B., Cheung, P.Y., Luk, W., Howes, L., 2010. Performance comparison of graphics processors to reconfigurable logic: A case study. IEEE Trans. Comput. 59 (4), 433–448.

CORSA, 2019. Is your network security keeping up?. https://www.corsa.com/wp-content/uploads/PDFs/Corsa_WP-Is_Your_Network_Security_Keeping_Up.pdf.

Detail, C., 2021. The ultimate security vulnerability datasource. https://www.cvedetails.com.

Emerging Threats, 2021. Emerging threats rulesets. https://rules.emergingthreats.net https://doc.emergingthreats.net.

Fahmy, S.A., Vipin, K., Shreejith, S., 2015. Virtualized FPGA accelerators for efficient cloud computing. In: 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, pp. 430–435.

Fernandes, G., Rodrigues, J.J., Carvalho, L.F., Al-Muhtadi, J.F., Proença, M.L., 2019. A comprehensive survey on network anomaly detection. Telecommun. Syst. 70 (3), 447–489.

FireEye, 2021. FireEye dynamic threat intelligence cloud. https://www.threatprotectworks.com.

Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., et al., 2018. Azure accelerated networking: SmartNICs in the public cloud. In: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18). Renton, WA.

Fowers, J., Brown, G., Cooke, P., Stitt, G., 2012. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays. ACM, pp. 47–56.

Ganegedara, T., Yang, Y.-H.E., Prasanna, V.K., 2010. Automation framework for large-scale regular expression matching on FPGA. In: 2010 International Conference on Field Programmable Logic and Applications. IEEE, pp. 50–55.

Gupta, P., 2016. Accelerating datacenter workloads. In: 26th International Conference on Field Programmable Logic and Applications.

Hamming, R.W., 1950. Error detecting and error correcting codes. Bell Syst. Tech. J. 29 (2), 147–160.

Hazel, P., 2005. Pcre: Perl compatible regular expressions.

Hieu, T.T., Thinh, T.N., Tomiyama, S., 2013. ENREM: An efficient NFA-based regular expression matching engine on reconfigurable hardware for NIDS. J. Syst. Archit. 59 (4–5), 202–212.

HPC, I., 0000. Inside HPC Special Report: Are FPGAs the Answer to the "Compute Gap"?, https://plan.seek.intel.com/PSG_WW_NC_LPOI_EN_2018_HPCComputeGap.

Hutchings, B.L., Franklin, R., Carver, D., 2002. Assisting network intrusion detection with reconfigurable hardware. In: Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on. IEEE, pp. 111–120.

Hypolite, J., Sonchack, J., Hershkop, S., Dautenhahn, N., DeHon, A., Smith, J.M., 2020. DeepMatch: practical deep packet inspection in the data plane using network processors. In: Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies, pp. 336–350.

Imran, M., Bashir, F., Jafri, A.R., Rashid, M., ul Islam, M.N., 2021. A systematic review of scalable hardware architectures for pattern matching in network security. Comput. Electr. Eng. 92, 107169.

InfoSecurity, 2019. Advanced malware detection - signatures vs. Behavior analysis. https://www.infosecurity-magazine.com/opinions/malware-detection-signatures/.

Intel, 0000. Cloud computing, https://www.intel.co.kr/content/www/kr/ko/products/docs/storage/programmable/applications/cloud.html.

Intel, 0000. 5G Wireless, https://www.intel.co.kr/content/www/kr/ko/communications/products/programmable/applications/baseband.html.

Intel, 2019. Hyperscan. https://github.com/intel/hyperscan.

Intel, 2021. DPDK: Data plane development kit. http://dpdk.org.

Intel 01.org, 2019. Hyperscan sample data. https://01.org/downloads/sample-data-hyperscan-hsbench-performance-measurement.

Irfan, M., Sanka, A.I., Ullah, Z., Cheung, R.C., 2022. Reconfigurable content-addressable memory (CAM) on FPGAs: A tutorial and survey. Future Gener. Comput. Syst. 128, 451–465.

Jamshed, M.A., Lee, J., Moon, S., Yun, I., Kim, D., Lee, S., Yi, Y., Park, K., 2012. Kargus: a highly-scalable software-based intrusion detection system. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. ACM, pp. 317–328.

Jepsen, T., Alvarez, D., Foster, N., Kim, C., Lee, J., Moshref, M., Soulé, R., 2019. Fast string searching on pisa. In: roceedings of the 2019 ACM Symposium on SDN Research, pp. 21–28.

Johnson, A., Mackenzie, K., 2001. Pattern matching in reconfigurable logic for packet classification. In: Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. ACM, pp. 126–130.

Kreibich, C., Handley, M., Paxson, V., 2001. Network intrusion detection: Evasion, traffic normalization, and nd-to-end protocol semantics. In: Proc. USENIX Security Symposium, Vol. 2001.

Kumar, S., 2007. Survey of Current Network Intrusion Detection Techniques. Washington Univ. in St. Louis, pp. 1–18.

Lavin, C., Nelson, B., Hutchings, B., 2013. Impact of hard macro size on FPGA clock rate and place/route time. In: International Conference on Field Programmable Logic and Applications. IEEE, pp. 1–6.

Lavin, C., Padilla, M., Lamprecht, J., Lundrigan, P., Nelson, B., Hutchings, B., 2011. HMFlow: accelerating FPGA compilation with hard macros for rapid prototyping. In: 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines. IEEE, pp. 117–124.

Li, B., Tan, K., Luo, L.L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., Chen, E., 2016. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In: Proceedings of the 2016 ACM SIGCOMM Conference. ACM, pp. 1–14.

Lin, C.-H., Huang, C.-T., Jiang, C.-P., Chang, S.-C., 2007. Optimization of pattern matching circuits for regular expression on FPGA. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 15 (12), 1303–1310.

Love, A., Zha, W., Athanas, P., 2013. In pursuit of instant gratification for FPGA design. In: 2013 23rd International Conference on Field Programmable Logic and Applications. IEEE, pp. 1–8.

Meiners, C.R., Patel, J., Norige, E., Torng, E., Liu, A.X., 2010. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In: Proceedings of the 19th USENIX Conference on Security. USENIX Association, p. 8.

Mellanox, 2021. Titan IC RXP. https://www.mellanox.com/titan-ic.

Mitra, A., Najjar, W., Bhuyan, L., 2007. Compiling pcre to fpga for accelerating snort ids. In: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems. ACM, pp. 127–136.

Mitre, 2021. Common vulnerabilities and exposures (CVE). https://cve.mitre.org/.

Nakahara, H., Sasao, T., Matsuura, M., 2010. A regular expression matching circuit based on a modular non-deterministic finite automaton with multi-character transition. In: Proc. 16th Workshop on Synthesis and System Integration of Mixed Information Technologies, pp. 359–364.

Nakahara, H., Sasao, T., Matsuura, M., 2012. A design method of a regular expression matching circuit based on decomposed automaton. IEICE Trans. Inf. Syst. 95 (2), 364–373.

Nam, J., Jamshed, M., Choi, B., Han, D., Park, K., 2015. Haetae: Scaling the performance of network intrusion detection with many-core processors. In: International Workshop on Recent Advances in Intrusion Detection. Springer, pp. 89–110.

NetFPGA, 2014. NetFPGA-SUME board. https://netfpga.org/site/#/systems/1netfpga-sume/details/.

NetFPGA-SUME, 2020. NetFPGA SUME reference NIC. https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/NetFPGA-SUME-Reference-NIC.

Nping, 2021. An open source network packet generation. https://nmap.org/nping/.

Park, T., Shin, S., 2021. Mobius: Packet re-processing hardware architecture for rich policy handling on a network processor. J. Netw. Syst. Manage. 29 (1), 1–26.

PcapPlusPlus, 2020. Dpdkbridge. https://github.com/seladb/PcapPlusPlus/tree/master/Examples/DpdkBridge.

Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., et al., 2014. A reconfigurable fabric for accelerating large-scale datacenter services. ACM SIGARCH Comput. Archit. News 42 (3), 13–24.

RAPID7, 2017. The pros & cons of intrusion detection systems. https://blog.rapid7.com/2017/01/11/the-pros-cons-of-intrusion-detection-systems/.

Ribeiro, C., Gameiro, A., 2017. A software-defined radio FPGA implementation of OFDM-based PHY transceiver for 5G. Analog Integr. Circuits Signal Process. 91 (2), 343–351.

Ricart-Sanchez, R., Malagon, P., Salva-Garcia, P., Perez, E.C., Wang, Q., Calero, J.M.A., 2018. Towards an FPGA-accelerated programmable data path for edge-to-core communications in 5G networks. J. Netw. Comput. Appl. 124, 80–93.

Sert, K., Bazlamacci, C.F., 2021. NFA based regular expression matching on FPGA. In: 2021 International Conference on Computer, Information and Telecommunication Systems (CITS). IEEE, pp. 1–5.

Sidhu, R., Prasanna, V.K., 2001. Fast regular expression matching using FPGAs. In: Annual IEEE Symposium on Field-Programmable Custom Computing Machines. IEEE, pp. 227–238.

Sidler, D., István, Z., Owaida, M., Alonso, G., 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 403–415.

Smith, R., Goyal, N., Ormont, J., Sankaralingam, K., Estan, C., 2009. Evaluating GPUs for network packet signature matching. In: 2009 IEEE International Symposium on Performance Analysis of Systems and Software. IEEE, pp. 175–184.

Snort, 2021. Open-source NIDS/IPS. https://www.snort.org/.

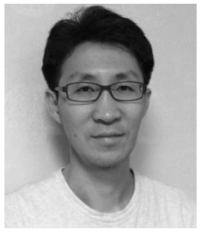
Sourdis, I., Bispo, J., Cardoso, J.M., Vassiliadis, S., 2008. Regular expression matching in reconfigurable hardware. J. Signal Process. Syst. 51 (1), 99–121.

Sourdis, I., Pnevmatikatos, D., 2004. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In: 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. IEEE, pp. 258–267.

Srinivasavarma, V.S., Pydi, S.R., Mahammad, S.N., 2022. Hardware-based multi-match packet classification in NIDS: an overview and novel extensions for improving the energy efficiency of TCAM-based classifiers. J. Supercomput. 1–36.
Suricata, 2021. An open source-based intrusion detection system (IDS). https://suricata-ids.org/.
Symantec, 2002. Managing intrusion detection systems in large organizations, part one. https://www.symantec.com/connect/articles/managing-intrusion-detection-systems-large-organizations-part-one.
Tang, Q., Jiang, L., Liu, X.-x., Dai, Q., 2014. A real-time updatable FPGA-based architecture for fast regular expression matching. Procedia Comput. Sci. 31, 852–859.
Thomas, D.B., Howes, L., Luk, W., 2009. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays. ACM, pp. 63–72.
Thompson, K., 1968. Programming techniques: Regular expression search algorithm. Commun. ACM 11 (6), 419–422.
Tupakula, U., Varadharajan, V., Akku, N., 2011. Intrusion detection techniques for infrastructure as a service cloud. In: 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing. IEEE, pp. 744–751.
Van Lunteren, J., 2006. High-performance pattern-matching for intrusion detection. In: Proceedings IEEE International Conference on Computer Communications. Citeseer, pp. 1–13.
Vaquero, L.M., Morán, D., Galán, F., Alcaraz-Calero, J.M., 2012. Towards runtime reconfiguration of application control policies in the cloud. J. Netw. Syst. Manage. 20 (4), 489–512.
Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E.P., Ioannidis, S., 2008. Gnort: High performance network intrusion detection using graphics processors. In: International Workshop on Recent Advances in Intrusion Detection. Springer, pp. 116–134.
Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G., Hu, J., Zhu, H., 2019. Hyperscan: a fast multi-pattern regex matcher for modern CPUs. In: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). pp. 631–648.
Wang, H., Pu, S., Knezek, G., Liu, J.-C., 2013. Min-max: A counter-based algorithm for regular expression matching. IEEE Trans. Parallel Distrib. Syst. 24 (1), 92–103.
Xu, C., Chen, S., Su, J., Yiu, S.-M., Hui, L.C., 2016. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. IEEE Commun. Surv. Tutor. 18 (4), 2991–3029.
Yamagaki, N., Sidhu, R., Kamiya, S., 2008. High-speed regular expression matching engine using multi-character NFA. In: Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on. IEEE, pp. 131–136.
Yang, Y.-H.E., Jiang, W., Prasanna, V.K., 2008. 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. ACM, pp. 30–39.
Yang, Y.-H.E., Prasanna, V.K., 2011. Space-time tradeoff in regular expression matching with semi-deterministic finite automata. In: 2011 Proceedings IEEE INFOCOM. IEEE, pp. 1853–1861.
Yang, Y.-H., Prasanna, V., 2012. High-performance and compact architecture for regular expression matching on FPGA. IEEE Trans. Comput. 61 (7), 1013–1025.
Yoon, C., Park, T., Lee, S., Kang, H., Shin, S., Zhang, Z., 2015. Enabling security functions with SDN: A feasibility study. Comput. Netw. 85, 19–35.
Yu, X., Becchi, M., 2013. GPU acceleration of regular expression matching for large datasets: exploring the implementation space. In: Proceedings of the ACM International Conference on Computing Frontiers. ACM, p. 18.
Yu, J., Zhu, Y., Xia, L., Qiu, M., Fu, Y., Rong, G., 2011. Grounding high efficiency cloud computing architecture: HW-sw co-design and implementation of a stand-alone web server on FPGA. In: Fourth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2011). IEEE, pp. 124–129.
Zeek (Bro), 2021. Network security monitor. https://www.zeek.org/.
Zilberman, N., Audzevich, Y., Covington, G.A., Moore, A.W., 2014. NetFPGA SUME: Toward 100 gbps as research commodity. IEEE Micro 34 (5), 32–41.
Zu, Y., Yang, M., Xu, Z., Wang, L., Tian, X., Peng, K., Dong, Q., 2012. GPU-based NFA implementation for memory efficient high speed regular expression matching. In: ACM SIGPLAN Not.. 47, (8), ACM, pp. 129–140.

Jaehyun Nam is an assistant professor at Department of Computer Engineering, Dankook University, South Korea. He received his Ph.D. and M.S. degree in School of Computing (Information Security) from KAIST and his B.S. degree in Computer Science and Engineering from Sogang University in Korea. His research interests focus on networked systems and security. He is especially interested in security issues in cloud and edge computing systems (including SDN, NFV, IoT, and containers).

Seung Ho Na is a Ph.D. candidate in the Network and System Security Lab, advised by Seungwon Shin, in the School of Electrical Engineering at KAIST. He received his M.S. degree and B.S. degree in Electrical Engineering from KAIST. His research interests span the areas of data-driven security, artificial intelligence (AI) security, and cyber threat intelligence (CTI). Currently, he focuses on preserving and improving privacy of machine learning systems.



Seungwon Shin is an associate professor in School of Electrical Engineering at KAIST. He received his Ph.D. degree in Computer Engineering from the Electrical and Computer Engineering Department, Texas A&M University, and his M.S. degree and B.S. degree from KAIST, both in Electrical and Computer Engineering. His research interests span the areas of SDN security, IoT security, Botnet analysis/detection, DarkWeb analysis and cyber threat intelligence (CTI).

Taejune Park is an assistant professor at the Department of Artificial Intelligence Convergence, Chonnam National University, South Korea. He received B.S. in Computer Engineering at Korea Maritime and Ocean University, South Korea, and M.S. and Ph.D. in information security at KAIST, South Korea. His research interests focus on network and IoT security and reliable low-latency communications.