

Reinhardt: Real-time Reconfigurable Hardware Architecture for Regular Expression Matching in DPI

Taejune Park
taejune.park@jnu.ac.kr
Chonnam National University
Republic of Korea

Jaehyun Nam
jn@accuknox.com
AccuKnox
USA

Seung Ho Na
harry.na@kaist.ac.kr
KAIST
Republic of Korea

Jaewoong Chung
jaewoong.chung@atto-research.com
Atto Research
Republic of Korea

Seungwon Shin
claude@kaist.ac.kr
KAIST
Republic of Korea

ABSTRACT

Regular expression (regex) matching is an integral part of deep packet inspection (DPI) but a major bottleneck due to its low performance. For regex matching (REM) acceleration, FPGA-based studies have emerged and exploited parallelism by matching multiple regex patterns concurrently. However, even though guaranteeing high-performance, existing FPGA-based regex solutions do not still support dynamic updates in run time. Hence, it was inappropriate as a DPI function due to frequently altered malicious signatures. In this work, we introduce Reinhardt, a real-time reconfigurable hardware architecture for REM. Reinhardt represents regex patterns as a combination of reconfigurable cells in hardware and updates regex patterns in real-time while providing high performance. We implement the prototype using NetFPGA-SUME, and our evaluation demonstrates that Reinhardt updates hundreds of patterns within a second and achieves up to 10 Gbps throughput (max. hardware bandwidth). Our case studies show that Reinhardt can operate as NIDS/NIPS and as the REM accelerator for them.

CCS CONCEPTS

• Security and privacy → Intrusion detection systems; Hardware security implementation.

KEYWORDS

Deep Packet Inspection, Pattern matching, Regex, Hardware

ACM Reference Format:

Taejune Park, Jaehyun Nam, Seung Ho Na, Jaewoong Chung, and Seungwon Shin. 2021. Reinhardt: Real-time Reconfigurable Hardware Architecture for Regular Expression Matching in DPI. In *Annual Computer Security Applications Conference (ACSAC '21)*, December 6–10, 2021, Virtual Event, USA. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3485832.3485878>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://permissions.acm.org).

ACSAC '21, December 6–10, 2021, Virtual Event, USA

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8579-4/21/12...\$15.00

<https://doi.org/10.1145/3485832.3485878>

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) that analyzes packet payloads plays a central role in classifying and handling application traffic as well as security (network intrusion detection/prevention systems (NIDS/IPS) [64, 66, 84]). The DPI system in modern networks should satisfy *high performance* and *dynamic updatability* to deal with a large amount of traffic and rapidly-changing networks [4, 7, 20, 72, 74, 79]. Unfortunately, regular expression matching (REM) is considered the major obstacle in achieving them; REM is the essential function of DPI because it enables DPI to search matches in packet payloads with specific patterns efficiently. However, as REM is commonly implemented by Finite-State Machines (FSM), a time-consuming and computationally intensive operation, REM is the major bottleneck in DPI performance. Thus, several trials have been to accelerate REM using hardware, mainly based on Field-Programmable Gate Arrays (FPGA), by matching multiple regex patterns in parallel to support *deterministic high performance* [29, 30, 44, 48, 62, 65, 82].

However, FPGA-based REM raises three critical obstacles in adoption as DPI in practice due to the lack of *dynamic updatability*. First, *updating patterns* in FPGA takes a significant amount of time. Any pattern change requires a long compilation (i.e., synthesis, map, placement, and routing), which may take at least a couple of hours. Second, while updating, *service interruption* is inevitable for initializing the device, which exposes a network to potential threats, and which impedes service availability. Lastly, the update has to perform in an *all-or-nothing* fashion. Even a tiny pattern change requires the entire recompilation processes and service interruptions to commence; the on-demand update of patterns is a burdensome task. FPGA-based REM has these challenges due to difficulty in update tasks, and therefore not very suitable for NIDS/IPS where signatures should be frequently updated to respond against emerging attacks. Also, while this issue in FPGA-based REM has been pointed out for years, it still remains as a significant unsolved limitation [14, 39, 75, 79].

To grant dynamic updatability to FPGA-based REM, we propose a real-time reconfigurable hardware architecture for high-performance DPI, Reinhardt. We shift the paradigm of regex pattern matching on FPGA from a circuit level to a logic level; Reinhardt consists of *reconfigurable cells* that can change their connections

in real-time. The combination of the cells will implement FSM corresponding to given regex patterns by our conversion algorithm. Therefore, the deployment and modification of regex patterns perform fast and dynamically without the long compilation and service interruptions. Furthermore, applying this updatability, Reinhardt stores the information of cell connections in memory and can dynamically fetch them by swapping pattern sets in processing, allowing a packet to be inspected with multiple patterns continuously.

We implement a Reinhardt prototype using NetFPGA-SUME [52, 85]. Our prototype supports 1.4-10 Gbps throughput with 800-160 regex patterns, respectively. The updating time is less than one second without service interruption, outperforming today's FPGA-based solutions. Also, comparing with DPDK-Hyperscan [35, 78], Reinhardt has competitive benefits in providing stable performance, enabling deterministic processing. Our case studies on NIDS/IPS and Snort IDS acceleration using Reinhardt give an intuition on how to leverage the unique strengths of Reinhardt as high-performance security services. In particular, Reinhardt NIDS/IPS covers up to 87% of signatures in Snort 2.9.7 default rules (6,411 signatures), and the hardware acceleration improves the overall throughput up to 65 times the original performance.

2 BACKGROUND AND MOTIVATION

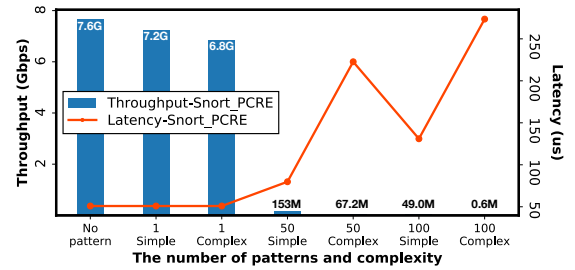
Regular expressions (regex) are helpful to structurize a string that contains a set of specific patterns (e.g., attack signatures) with *metacharacters* having a special meaning for literal characters. Thus, regex matching (REM) becomes one of the most critical functions in DPI to search for one or more matches of specific patterns in an observation string (i.e., packet payload). This section presents the performance degradation in DPI caused by REM and discusses the challenges of previous efforts for accelerating the matching performance by parallelizing the matching process using FPGA.

2.1 Performance Degradation in REM

REM generates an equivalent finite-state machine (FSM) for a given regex pattern and drives the state machine on an observation string. Unfortunately, traversing an FSM is time-consuming and memory-intensive work because it has to inquire state transitions over its state graph on each input character from an observation string. Also, as the complexity and number of regex patterns increase, REM requires more memory access to read the string and continuously traverse multiple state graphs. The complexity of its matching process leads to significant performance degradation.

We evaluate how REM severely causes performance degradation by conducting a microbenchmark through the PCRE engine [28] of Snort 2.9.7 [64], one of the most popular regex engines. For this, we randomly select regex patterns from the default Snort rule-set (i.e., select patterns having "pcre" option), and those randomly selected rules contain 1.6 (noted as Simple) and 7.6 (noted as Complex) metacharacters on average, respectively. This evaluation is conducted on Intel Xeon E5-2630, and the input traffic is generated by Intel DPDK-Pktgen [35].

Figure 1 illustrates the performance variations with different complexity and number of regex patterns. We dramatically see throughput degradations as the number of regex patterns increases; With 50 simple and complex rules, the throughputs become 153 and



* Simple and complex regex patterns contain 1.6 and 7.6 metacharacters on average.

Figure 1: PCRE Performance variations in Snort IDS

67 Mbps, respectively, dropping up to 97.9% and 99.1% compared to No pattern. With 100 rules, the throughputs are no longer viable. While the throughput is declining, the latency increases from 50 μ s to 273 μ s, 5.5 times the baseline with the number of patterns and complexity. These performance degradations mostly come from frequent state transitions along input strings (packet payloads) and metacharacter operations, incurring heavy memory accesses. Also, as the number of rules increases, the impact of the state transition overhead gets accumulated. These results conclude that REM is a bottleneck and should be improved for practical deployments.

2.2 Accelerating REM via Hardware (FPGA)

As depicted in Figure 1, REM suffers from the performance issue. Thus, to improve the performance of REM, prior studies have suggested some hardware accelerated approaches using FPGA [29, 30, 44, 48, 62, 65, 82]¹; FPGA-based REM situates the state machines of regex at a circuit level (H/W) and works massively parallel by exploiting the natural parallelism of hardware, and can be directly connected to network interfaces (i.e., bump-in-the-wire). Thus, they can handle incoming packets in a constant time regardless of the number of patterns and without an unnecessarily cumbersome procedure such as copying to memory, CPU, and applications, guaranteeing *deterministic performance* [12, 16, 24, 70]. Thus, FPGA-based solutions are more welcomed for mission-critical systems and time-sensitive networks [23, 27, 42, 60].

For regex processing, FPGA-based solutions mostly adopt non-deterministic finite automata (NFA) rather than deterministic-finite automata (DFA) [62, 79]. It is because 1) parallel processing in hardware allows concurrent access to multiple states, efficiently handling non-deterministic states, and 2) DFA requires much larger space than NFA while the space is one of the sensitive issues in FPGA due to its limited resource. Thus, we will describe this paper based on NFA-based REM.

2.3 Challenges in FPGA-based REM

Although the FPGA-based approaches improve the performance in DPI of REM, real networks often hesitate to adopt them as a critical function of DPI because of the *limited flexibility* on the FPGAs tied to specific implementations [14, 39, 75, 79]. Here, we discuss three critical challenges of FPGA-based REM.

¹Note that, in this paper, FPGA-based approaches mean only a circuit-based approach, not including memory-based approaches like [2, 8, 11, 68]. Because the memory-based one features sequential processing, it does not fully support massively parallel processing [10], i.e., out of our scope.

Regex Engine	# of Patterns	Time (hh:mm:ss)
Sourdis <i>et al.</i> [65]	1,504	4:53:50
Bisop <i>et al.</i> [10]	310	0:45:49
Johnson & Mackenzie [38]	200	1:38:57
Ganegedara <i>et al.</i> [25]	760	1:52:00

Table 1: Compilation time on previous FPGA-based REM

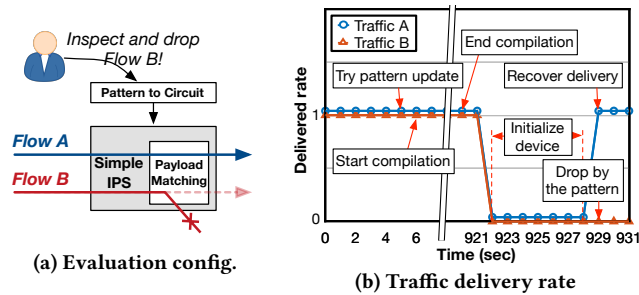


Figure 2: Update response time of the simple H.W.-based IPS

C1. Long compilation time: Unlike software-based solutions, FPGA-based solutions require the compilation process (i.e., synthesis, mapping, placement, and routing) to update regex patterns at a circuit level. It takes several hours, presenting difficulty in applying specific regex patterns into FPGA without long delay [40, 41, 46, 75]. Table 1 shows the compilation time of existing solutions [10, 25, 38, 65]. While it is difficult to directly compare their update times due to different design goals and the complexity of target regex patterns, we can see that updating regex patterns is a time-consuming task.

C2. Inevitable service interruption: After the compilation process, FPGAs require a system halt due to the initialization with updated regex patterns. During this time, the device will be interrupted from several seconds to a couple of minutes. In terms of serviceability, networks are temporarily unavailable during the initialization, increasing an operational burden.

C3. All-or-Nothing update operation: FPGA-based solutions should update new patterns in an all-or-nothing fashion since regex patterns deployed in FPGA are statically fixed at a circuit level [40, 41, 46, 75]. Even a tiny change in regex patterns requires the entire compilation process and initialization of FPGA. It enforces the pattern updates regularly (stacking updates and applying them at once) instead of actively applying the patterns on demand.

We demonstrate these challenges with a simple evaluation. As shown in Figure 2a, while Flow A and B pass through a simple FPGA-based IPS, we try to deploy a new pattern, which inspects packet payloads and drops Flow B at 5 seconds, and measures the time when the IPS blocks flow B. The hardware platform is NetFPGA-SUME [52], and the pattern was compiled using Vivado 2016.04 on Xeon E5-2630. As we can see in Figure 2b, it spends about 15 minutes only to compile the *single* pattern. Moreover, while installing the newly compiled pattern, the IPS stopped about 10 seconds for device initialization so that Flow A is also dropped unintentionally. At last, all update procedures are completed after 924 seconds, and the pattern is working properly to filter Flow B only.

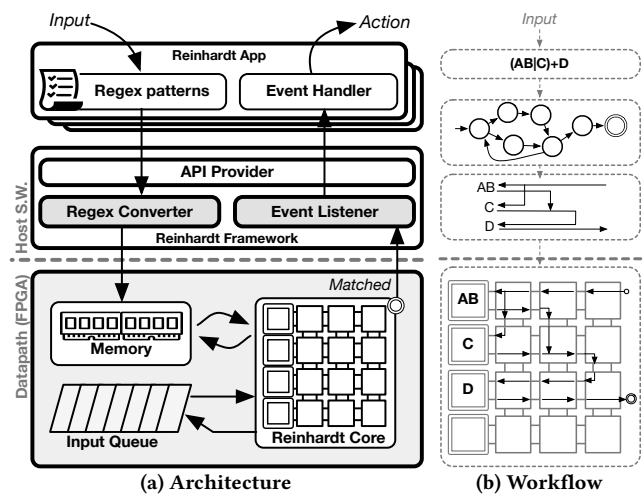


Figure 3: Overall Design of Reinhardt

2.4 Near Real-time Rule Update in DPI

It has not been a serious concern to update DPI rules (for pattern matching) in near real-time. We can stop or delay the operation of a hardware/software system for DPI services, upload newly compiled DPI patterns to the system, and relaunch the system after any update. This delayed operation was not a big problem so far.

However, the rapid increase in malware also drives the need for real-time updates in DPI as NIDS/IPS to detect known attack patterns (i.e., signatures) in the packet payload. According to the statistics of common vulnerabilities and exposures (CVE) [1, 49], more than 30 new vulnerabilities are registered on their index every day. Hence, to keep security up to date, we should update DPI rules (i.e., signatures) that can detect those vulnerabilities immediately; Indeed, in the update history of signatures [18], one or two updates occur every day, and also updates for critical threats sometimes occur multiple times within hours. For this reason, previous works [4, 7, 20, 72, 74, 79] as well as many eminent security articles [13, 17, 22, 33, 61, 67] address *dynamic updatability* as the main aspects to consider for NIDS/IPS.

Moreover, these days, as the development of network technology (e.g., 5G/6G) enables many things to connect to a network, numerous systems and services are provided over the network and cloud infrastructure, and DPI is not an option but plays a central role in handling heterogeneous protocols from the diverse systems/services [5, 21, 32, 54, 59]. Among them, since it contains mission-critical and time-sensitive services where network failure can cause catastrophic consequences such as automotive, smart factory, healthcare, and smart sensors, the network should guarantee reliable communication without loss [3, 19, 26, 47, 56, 69, 73].

3 DESIGN

Our design principles for a dynamic updatable FPGA-based REM are twofold. First, the compilation process must be minimized to apply new and updated regex patterns dynamically. Second, regex patterns in FPGA must be updated without service interruption. Considering these factors, we propose a novel reconfigurable FPGA architecture, called Reinhardt, that introduces new FPGA blocks

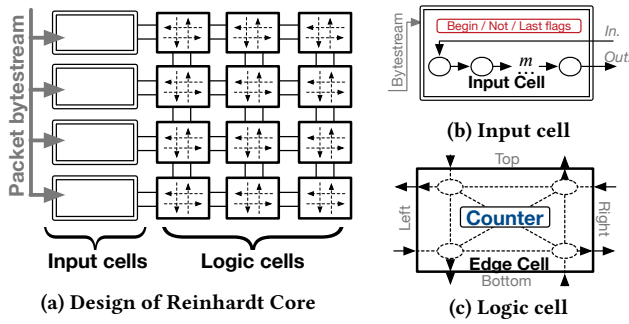


Figure 4: Design of Reinhardt Core

called reconfigurable cells; It transforms given regex patterns represented in NFA to a composition of the reconfigurable cells, allowing the regex patterns to be deployed into hardware in real-time with no service interruption. Their configurations are made by a host software so that administrators can update specific regex patterns on-demand, on-time in a programmable way.

Figure 3 shows the design of Reinhardt and its workflow. Reinhardt is mainly composed of two parts: datapath in hardware (FPGA) and a software framework on the host side. Three sub-components (i.e., core, memory, and input queue) process an observation string in the datapath. More specifically, the Reinhardt core consists of a set of reconfigurable cells that are connected in a $w \times h$ grid topology with input/output ports for each direction, *Top-Bottom-Left-Right*. Each cell can dynamically determine the output directions of the input signals in accordance with Reinhardt configurations. In the software, a regex converter and an event listener manages and controls the Reinhardt datapath. The framework provides APIs which allow Reinhardt applications to update regex patterns and receive messages from the datapath.

The most significant advantage of Reinhardt is to update regex patterns in real-time onto hardware without any service interruption. An update includes all actions related to managing regex patterns, such as 1) *deploying new patterns* and 2) *modifying/removing parts of a previously deployed pattern*. The key idea of Reinhardt is that a target regex is directly represented by state machines in the Reinhardt core through a combination of the cell connections.

As seen in Figure 3b, when given regular expressions, the corresponding NFA structures are represented as the combination of the reconfigurable cells by the Regex Converter. The conversion results are stored in the memory to change the input/output directions of the cells in the core; thus, the equivalent state machines for the regex patterns are implemented in real-time. The current design of Reinhardt can support all common metacharacters. An observation string is inspected by driving the NFA logic in the core, and if any matched pattern is found, the Reinhardt core sends a notification to the event listener on the host. Then, the event listener notifies applications that utilize the event handler and take actions according to the matched result, e.g., blocking suspected traffic or updating the list of patterns to strengthen inspection.

3.1 Reinhardt Core

To understand how Reinhardt implements NFA in FPGA, we view the structure of the Reinhardt core. Figure 4 illustrates the $w \times$

h Reinhardt core. It consists of two kinds of reconfigurable cells; the 0th column of the grid is composed of input cells, and the rests are logic cells. Each cell is connected to neighbor cells by the input/output ports for each direction *Top-Bottom-Left-Right*, and the position of each cell is expressed in (x, y) coordinates. On the left side of the core, the character input bus is connected to the input cells, and the ϵ -signal bus and accept-signal bus on the other side are connected to the logic cells.

For regex pattern matching, a given regex pattern is converted into NFA in the core by combining the abovementioned cells. The pattern is split into substrings and metacharacters, and the substrings are placed in the input cells while the respective connection of logic cells represents their relation (i.e., operation by metacharacter). For clarification, a regex pattern of `abc|xyz` would be separated into substrings `abc` and `xyz`. The metacharacter `|` would be expressed by the connection of the logic cells afterward.

Input cell (Figure 4b): The input cells represent states, and they are activated or transitioned via the input/output ports connected to the logic cells on the right side of the input cells. The input cells are the linear-chained automata states of length m , which works as a simple m string matcher. An input cell starts comparing observation characters when their state is activated and executes a state transition to the output port when all characters match. Input cells can function as a simple string matcher, but they can compare observation characters with a min-max range. For example, to find a character between `'a'` and `'f'` (i.e., a bracket expression `[a-f]`), the min target character is set as `'a'`, and the max target character is set as `'f'`. Also, the input cell can take flags about the observation character to indicate the beginning and end of the string or not contained condition (i.e., `^`, `$` and `[^]`).

Logic cell (Figure 4c): It serves as the directed edges connecting the states in state machines. They function by forwarding input signals (i.e., state transition) from each direction to designated directions by opening/closing internal gates and switches. In addition, the logic cells include a counter that can measure how many times a state transition occurs in its region, allowing logic cells to implement states where the condition has to be met a certain number of times, helping express interval operators `{m, n}`.

Core I/O: There are three input/output buses to drive NFA logics in the core; 1) The character input bus delivers each character in the observation string (i.e., packet bytestream) sequentially to all input cells every clock; thus, all NFA logics implemented in the core will work simultaneously. 2) The ϵ -signal indicates the start of the NFA by triggering the initial state (i.e., the input cell for the first substring of a given regex pattern) through the logic cells. 3) The accept-signal notifies the end of the NFA by triggering the final state (i.e., the input cell for the last substring). This signal is sent when the observation string matches the given regex pattern.

3.2 Converting Regex to Reinhardt Cell Logic

We follow Thompson's algorithm to transform a regex pattern to its NFA [71]; a given regex pattern is split into its constituent subexpressions (i.g., literal characters and metacharacters) and converted to partial NFAs. The concatenation of the partial NFAs constructs the complete NFA.

NFA snippet	Logic Template	NFA snippet	Logic template
$[0-9]$	(Match to min/max range)	$A\{m,n\} / A\{m\} / A\{m,\}$	(With a counter)
A		$[^a-z]$	
	(With 'begin' flag)		(With 'not' flag)
$A\$$		A^*	
	(With 'last' flag)		
A^+		$A?$	
$A B$		$(AB)(CD)$	

Table 2: Templates for Regex Pattern to Reinhardt Logic

Algorithm 1: Regex Pattern to Reinhardt Cell Logic

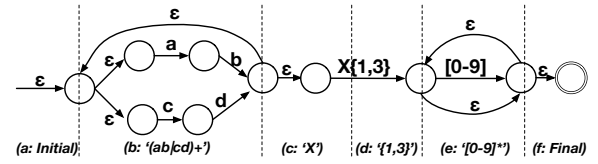
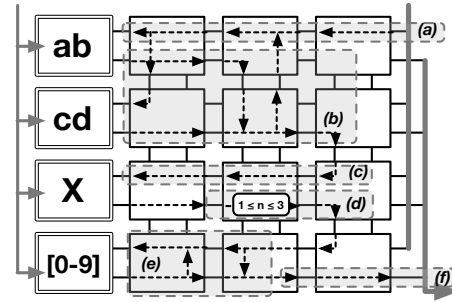
```

Input: start_row, Start row in the Reinhardt core
Input: regex, Given regex pattern
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 )

```

To implement NFA in the Reinhardt core, the Reinhardt software enforces the cell configurations to construct the NFA logic similarly to Thompson's algorithm. The subexpression is classified as substrings and metacharacters, and each subexpression is *templated*, representing their partial NFA structures by a combination of the cells as described in Table 2. Connecting the templates builds up the more extensive NFA logic recursively in the core.

Algorithm 1 describes this operation. It first takes the number of a start row in the core to place a generated NFA and a target regex pattern to deploy and initializes a stack that stores the input/output coordinate and its direction (i.e., *Top-Bottom-Left-Right*) for the last generated partial NFA(s) so far. Then, the given regex pattern is converted to the postfix form to reflect the precedence in which the partial NFAs are generated and parsed by reading the postfix sequentially. 1) The substrings are placed in the input cell on each row, and whenever input of one substring is completed, one substring is considered as one small NFA so that the input/output coordinates

(a) NFA State Diagram for $(ab|cd)^+X\{1,3\}[0-9]^*$ 

(b) NFA on the Reinhardt core

Figure 5: Deployment of Regex Pattern to Reinhardt

of its input cell are stored in the stack (In here, the I/O port of the input cell is always the right side, so the directions are fixed at R). 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 operation is recursively performed, the entire NFA is completed by connecting the ϵ and accept signals to the last stacked NFA. Finally, this result is transferred and stored to the hardware memory.

3.3 Pattern Deployment Breakdown

Figure 5 shows the deployed Reinhardt logic for regex $(ab|cd)^+X\{1,3\}[0-9]^*$ on the 4×4 Reinhardt core from Algorithm 1 (See Appendix A for the detailed steps). Comparing its state diagram to the configured logic (Figure 5a and 5b), the edges of the state diagram are represented by the logic cells in the same shape, and the characters chained in linear are represented by the input cells. Each subexpression is mapped one to one, resulting in the Reinhardt core consequently implementing the equivalent state machine; starting from the ϵ -signal of (a) in Figure 5b, the state of the observation string is transitioned through each section until (f). This accept-signal notes that the observation string matches the regex pattern and that it is accepted. Conversely, removing a pattern is done simply by initializing the involved cells. New patterns can use the initialized spaces; the modification of a deployed pattern is made by generating new logic for the modified pattern from the algorithm, releasing the previous logic placing the new logic at that place.

Here, We note that the size of the Reinhardt core in this example is only 4×4 with one regex pattern, but it is only scaled down for clear understanding. The actual core size is larger and can represent a more complex and more number of regex. Generally, as the width w of the core increases, more metacharacters can be represented (i.e., a more complex regex pattern), and as the height of the core h increases, more regex patterns can be expressed and matched simultaneously. The size of the core is closely related to available FPGA resource, and the details are covered in §5.

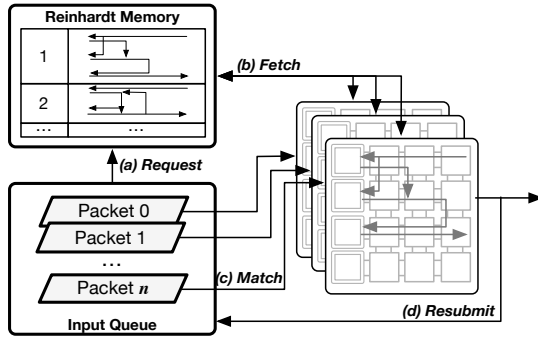


Figure 6: Reinhardt datapath components and resubmitting

3.4 Memory and Input queue

Memory: The memory manages cell configurations and their state transitions of the core. By exploiting dynamic updatability, Reinhardt can store multiple Reinhardt logic for the core (i.e., cell configurations) into the memory with an ID and fetch one of them to the core instantly, enabling *seamless* updates. Instead of modifying the logic on the core directly (certainly, it is also allowed), new Reinhardt logic for new patterns is first stored into the memory and swapped entirely with the currently active logic in the core. It ensures full availability of zero delays in updating and allows regex patterns to be provided by switching them per group (i.e., different IDs on the memory) for each observation string, e.g., regex for HTTP or regex for malware.

Multiple queues and cores: To process an observation string with NEA of Reinhardt logic, each character in the string has to be sequentially entered in Reinhardt. However, this procedure delays the next observation strings until the observation string currently being processed is finished, causing the throughput degradation. To improve the throughput, as seen in Figure 6, we design the input queue to n multiple queues and the Reinhardt core to be also multi-layered, mapping one to one to each queue. Therefore, multiple packets can be processed concurrently, and the throughput can increase as much as the number of the queues and cores n increases. However, as more queues and cores proportionally require more FPGA resource in implementation, it is important to find the optimal number n . It is covered in Evaluation (§5).

By combining the two features of the memory fetching and multiple queueing, Reinhardt can process an observation string multiple times with different regex patterns back-to-back [57]. Figure 6 shows the steps of this resubmitting feature. Step (a) involves the requesting of a memory ID set to match. Next, the logic from the memory ID is fetched and deployed onto the Reinhardt core (b), and (c) shows the actual match of the observation string. The final step (d) resubmits the observation string into the input queue for the following pattern of the memory ID, and the process repeats. This resubmitting allows more regex matching beyond the core size and establishes a method for hierarchical processing on regex sets.

3.5 Host-FPGA communication

Handling detection: 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 applications that register the event handler at the event listener to follow up on the received matching result. For example, we can implement an automated system that blocks malicious IP addresses from matching results.

Datapath configuration: The host configures the Reinhardt datapath through two APIs mainly, 1) `setCell(ID, x, y, args)`, which configures each cell, and 2) `setFetch(ID1, ID2)`, which specifies memory IDs used for matching; 1) `setCell()` takes the *memory ID* to store this cell configuration, *cell coordinate*, and *argument* to designate a cell instruction. If a ‘x’ is set to zero indicating the input cells, its argument needs a target substring of regex, otherwise the argument needs signal directions, e.g., ‘t→b’ meaning a signal from the top is forwarded to the bottom. Reinhardt logic derived from Algorithm 1 is also deployed into the Reinhardt datapath by repeatedly calling this `setCell()` for each cell. 2) `setFetch()` takes *preceding ID* and *following ID* for resubmitting, e.g., if ID 10 should be performed after ID 5, set as `setFetch(5, 10)`. There are also constant IDs to denote first/last rounds, ‘INIT’ and ‘LAST’, e.g., `setFetch(INIT, 5)` means the memory that is fetched first when matching starts is 5, and `setFetch(10, LAST)` means ID 10 is the last one for the matching.

4 IMPLEMENTATION

To validate Reinhardt’s design, we implemented a prototype using NetFPGA-SUME, an FPGA-based PCI Express board with Xilinx Virtex-7 XC7V690T and four SFP+ 10 Gbps interfaces [52, 85], and it processes packets in chunks of 256-bit at 160 MHz. We also implemented a device driver based on the NetFPGA-SUME reference driver [53] to handle the prototype with the APIs.

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 limited hardware resource. To set the constraints, we collected 2,735 regexes 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 draw the constraints that can express 90% of regex forms and accommodate as many patterns as possible with Reinhardt. Please note that these constraints are statistically specified values to *represent generic patterns*, and they can vary depending on requirements.

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 all of the given regex patterns, and Figure 7a shows its frequency distribution in the regex patterns. From the result, when $w = 24$, Reinhardt can cover 90% of regex patterns regardless of the ruleset choice.

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

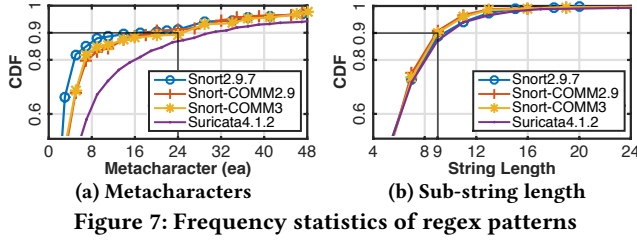
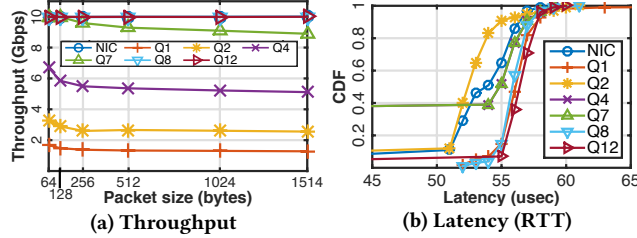


Figure 7: Frequency statistics of regex patterns

Figure 8: Performance variations by number of queues n

an optimal length, we examine the distribution of the lengths of substrings in the given regex patterns. As shown in Figure 7b, 90% of the substrings have up to 9 characters, i.e., $m = 9$.

Core height and the number of queues: While the constraints mentioned above 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 in FPGA (i.e., LUTs) are utilized.

As a result, we implement *four* Reinhardt datapaths with different core size as $24 \times 160 \times 8$, $24 \times 300 \times 4$, $24 \times 665 \times 2$ and $24 \times 1580 \times 1$ (Width $w \times$ Height $h \times$ 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.

5 EVALUATION

5.1 Performance Measurement

We measure the throughput and latency variations of Reinhardt to see how many queues are required to get the line-rate performance (i.e., 10 Gbps) and to see the performance degradations due to resubmissions. For this, 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 other two are used as a packet generator and its receiver using Intel DPDK-Pktgen [35] and nping [55] for throughput and latency measurements, respectively. As criteria, we also measure the performance of a direct connection between the hosts without Reinhardt.

Performance by number of queues: Figure 8a shows the throughput variations of Reinhardt under the different number of queues. The throughput with a single queue is from 1.68 to 1.28 Gbps, but as the number of queues increases, the overall throughput increases, achieving the line-rate starting from *eight* queues.

This required number of queues to achieve 10 Gbps can be proved arithmetically. We have implemented 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, i.e., the delay of 32 clocks takes 200

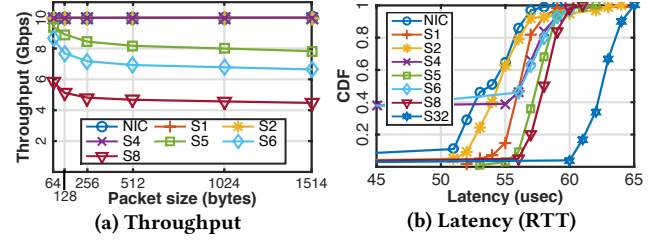


Figure 9: Performance degradations due to resubmitting

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.

Theoretically, the processing time of Reinhardt takes 32 clocks of delay, so there should be more latency in Reinhardt compared to NIC connections. However, Figure 8b shows that latencies of Reinhardt are in very close proximity to the latency of the NIC connections regardless of the number of queues because the latency increase is a negligible amount in the unit of nanoseconds, whereas the unit of latency in the figure is microseconds. By using Reinhardt, packet transmission can be guaranteed with a latency of 60 μ s, regardless of the number of queues.

Performance degradations by resubmitting: We measure the throughput and latency variations of Reinhardt to see the degree of performance degradations under the different number of resubmissions. For this, we configure Reinhardt with *eight* queues. As shown in Figure 9a, the throughputs with up to 4 times submissions are steady. However, as the number of submissions increases, 20% of throughput degradations occur every resubmission. Latency is slightly different, but most of them fall within the error range of measurement and are still under 60 μ s because there is a delay of only about 200 ns per round, so even if 4 resubmits occur, the delay is less than 1 μ s. As the number of submissions increases, the delay gradually accumulates; 32 resubmissions increase the overall latency by about 6-8 μ s.

These results are because the dynamic configurability 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.

5.2 Regex Pattern Deployment

Pattern capacity: Pattern capacity means how many regex patterns Reinhardt can accommodate at once without performance loss. However, it is difficult to make universal claims since Reinhardt aims at the dynamic configuration for various regex patterns rather than fixed regex patterns, and the complexity of the patterns varies the number of patterns. Thus, we randomly select regex patterns among the regex set used to determine the core constraints in §4 until the core becomes full, including the four resubmitting. The experiment was repeated 100 times; Figure 10 describes the number of deployed regex patterns for the different core sizes into a candlestick graph of the body as the range of standard deviations

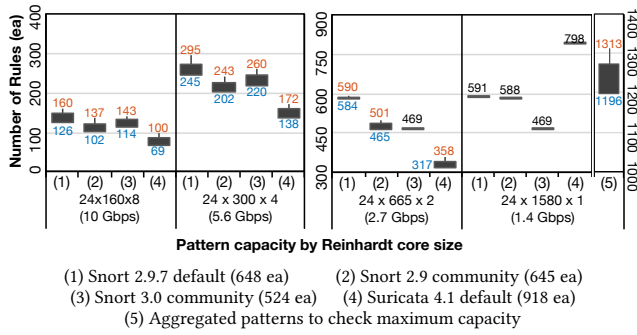


Figure 10: Pattern capacities with the different core sizes

Core size $w \times h \times n$	# of the cells	# of patterns	Time (sec)
24x160x8	15,360	≤ 160	0.116
24x300x4	28,800	≤ 295	0.186
24x665x2	63,840	≤ 590	0.403
24x1580x1	151,680	≤ 1313	0.965

* The number of the cells to configure considers four submissions
 * The number of patterns is referenced in Figure 10

Table 3: Reinhardt cell configuration time

around the average and its shadow as a min-max. The number of deployed patterns increases 160-1313 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 (i.e., worst-case); Table 3 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. With the consideration of the number of deployable regex patterns shown in Table 10, the configuration time takes 116 ms for 160 patterns and 965 ms for 798 patterns, which is much faster than the configuration times in existing FPGA-based REM described in Table 1. The configuration in Reinhardt is mostly spent in communication between the datapath and software. The update is instantly performed at the device.

Update response time in NIDS/IPS: To validate how effectively Reinhardt addressed the challenge of FPGA-based DPI, we back to our motivating example of Figure 2 in §2.3 and perform the same evaluation with Reinhardt. Figure 11 shows its result. As installing the new pattern to inspect and filter Flow B, the new pattern works instantly while the device is up and running as ever. Hence, unlike the motivating example, Flow A is delivered continuously, but only Flow B is dropped immediately after updating.

5.3 Comparison with DPDK-Hyperscan

Intel DPDK-Hyperscan [35, 78] is one of the best-of-breed baselines for fast regex processing running with a multi-core CPU. Here, we analyze the advantages and disadvantages of Reinhardt through comparison with DPDK-Hyperscan. We implement a simple DPDK-Hyperscan application using the Hyperscan open-source [34] and DpdkBridge [58] that receives packets from network interfaces and

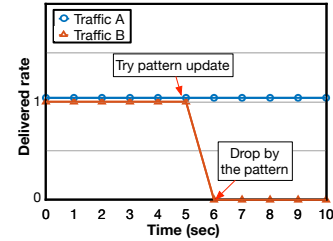
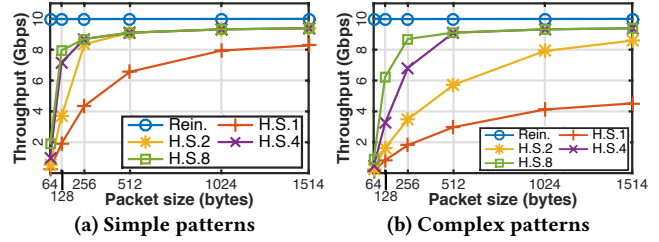


Figure 11: Update response time of the Simple IPS with Reinhardt (See with Figure 2 in §2.3)



* The core size of Reinhardt is 24x160x8 // H.S.n means Hyperscan with n cores
 * Simple and complex patterns contain 1.6 and 7.6 metacharacters on average.

Figure 12: Throughput for 100 patterns (vs Hyperscan)

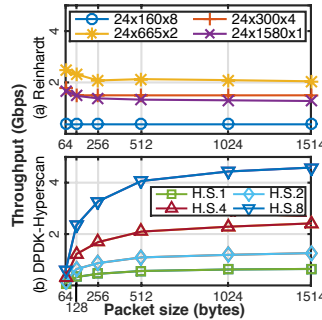


Figure 13: Throughput for 843 patterns (vs Hyperscan)

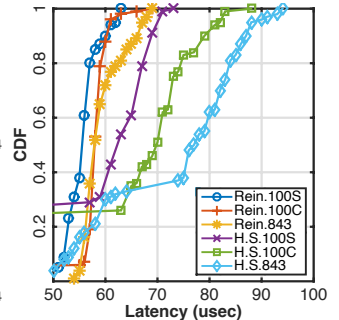


Figure 14: Latency (RTT) for 100 and 843 patterns (vs Hyperscan)

matches them to target patterns. It also runs on Intel Xeon E5-2630 (10 cores, Hyper-Threading disabled), 64 GB of RAM and Intel X520 10GbE NICs. The target patterns used 843 compatible to Reinhardt of 847 pcrs from Intel’s sample data [36].

Throughput in capacity: To compare performance within the Reinhardt capacity, we randomly select 100 regex patterns among the 843 patterns into simple and complex cases respectively and measure processing throughputs with Reinhardt 24x160x8 and DPDK-Hyperscan, respectively. We repeat 20 times, and Figure 12 shows its average values; Reinhardt constantly achieves 10 Gbps regardless of complexity, while DPDK-Hyperscan not only shows performance degradation according to packet size and pattern complexity. Four cores for simple patterns and eight for complex patterns are required to extract the maximum performance with DPDK-Hyperscan, but performance degradation is still observed in the cases of 64-256 bytes packets, and overall throughput is up to 9.3 Gbps, slightly below the line rate (i.e., 10 Gbps).

Throughput in excess of capacity: To measure performance on a larger pattern set, we deploy all 843 patterns into each different size of Reinhardt, taking into account the excess of the number of resubmissions ensuring maximum speed (i.e., 4 times); Each core $24 \times 160 \times 8$, $24 \times 300 \times 4$, $24 \times 665 \times 2$ and $24 \times 1580 \times 1$ can accommodate all patterns by taking total 19, 10, 5 and 2 submission rounds respectively. Figure 13 presents their throughput averages for 20 iterations; Reinhardt with the core sizes $24 \times 160 \times 8$ and $24 \times 300 \times 4$ suffers significant performance degradation from 10 / 5.6 Gbps to 0.37 / 1.5 Gbps respectively due to processing delay from too many resubmissions. On the other hand, since the core $24 \times 665 \times 2$ and $24 \times 1580 \times 1$ can handle all patterns within the capacity or with only one more submission, they almost preserve the original throughput. While Reinhardt shows the similar performance of Hyperscan with 1-4 cores, Hyperscan can perform well with more cores (e.g., Hyperscan approaches 4.5 Gbps with 8 cores).

Latency: We compare the latencies of Reinhardt and the DPDK version of Hyperscan while handling 100 simple/complex patterns and 843 patterns. As seen in Figure 14, the latency of Reinhardt is almost similar regardless of the complexity and number of patterns because the overhead by 19 resubmissions is arithmetically only about 4 us in Reinhardt, it is reasonable to arrive at similar results within an error bound of measurement. In DPDK-Hyperscan, its latency is slower to 50% as the complexity and number of patterns increase, and the variation (i.e., jitter) becomes wide.

Discussion: This evaluation shows that Reinhardt guarantees line-rate throughput within the capacity, and if overloaded, there is a decrease but still provides stable throughput and latency regardless of the packet size or patterns. DPDK-Hyperscan also achieves outstanding throughput, moreover, obtains better than Reinhardt when processing a large number of regex as utilizing many CPU cores. However, its throughput and latency are fairly affected by regex complexity and the packet size.

Furthermore, while DPDK-Hyperscan should consume lots of host resources, the matching process of Reinhardt runs standalone on hardware. Hence, we expect that the rest of the resources can be leveraged to facilitate extra services to reduce operating costs as Microsoft’s AccelNet suggested [23].

6 CASE STUDY: NIDS AND SNORT ACCELERATION

To understand how real-world networks benefit from Reinhardt, we implement two security systems applying Reinhardt; NIDS/IPS and PCRE replacement in Snort IDS.

6.1 NIDS/IPS using Reinhardt

Experiment setup: Figure 15 shows the extensions for Reinhardt as NIDS/IPS. Signatures are parsed into headers in the 5-tuple lookup table and corresponding patterns (i.e., the “content” and the “pcre”) are converted to Reinhardt logics in the memory. Here, header and pattern pairs are placed in the same memory ID to consider resubmitting, and the IDs of the corresponding area are assigned to each header. When packets arrive, Reinhardt fetches the matching logic from the memory to the Reinhardt core, and the core performs the inspection. When Reinhardt finds any matches, the

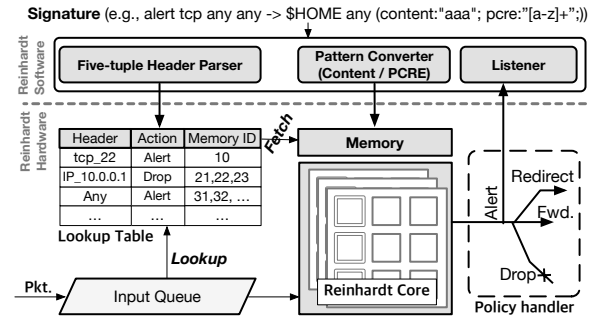


Figure 15: NIDS/NIPS using Reinhardt

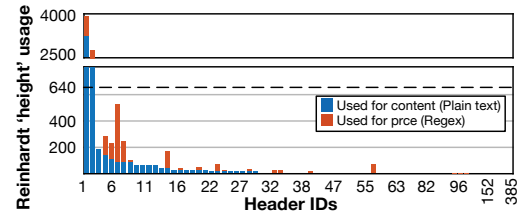


Figure 16: The core ‘height’ usage distribution by headers

policy handler follows the predefined action; *alert*, *drop* or *redirect* the packet to alternative routes (e.g., honeypot).

Rule coverage evaluation: To measure the coverage of Reinhardt NIDS/IPS, we try to deploy Snort 2.9.7 default ruleset (6,411 signatures) on Reinhardt NIDS/IPS with the $24 \times 160 \times 8$ core including 4 times submissions. As a result, the Reinhardt NIDS/IPS accepts 75% of the signatures.

For detailed analysis, we measure the core usage of patterns by the occupied heights per each header (Figure 16). The signatures require a total of 9,510 core heights and it far exceeds the capacity of Reinhardt even considering resubmitting (i.e., $160 \times 4 = 640$). However, they are distributed across 385 headers, and the average height usage for each header is only 138. That is, Reinhardt could accommodate all signatures, except two exceeding headers 1 and 2, by swapping the activated patterns on incoming packets by exploiting the fast-dynamic configurability of Reinhardt.

The two huge headers requiring 3,885 and 2,767 heights, respectively, are general rules for HTTP and SQL, so a lot of similar patterns were indiscreetly stacked from different signatures. While we omit details about it in this paper since signature optimization is beyond the scope of this paper, we could reduce their usage up to 50% by merging duplicates and manually optimizing the patterns. As a result, the Reinhardt NIDS/IPS could accept more signatures to 87%. We expect to accommodate all signatures if they are categorized in a more fine-grained fashion.

Performance evaluation: The performance of the Reinhardt NIDS/IPS is equal to the naive Reinhardt we measured in §5 (i.e., 10 Gbps). In fact, some delays are added to search the lookup table, but this delay is only a few nanoseconds that are virtually hard to measure on the source and destination hosts.

Implication: This result demonstrates that Reinhardt can effectively work as a high-performance NIDS/IPS. An important implication is that the real-time updatability of Reinhardt allows the patterns to be dynamically loaded on the core at the appropriate time so that the total number of active signatures can be much

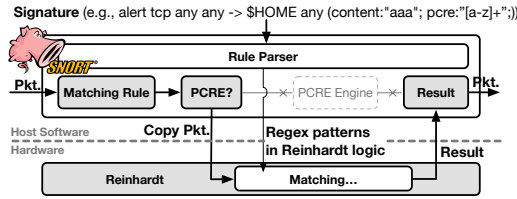
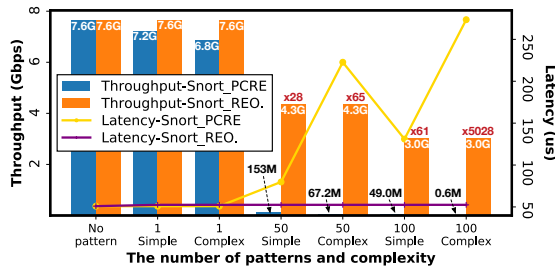


Figure 17: Snort IDS with Reinhardt-based REM



* Simple and complex regex patterns contain 1.6 and 7.6 metacharacters on average.

Figure 18: Performance comparison for Snort acceleration

greater than the capacity of the core. Therefore, if signatures are well established on each header, Reinhardt NIDS/IPS can accept the comparable amount of patterns as software. This advanced feature is difficult to perform with previous FPGA-based REM.

6.2 PCRE Replacement in Snort IDS

In §2.1, we have presented the performance of software-based REM by borrowing Snort IDS. Here, we present Reinhardt as the replacement of the PCRE engine to accelerate the performance.

Experiment setup: Figure 17 shows the overall design of Snort IDS with Reinhardt. We have modified Snort IDS to copy packets to Reinhardt to perform matching on Reinhardt instead of the PCRE engine. A matching result is replaced to take from Reinhardt, not the PCRE engine. Regex patterns in signatures (i.e., the pcre option) are parsed into Reinhardt logic and stored in Reinhardt.

Performance evaluation: Figure 18 shows the performance variations of Snort IDS with Reinhardt. Here, the test environment is the same as that of the PCRE throughput benchmark (described in §2.1). One of the conspicuous results is that Reinhardt provides stable performance (i.e., 7.6, 4.3, and 3.0 Gbps) regardless of the complexity of regex patterns, unlike the PCRE engine. Unfortunately, there are gradual throughput degradations from 7.6 Gbps to 3.0 Gbps as the number of rules increases from 1 to 100, but these throughput degradations mostly come from the hidden overheads in Snort IDS, which are the iterations to check the existences of other rule options (e.g., *offset*, *distance*, and *within* options) for each rule. Therefore, we believe that if the Snort internal procedure can be optimized in consideration of Reinhardt, this degradation can be eliminated. The latency improvement is remarkable. Even if there is slight overhead due to packet copying from software to hardware, it is below a few us negligible at this latency scale of Snort IDS.

Compared to those with the original PCRE engine, the overall throughput with Reinhardt is significantly improved up to x5,028. Even though we ignore 0.6 Mbps on the PCRE engine of 100/Complex, there is a significant performance improvement of up to x65.

Implication: This case study is a kind of hardware acceleration for REM, and it is possible because Reinhardt can immediately reflect software changes. Considering previous FPGA-based matching solutions that only support fixed patterns or take a long time to change, Reinhardt first presents how FPGA can be used as a hardware accelerator for REM; Acceleration for Snort has been usually with GPU [6, 43, 45, 75, 76] owing to its efficient programmability in deploying patterns. However, it should involve critical overhead for copying packet payload from network interfaces to CPU and from CPU to GPU, and for scheduling between the GPU cores [12, 15, 16, 24, 70]. Reinhardt, however, works in bump-in-the-wire, so that has much less loss in performance, particularly latency.

7 RELATED WORK

FPGA-based REM: Sidhu *et al.* [62] proposed a one-hot encoding scheme to express NFA with circuit blocks, and its subsequent studies [30, 44] inspire Reinhardt. Some studies [29, 51, 77] suggested resource efficient regex circuits. Other studies [48, 50, 80, 81] focused on high-performance FPGA-based REM.

Configurable FPGA-based REM: One strategy is generating FPGA source codes (i.e. HDL) from regex patterns automatically [9, 48, 65]. However, compiling the generated HDL cores to FPGA remains and is far from real-time configurability. Memory-based approaches can be configurable [8, 11, 68], and Sidler *et al.* [63] proposed CPU-FPGA hybrid approach. However, as they are memory-intensive, they should work in sequential processing and cannot fully support massive parallel processing, i.e., less performance than circuit-based ones [10, 83]. Also, in security aspects, they cannot support *constraint repetitions* (i.e., *, +), 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 [10, 83]. To the best of our knowledge, Reinhardt is the first work that proposes the real-time reconfigurable REM on FPGA [79].

Programmable-dataplane-based REM: P4 allows a limited syntax in pattern matching. For example, DeepMatch [31] and Jepsen *et al* [37] 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]). Whereas, Reinhardt is specialized in pattern matching, supporting the full regex matching syntax.

8 CONCLUSION

FPGA-based REM satisfies high-performance, but flexibility is a significant limitation as it involves a time-consuming process to update patterns. To address this, we have presented Reinhardt, an improved hardware architecture of implementing regex with its reconfigurable cells to support dynamic updates. Our evaluation and case studies demonstrate that Reinhardt updates patterns promptly without service interruption and serves well 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.

ACKNOWLEDGMENTS


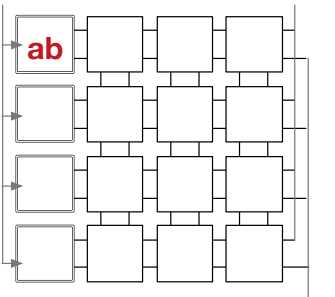
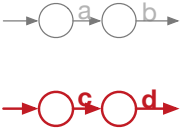
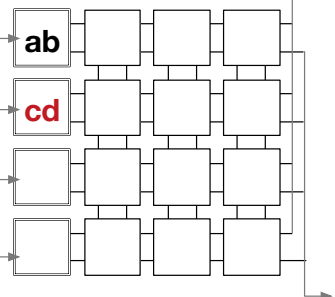
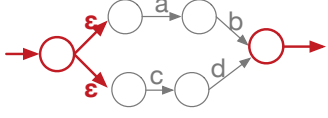
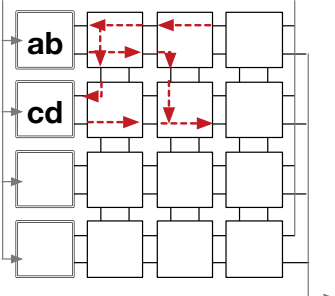
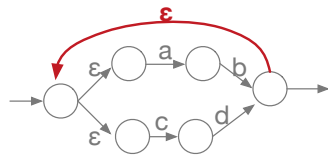
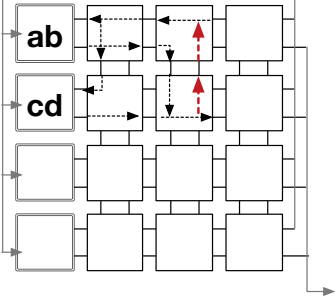
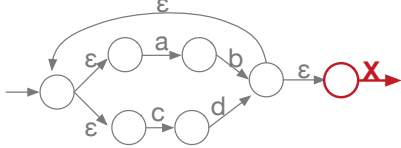
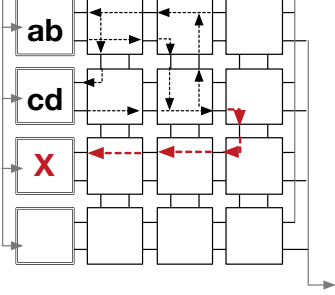
This research was supported by the Engineering Research Center Program through the National Research Foundation of Korea (NRF) funded by the Korean Government MSIT (NRF-2018R1A5A1059921).

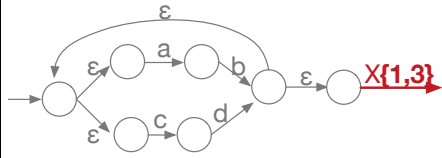
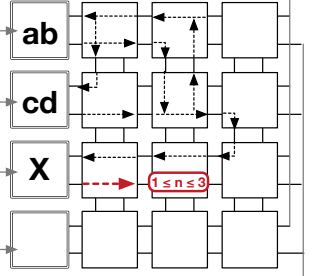
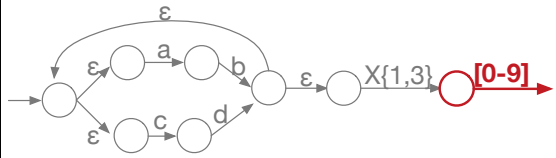
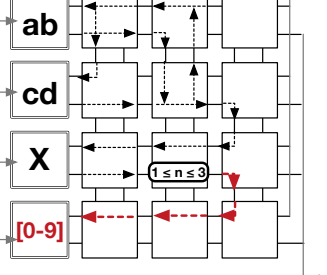
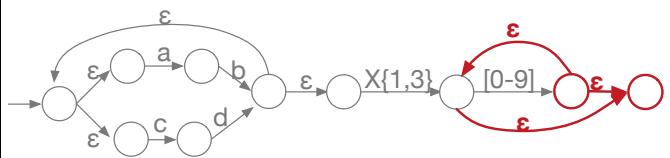
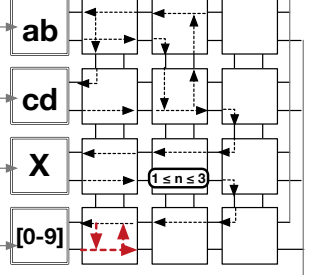
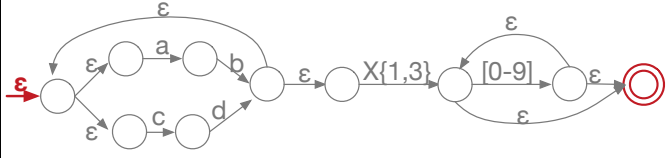
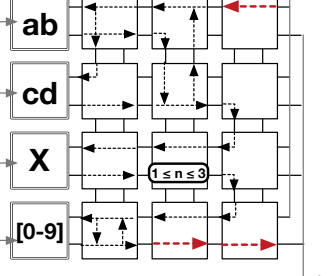
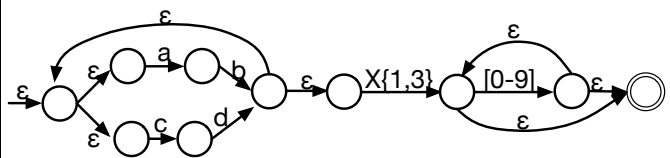
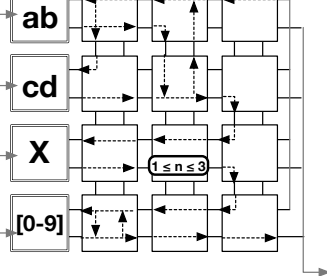
REFERENCES

- [1] 2021. CVE Detail. <https://www.cvedetails.com>.
- [2] 2021. Titan IC RXP. <https://www.mellanox.com/titan-ic>.
- [3] Mohammad Aazam and Eui-Nam Huh. 2015. E-HAMC: Leveraging Fog computing for emergency alert service. In *2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*. IEEE, 518–523.
- [4] Tamer AbuHmed, Abedelaziz Mohaisen, and DaeHun Nyang. 2008. A survey on deep packet inspection for intrusion detection systems. *arXiv preprint arXiv:0803.0037* (2008).
- [5] Patrick Kwadwo Agyapong, Mikio Iwamura, Dirk Staehle, Wolfgang Kiess, and Anass Benjebbour. 2014. Design considerations for a 5G network architecture. *IEEE Communications Magazine* 52, 11 (2014), 65–75.
- [6] Igor M Araújo, Carlos Natalino, Ádamo L Santana, and Diego L Cardoso. 2018. Accelerating VNF-based Deep Packet Inspection with the use of GPUs. In *2018 20th International Conference on Transparent Optical Networks (ICTON)*. IEEE, 1–4.
- [7] Kubilay Atasu, Raphael Polig, Jonathan Rohrer, and Christoph Hagleitner. 2013. Exploring the design space of programmable regular expression matching accelerators. *Journal of Systems Architecture* 9, 10 (2013), 1184–1196.
- [8] Zachary K Baker, Hong-Jip Jung, and Viktor K Prasanna. 2006. Regular expression software deceleration for intrusion detection systems. In *2006 International Conference on Field Programmable Logic and Applications*. IEEE, 1–8.
- [9] Joao Bispo, Ioannis Sourdis, Joao MP Cardoso, and Stamatis Vassiliadis. 2006. Regular expression matching for reconfigurable packet inspection. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*. IEEE, 119–126.
- [10] Joao Bispo, Ioannis Sourdis, Joao MP Cardoso, and Stamatis Vassiliadis. 2007. Synthesis of regular expressions targeting fpgas: Current status and open issues. In *International Workshop on Applied Reconfigurable Computing*. Springer, 179–190.
- [11] Benjamin C Brodie, David E Taylor, and Ron K Cytron. 2006. A scalable architecture for high-throughput regular-expression pattern matching. In *33rd International Symposium on Computer Architecture (ISCA'06)*. IEEE, 191–202.
- [12] Shuai Che, Jie Li, Jeremy W Sheaffer, Kevin Skadron, and John Lach. 2008. Accelerating compute-intensive applications with GPUs and FPGAs. In *2008 Symposium on Application Specific Processors*. IEEE, 101–107.
- [13] 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.
- [14] Hao Chen, Yu Chen, and Douglas H Summerville. 2010. A survey on the application of FPGAs for network infrastructure security. *IEEE Communications Surveys & Tutorials* 13, 4 (2010), 541–561.
- [15] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. 2018. Understanding performance differences of FPGAs and GPUs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 93–96.
- [16] Ben Cope, Peter YK Cheung, Wayne Luk, and Lee Howes. 2010. Performance comparison of graphics processors to reconfigurable logic: A case study. *IEEE Transactions on computers* 59, 4 (2010), 433–448.
- [17] 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.
- [18] Emerging Threats. 2021. Emerging Threats Rulesets. <https://rules.emergingthreats.net>, <https://doc.emergingthreats.net>.
- [19] GSMCEIEG ETSI. 2015. 004, Mobile Edge Computing (MEC) Service Scenarios V1. 1.1,(2015).
- [20] Gilberto Fernandes, Joel JPC Rodrigues, Luiz Fernando Carvalho, Jalal F Al-Muhtadi, and Mario Lemes Proença. 2019. A comprehensive survey on network anomaly detection. *Telecommunication Systems* 70, 3 (2019), 447–489.
- [21] FierceWireless. 2019. Deep Packet Inspection: Getting the Most Out of 5G. <https://www.fiercewireless.com/sponsored/deep-packet-inspection-getting-most-out-5g>.
- [22] FireEye. 2021. FireEye Dynamic Threat Intelligence Cloud. <https://www.threatprotectworks.com/Dynamic-Threat-Intelligence-cloud.asp>.
- [23] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, 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.
- [24] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. 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, 47–56.
- [25] Thilan Ganegedara, Yi-Hua E Yang, and Viktor K Prasanna. 2010. Automation framework for large-scale regular expression matching on FPGA. In *2010 International Conference on Field Programmable Logic and Applications*. IEEE, 50–55.
- [26] Melvin B Greer Jr and John W Ngo. 2012. Personal emergency preparedness plan (pepp) facebook app: Using cloud computing, mobile technology, and social networking services to decompress traditional channels of communication during emergencies and disasters. In *2012 IEEE Ninth International Conference on Services Computing*. IEEE, 494–498.
- [27] PK Gupta. 2016. Accelerating datacenter workloads. In *26th International Conference on Field Programmable Logic and Applications (FPL)*.
- [28] Philip Hazel. 2005. Pcre: Perl compatible regular expressions. *Online* <http://www.pcre.org> (2005).
- [29] Tran Trung Hieu, Tran Ngoc Thinh, and Shigenori Tomiyama. 2013. ENREM: An efficient NFA-based regular expression matching engine on reconfigurable hardware for NIDS. *Journal of Systems Architecture* 9, 4-5 (2013), 202–212.
- [30] Brad L Hutchings, Rob Franklin, and Daniel Carver. 2002. Assisting network intrusion detection with reconfigurable hardware. In *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*. IEEE, 111–120.
- [31] Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. 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*. 336–350.
- [32] Jonghwan Hyun, Jian Li, ChaeTae Im, Jae-Young Yoo, and James Won-Ki Hong. 2014. A VoLTE traffic classification method in LTE network. In *2014 Asia-Pacific Network Operations and Management Symposium*. IEEE, 1–6.
- [33] InfoSecurity. 2019. Advanced Malware Detection - Signatures vs. Behavior Analysis. <https://www.infosecurity-magazine.com/opinions/malware-detection-signatures/>.
- [34] Intel. 2019. Hyperscan. <https://github.com/intel/hyperscan>.
- [35] Intel. 2021. Intel DPDK: Data Plane Development Kit. <http://dpdk.org>.
- [36] Intel 01.org. 2019. Hyperscan sample data. <https://01.org/downloads/sample-data-hyperscan-hsbench-performance-measurement>.
- [37] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. 2019. Fast string searching on pisa. In *Proceedings of the 2019 ACM Symposium on SDN Research*. 21–28.
- [38] Adam Johnson and Kenneth Mackenzie. 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, 126–130.
- [39] Saitesh Kumar. 2007. Survey of current network intrusion detection techniques. *Washington Univ. in St. Louis* (2007), 1–18.
- [40] Christopher Lavin, Brent Nelson, and Brad Hutchings. 2013. Impact of hard macro size on FPGA clock rate and place/route time. In *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE, 1–6.
- [41] Christopher Lavin, Marc Padilla, Jaren Lamprrecht, Philip Lundrigan, Brent Nelson, and Brad Hutchings. 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, 117–124.
- [42] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 1–14.
- [43] Cheng-Hung Lin and Cheng-Hung Hsieh. 2018. A novel hierarchical parallelism for accelerating NIDS using GPUs. In *2018 IEEE International Conference on Applied System Invention (ICASI)*. IEEE, 578–581.
- [44] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. 2007. Optimization of pattern matching circuits for regular expression on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 15, 12 (2007), 1303–1310.
- [45] Cheng-Hung Lin, Chen-Hsiung Liu, Lung-Sheng Chien, and Shih-Chieh Chang. 2012. Accelerating pattern matching using a novel parallel algorithm on GPUs. *IEEE Trans. Comput.* 62, 10 (2012), 1906–1916.
- [46] Andrew Love, Wenwei Zha, and Peter Athanas. 2013. In pursuit of instant gratification for FPGA design. In *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE, 1–8.
- [47] Ning Lu, Nan Cheng, Ning Zhang, Xuemin Shen, and Jon W Mark. 2014. Connected vehicles: Solutions and challenges. *IEEE internet of things journal* 1, 4 (2014), 289–299.
- [48] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. 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, 127–136.
- [49] Mitre. 2021. Common Vulnerabilities and Exposures (CVE). <https://cve.mitre.org/>.

- [50] Hiroki Nakahara, Tsutomu Sasao, and Munehiro Matsuura. 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*. 359–364.
- [51] Hiroki Nakahara, Tsutomu Sasao, and Munehiro Matsuura. 2012. A design method of a regular expression matching circuit based on decomposed automaton. *IEICE TRANSACTIONS on Information and Systems* 95, 2 (2012), 364–373.
- [52] NetFPGA. 2014. NetFPGA-SUME board. <https://netfpga.org/site/#/systems/inetfpga-sume/details/>.
- [53] NetFPGA-SUME. 2020. NetFPGA SUME Reference NIC. <https://github.com/NetFPGA/NetFPGA-SUME-public/wiki/NetFPGA-SUME-Reference-NIC>.
- [54] Tien-Thinh Nguyen, Christian Bonnet, and Jérôme Harri. 2016. SDN-based distributed mobility management for 5G networks. In *2016 IEEE Wireless Communications and Networking Conference*. IEEE, 1–7.
- [55] Nping. 2021. An Open source network packet generation. <https://nmap.org/nping/>.
- [56] Panos Papadimitratos, Arnaud De La Fortelle, Knut Evenssen, Roberto Brignolo, and Stefano Cosenza. 2009. Vehicular communication systems: Enabling technologies, applications, and future outlook on intelligent transportation. *IEEE communications magazine* 47, 11 (2009), 84–95.
- [57] Taejune Park and Seungwon Shin. 2021. Mobius: Packet re-processing hardware architecture for rich policy handling on a network processor. *Journal of Network and Systems Management* 29, 1 (2021), 1–26.
- [58] PcapPlusPlus. 2020. DpdkBridge. <https://github.com/seladb/PcapPlusPlus/tree/master/Examples/DpdkBridge>.
- [59] Pupuweb. 2019. Deep Packet Inspection (DPI) and 5G: Network Visibility and Real-time Application Awareness. <https://pupuweb.com/dpi-5g-network-visibility-real-time-application-awareness/>.
- [60] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 13–24.
- [61] RAPID7. 2017. The Pros & Cons of Intrusion Detection Systems. <https://blog.rapid7.com/2017/01/11/the-pros-cons-of-intrusion-detection-systems/>.
- [62] Reetinder Sidhu and Viktor K Prasanna. 2001. Fast regular expression matching using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*. IEEE, 227–238.
- [63] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 403–415.
- [64] Snort. 2021. Network Intrusion Detection System. <https://www.snort.org/>.
- [65] Ioannis Sourdis, João Bispo, Joao MP Cardoso, and Stamatis Vassiliadis. 2008. Regular expression matching in reconfigurable hardware. *Journal of Signal Processing Systems* 51, 1 (2008), 99–121.
- [66] Suricata. 2021. An open source-based intrusion detection system (IDS). <https://suricata-ids.org/>.
- [67] 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>.
- [68] Qiu Tang, Lei Jiang, Xin-xing Liu, and Qiong Dai. 2014. A real-time updatable FPGA-based architecture for fast regular expression matching. *Procedia Computer Science* 31 (2014), 852–859.
- [69] Gautam S Thakur, Mukul Sharma, and Ahmed Helmy. 2010. Shield: Social sensing and help in emergency using mobile devices. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*. IEEE, 1–5.
- [70] David Barrie Thomas, Lee Howes, and Wayne Luk. 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, 63–72.
- [71] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422.
- [72] Udaya Tupakula, Vijay Varadharajan, and Naveen Akku. 2011. Intrusion detection techniques for infrastructure as a service cloud. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*. IEEE, 744–751.
- [73] Elisabeth Uhlemann. 2015. Introducing connected vehicles [connected vehicles]. *IEEE Vehicular Technology Magazine* 10, 1 (2015), 23–31.
- [74] Jan Van Lunteren. 2006. High-performance pattern-matching for intrusion detection. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. Citeseer, 1–13.
- [75] Giorgos Vassiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P Markatos, and Sotiris Ioannidis. 2008. Gnotr: High performance network intrusion detection using graphics processors. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 116–134.
- [76] Giorgos Vassiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P Markatos, and Sotiris Ioannidis. 2009. Regular expression matching on graphics hardware for intrusion detection. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 265–283.
- [77] Hao Wang, Shi Pu, Gabe Knezeck, and Jyh-Charn Liu. 2013. Min-max: A counter-based algorithm for regular expression matching. *IEEE Transactions on Parallel and Distributed Systems* 24, 1 (2013), 92–103.
- [78] Xiang Wang, Yang Hong, Harry Chang, Kyoungsoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: a fast multi-pattern regex matcher for modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 631–648.
- [79] Chengcheng Xu, Shuhui Chen, Jinshu Su, Siu-Ming Yiu, and Lucas CK Hui. 2016. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. *IEEE Communications Surveys & Tutorials* 18, 4 (2016), 2991–3029.
- [80] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya. 2008. High-speed regular expression matching engine using multi-character NFA. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE, 131–136.
- [81] Yi-Hua Yang and Viktor Prasanna. 2012. High-performance and compact architecture for regular expression matching on FPGA. *IEEE Trans. Comput.* 61, 7 (2012), 1013–1025.
- [82] Yi-Hua E Yang, Weirong Jiang, and Viktor K Prasanna. 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, 30–39.
- [83] Yi-Hua E Yang and Viktor K Prasanna. 2011. Space-time tradeoff in regular expression matching with semi-deterministic finite automata. In *2011 Proceedings IEEE INFOCOM*. IEEE, 1853–1861.
- [84] Zeek (Bro). 2021. The Zeek (Bro) Network Security Monitor. <https://www.zeek.org/>.
- [85] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. 2014. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE micro* 34, 5 (2014), 32–41.

Appendix A. Converting a regex pattern to Reinhardt logic

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

#	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	$(ab cd)+X\{1,3\}[0-9]^*$		

Converting a regex pattern '(ab|cd)+X{1,3}[0-9]*' to Reinhardt logic