Stateful InREC: Stateful In-network REal Number Computation with Recursive Functions

Matthews Jose, Kahina Lazri, Jérôme François, Olivier Festor

▶ **To cite this version:**

Matthews Jose, Kahina Lazri, Jérôme François, Olivier Festor. Stateful InREC: Stateful In-network REal Number Computation with Recursive Functions. IEEE Transactions on Network and Service Management, 2022, pp.1-1. 10.1109/TNSM.2022.3198008 . hal-03794876

HAL Id: hal-03794876
https://inria.hal.science/hal-03794876v1

Submitted on 7 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Stateful InREC: Stateful In-network REal Number Computation with Recursive Functions

Matthews Jose, Kahina Lazri, Jérôme François, *Member, IEEE*, Olivier Festor

***Abstract*—Current generation of Reconfigurable Match-Action Tables switches are highly programmable, able to support stateful operations and pipeline specifications using languages like P4. Nevertheless, these switches do not offer primitives to support real-valued operations on the data plane, thus requiring support from external servers or middleboxes to perform advanced operations. We introduce Stateful InREC, a system that extends the capabilities of programmable switches to support in-network real-valued operations using the IEEE half-precision floating point representation. Stateful InREC relies on decomposing real-valued functions into lookup tables taking into account the RMT model constraints to reach the right trade-off between accuracy and resource usage. It also supports state management for the computation of recursive function over time series. Stateful InREC prototype on Barefoot Tofino switches demonstrates the efficiency of Stateful InREC for in-network computation of different types of operations and its application for in-network logistic regression models used for classification problems. We also demonstrate the use of Stateful InREC to implement an ARIMA model on a Tofino switch for DDoS detection. Our evaluation of Stateful InREC shows that it is possible to implement complex in-network applications with high accuracy and low latency.**

***Index Terms*—Real Number, Floating Point, SDN, RMT, P4, Data plane Programming**

I. Introduction

Software-Defined Networks introduced network programmability. Recent advances in hardware switch architectures allow flexible systems where data plane operations are enriched with the support per-packet processing capabilities through the execution of custom match-action table pipelines [1]. Proposals like domino [2] and open state [3] provide new programming languages and abstractions that give access to the persistent memory of a switch. This has resulted in mounting interest for deploying basic stateful applications like load-balancers [4], [5], DDoS detectors [6]–[8] and DNS servers [9], on the dataplane.

Operating networks involves the execution of complex operations such as data analytics, intrusion detection and encryption/decryption, currently available in dedicated appliances. These applications require computational capabilities of modern computers. Recent proposals have discussed methods for leveraging network elements to execute parts of such applications directly within the data plane [10]–[12].

Despite these efforts, the lack of native primitives to support computations, equivalent to those provided by end servers, have resulted in modern day networks still being plagued with expensive middleboxes [12]–[14].

A programmable switch that supports real-valued operations would subsume many of the functionalities performed by these devices. Real numbers on computers can be encoded using predominantly fixed-point or floating-point representations. Computation using fixed point numbers requires less bit manipulation and can be implemented with simple arithmetic instructions such as those provided by P4 [15]. As a result, most work promoting in-network real-value computation relies on it [16]–[18]. Complex computational tasks including machine learning algorithms rely on floating-point numbers. Even network applications use several metrics such as average packet or byte counts that use large numbers. Besides, a distributed application that involves switches doing pre-computation, and end-devices, applying complex algorithms, will benefit from a common representation to avoid re-encoding numbers.

Our objective is to empower the capabilities of RMT-based programmable switches with floating-point number operations.

However, the limited resources on programmable switches [19] make it difficult to use traditional approaches to implement real-valued operations. We promote the use of LUTs (Lookup Tables) to support fast match between inputs and outputs of a real-valued function. A careful design and optimization of the LUTs is necessary to have a good trade-off between operation error and memory usage as a LUT cannot represent all possible inputs and is, by design, an approximation.

This paper extends our initial work, InREC, presented in [20]. First, we provide more insights about the challenges of performing rounding in floating point numbers within a pipeline to emphasize our choice of not supporting it in section IV-B and also we extended our solution to include stateful recursive real-value functions compared to InREC [20] that does not maintain any states among subsequent packets in-network.

We have also made changes to the evaluation section, including a comparison to NetFC [21] between our different approaches to addition. However, the most notable extension concerns the evaluation with a concrete use case application in section VII-B. Its aims so to boost the performance of anti-phishing platform based on an ARIMA (Autoregressive Integrated Moving Average) [22] model to detect DNS amplification attacks. It necessitates to revisit our initial proposal by the addition of a support for recursive functions. We provide various evaluations to demonstrate why its difficult to implement such recursive functions on programmable hardware switches. Finally, the intrinsic limitations of current

M. Jose (email: matthews.jose@loria.fr) is with Orange Labs and Université de Lorraine – INRIA, France.

K. Lazri (email: kahina.lazri@orange.com) is with Orange Labs, France

J. François (email: jerome.francois@loria.fr), and O. Festor (email: olivier.festort@loria.fr) are with Université de Lorraine – INRIA, Nancy, France.

programmable hardware architecture in regards to our work are discussed in section IX and a more detailed state of the art is presented in section VIII.

The main contributions of this paper are:

- a lookup table assisted implementation for a set of elementary operations,
- a procedure to automatically decompose a given real-valued function as a set of elementary operations including a set of optimizations to keep the LUTs minimal, in respect to the limited resources available on switches,
- an evaluation on a Barefoot Tofino switch to assess the accuracy and the overhead of InREC, the application of InREC for logistic regression, widely used in common classification problems.
- a study on deploying recursive real-value functions with a case study using ARIMA to identify in-network DNS amplification attacks.

The rest of the paper is structured as follows. Section II introduces the limitations of the RMT models. Starting from these limitations, the objective and the overview of Stateful InREC are given in Section III highlighting its two main blocks: elementary operations in Section IV and their automated composition in Section V. The results of the evaluation are reported in Section VI. Applications to concrete use-cases is described in section VII. Related work is addressed in Section VIII. We discuss some limitations and challenges when designing Stateful InREC in section IX. Section X concludes the paper.

II. Real-numbers support limitations in switches

A. Re-configurable match-action tables or RMT

RMT [19] is an architecture for programmable switches. It consists of 3 main blocks: (i) a parser that extracts fields from a packet, which, combined with metadata, forms a Packet Header Vector (PHV) as shown in figure 1, (ii) a match unit that performs match operations (exact, ternary, longest prefix match, etc.) on a range of bits from the PHV and (iii) action units that can modify the PHV using a set of operations. The pipeline starts with the parser followed by a sequential series of tightly coupled match-action units. These units are present in resource clusters called stages. Each stage is allocated a fixed amount of memory (TCAM and SRAM) and CPU resources. Each stage in the pipeline also consists of an optional register (provided by switch vendor) that provides the means of stateful storage on the switch.

P4 [15] is a platform-agnostic language inspired from the RMT model to allow a programmer to define a packet processing logic including its parsing to extract header fields, match-action tables to have conditional actions based on field match and deparsing (packet reconstruction). However, P4 does not support native floating-point data type.

B. Background on floating-point real numbers

Floating-point [23] numbers are composed of three main parts: the sign, the exponent and the mantissa. Since we use half-precision numbers in this paper, their respective widths

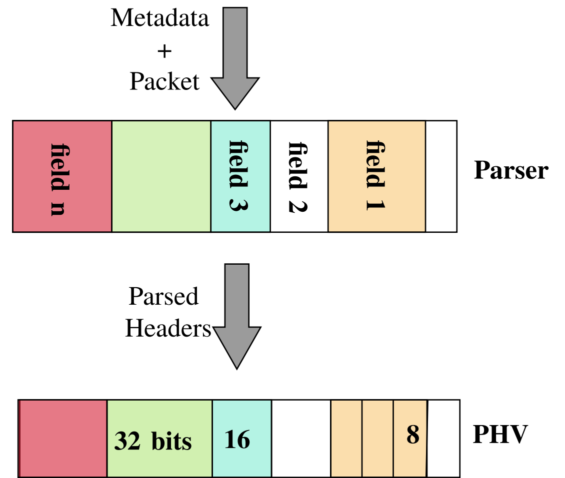


Fig. 1: Incoming packets are parsed and stored in discrete buckets of 8,16,32 bits size on the PHV. Several smaller buckets can be combined together to accommodate larger packet headers and larger buckets can store multiple headers

are : 1, 5 and 10 bits. Except for particular numbers (zero, infinite and subnormal numbers), the value of a floating point number is computed as follows:

$$(-1)^s \; * \; 2^{e-15} \; * M$$

with s the sign bit, e the exponent and M the mantissa

The IEEE 754 standard [23] requires that floating point numbers be *normalized*. This implies that the most significant bit of the mantissa is always set to 1 and hence can be omitted from the mantissa. Thus 1 is always prepended to the mantissa whenever a number is used in computation. Numbers between $[-1, 1]$ do not have this requirement and have their most significant bit set to 0. They are called sub-normal numbers.

C. Limitations of RMT based switches

The rigid structure of RMT and the implementation in silicon creates a number of physical restrictions. Since our objective is to support real numbers, they will need to be stored in the PHV. However, the latter consists of a large number of words or containers. These are usually 8, 16 or 32 bits wide. A single word can contain multiple small headers or a single large header that can span over multiple words. Hence, the different parts of a floating-point number (mantissa, exponent) are not necessarily aligned with the original word-length and can span over multiple words. They therefore cannot be easily extracted individually to perform operations. In addition, switch restrictions limit to one per stage, the number of single arithmetic operations (such bit-field addition) that can be performed on a word.

In contrast, operations on floating-point numbers require several steps including extraction of the different parts and normalization. Adding two real numbers requires re-arranging the smallest number to have the two equal exponents, applying the binary addition over the two mantissas and finally normalizing the result by making the mantissa value to be between 1 and 2.

Implementing floating point number arithmetic as in general-purpose computers (defined in IEEE 754 [23]) on switches also requires many stages. Because memory resources cannot be shared between different stages of a pipeline, unused memory from a stage is wasted [24]. Therefore, supporting floating point arithmetic would be very costly, supposing enough stages are available.

III. Objective and approach overview

Our objective is to take a real-valued functions and create a switch specific processing pipeline using the P4 language and RMT-based hardware. The resulting pipeline should be minimal in terms of resource usage and latency. Due to challenges highlighted in the previous section, traditional hardware-based computational methods for floating point arithmetic are impossible.

Stateful InREC promotes the use of Lookup Tables (LUTs) as a workaround. A LUT [25] is a pre-computed table that maps a set of input values of a function to a set of output values. Using this method, a computational operation can be translated into a match operation on the switch, hence circumventing most of the restrictions mentioned above. They also fit well with the common capabilities of a switch. Switches are designed to be efficient in two main tasks: classifying a network packet by means of match tables and then transmitting it. For instance, a switch with a throughput of 2.0 Tbps [1], [26] can classify more than 2 billion 1000 bytes size packets. On the other hand, a LUT performs a computational task by means of classification and thus can leverage the high speed capabilities provided by switches.

For example, to compute $log(x/y)$, we can define a LUT for log and one for $/$ then use them sequentially. However, it is challenging to find the right trade-off between resource usage (both match-action tables and LUTs will be placed in SRAM) and accuracy depending of the number of entries in the LUTs. Indeed, a LUT represents a finite set of input values and approximates the other ones by rounding leading to loss of accuracy. Providing a manually-tailored specific pipeline to a particular function is the best way to optimize this trade-off. For instance, a single LUT can represent the full $log(x/y)$ operation but this also limits its applicability to other computational functions if not prohibiting them.

Stateful InREC consists of two main building blocks. The first one (section IV) defines a P4 processing pipeline for a set of elementary operations such as $log(x)$ or $x+y$. This process is performed manually for each elementary operations, with several pre-computed variants for different settings. They are based on the trade-off between resource utilization, accuracy and the properties of the inputs. For example, the division has 2 variants: 1) when both numerator and denominator are variables and 2) when one of the inputs is a constant. The choice of the variant is decided in the last step of the pipeline building process depending on the setting in the final graph. The second one relies on this set of predefined elementary operations to automatically derive the processing pipeline of any compound function through multiple steps (detailed in section V):

1) combine elementary operations in a graph-representation of the function to compute;
2) identify the domain of variables and range of operations represented in the graph in order to better adjust the LUT entries, *i.e.* avoiding populating a LUT with useless out-of-domain input values;
3) aggregate several elementary operations in a single node in the graph if the resulting aggregated LUT to save some memory space;
4) transform the final graph into P4 instructions to implement the different operations represented as nodes based on the set of elementary operations (step 1) and optimizations done through steps 3 and 4;
5) do a final optimization by rearranging the composition of elementary operations to reduce output-input dependency between them because dependent operations cannot be executed in the same stage (output of a function cannot be reused as an input of another one in the same stage as explained in section II-C).

This process is executed on a controller and the resulting fully specified pipeline is installed on the switch.

IV. Elementary operations

A. LUT-assisted elementary operations

Elementary operations are the basic building blocks of any mathematical function (addition, multiplication,...). Elementary operations are classified as either *native* or *lookup table based*. Actually, native operations are limited by the hardware to the bit-wise operations (shift, concatenation,...) and the bit-field arithmetic operators. Furthermore, limitations described in section II-C still apply in particular on the number of operations that can be applied per stage on each word.

As an example, assuming $f(x,y) = x + y$, $x > 0$ and $y > 0$. To compute $f(x,y)$, x and y must have the same exponent values by shifting the bits of the smallest number. Once done, bit-field addition is applied on the mantissas followed by the normalization of the result. The number of shift operations to perform depends on the difference between the exponents e_x and e_y of x and y respectively. If $y < x$, the mantissa of M_y is shifted by $s = e_x - e_y$ bits, *i.e* $M_y >> s$. However, this is not allowed by the hardware we use (Tofino) because s cannot be a variable in a bit-shift operation. Alternatively, multiple hard-coded conditional statements based on the value of s can represent each possible shift. This is not efficient because the restrictions on operations described in section II-C forces each individual conditional statement to be in a different stage as they modify the same variable M_y. The only viable option is to use a LUT, $l(d,m)$, to implement the shift operation in regards of a compound input key composed of the exponent differences, d, and the mantissa m, so having $l(s, M_y)$ equivalent to $M_y >> s$. Followed by the shift operation, the mantissas are added, using the native addition operator provided by P4. The native addition operator is designed to function optimally on fields of 8, 16 or 32 bits which is different from the mantissa length (10 bits). So, the mantissa of the larger number M_x should be extracted and casted beforehand but only full words can be

extracted (trying to get 10 bits leads to an alignment error). Therefore, the whole real number x is directly added with M_y to compute R. Unfortunately, x includes both exponent and sign bits which are removed from R using again a LUT to find the correct result.

As illustrated above, even a simple operation relying on the native addition operator must be LUT-assisted. Actually, all other elementary operations do not have a equivalent native operator. They must be pre-computed with a LUT but also rely on bit-wise operations for side manipulation (*e.g.* bit shift). Hence, the processing pipelines of elementary operations are designed beforehand and comprise a combination of lookup tables and native operations.

B. Rounding issue when performing elementary operations

Rounding would be needed when using native based elementary operations. Actually, a similar solution using LUT could fit but at a high cost. For example, the multiplication $X \times Y$ requires to multiply the mantissas $M(X) * M(Y)$. A naive approach using a LUT would lead to create a key representing the two mantissas. For example, with mantissa of 10 bits, the key will be 20-bits long. Thus, creating a table for approximation will require 2^{20} entries that are 20 bits wide. This cost is further exacerbated since rounding has to be performed after each elementary operation is computed. Hence Stateful InREC does not perform rounding, we show that Stateful InREC's implementation of elementary operations are within a 5% relative error margin in section VI-C.

C. Constraints on LUT

The size of a LUT is fixed and the function inputs (keys in the LUT) can have a large combination of values. Its accuracy increases with smaller intervals between values. So, if a bounded representation of the function to compute exists, it should be used instead of the original one in order to group the number of possible inputs (constrained by the memory size) in the restricted intervals.

For example, $f(x) = log(x)$ is partially bounded because x must be positive. However, an equivalent and bounded form is $f(x) = n + log(\frac{x}{2^n})$ where n is the integer that makes $1 < \frac{x}{2^n} < 2$. With this transformation, values for $[log(\frac{x}{2^n})]$ are stored in the lookup table. The process is further simplified when using floating-point numbers, since they are stored in normalized form, or are expressed in the form $2^e * \sum_{n=0}^{10} 2^{-n} * man[n] + 2^e$ where e is the exponent and $man[i]$ the ith bit in the mantissa. Hence the mantissa is always bounded between 1 and 2, removing the cost of finding an equivalent bounded form. This is true for the example mentioned above and several other functions like e^x, $\sqrt{x}$ and $\frac{x}{2^n}$.

Therefore, when manually designing processing pipelines for elementary operations, looking for a bounded form is a good approach. Dependency between operations is also important to take into consideration. If operations are badly ordered, a single one can force another depending on its output to be put in the next stag for example, due to RMT restrictions on variable modification, and so increases the latency to compute the result. A dependent operation has to wait a full 12 CPU cycles before starting [27]. Carefully constructing pipelines to minimize these dependencies can help reduce the number of stages. Also, reducing the range of LUT keys would allow to represent with a finer granularity the value of the smaller range. For example, all $sin(x)$ values can be easily computed from $[0, \frac{\pi}{2}]$ rather than its period $[0, 2\pi]$. Pruning a LUT from low-varying intervals, *e.g.* an asymptotic functions, can save a lot of memory.

Elementary Function	#stages	#LUTs	#native op.
$log_2(x)$	4	3	1
$log_2(x)$ assuming no change after x=N	1	1	0
2^x	2	1	1
$x * C$	1	1	0
x/C	1	1	0
$sin(x)$	1	1	0
$\sqrt{x}$	3	2	0
$x + y$	3	2	1

TABLE I: Resources used by elementary operations

D. Summary

The restrictions on the use of native operators in current generation RMT switches makes a table-less implementation of elementary operations impossible. Our approach relies on a fusion of lookup operations, careful bit manipulation and native operations. Similarly to other works focused on fixed-point numbers [16]–[18], the design of the processing pipeline to compute these elementary operations is heavily manual even if the general idea is to reduce the operations to a bounded form and find a good trade-off between memory (LUT size) and the number of native operations and stages. Also, the procedures for the elementary operations were designed to be scalable with respect to wider floating point schemes (eg. 32 bit float or 64 bit double) so that wider schemes only require additional pipeline stages and not changes to table structure.

Besides, several variations of match/action units exist for each elementary operation. The most generic form is when input variables are not bounded. However, if an input variable is bounded (the domain is subset of $\mathbb{R}$), a more efficient variant exists. For example, the $log(x)$ elementary operation, consists of a single lookup followed by an addition operation. If x is bounded to an interval, the alternative implementation uses a single lookup table for all values in the interval.

Table I lists the implemented elementary operations and the resources they use (in their most generic form with unbounded variables). Except for $x+y$, no more than 3 stages are required [1], therefore limiting the latency to perform the operations. Besides, many of them are derived from others and so the same logic can be reused when defining the pipelines. For example, x/y is equivalent to $2^{(exp_x - exp_y) + (log(man_x) - log(man_y))}$, so the building blocks of the log operation can be reused and adapted.

[1] $log(x)$ requires more than 3 stages if and only if the variables are not bounded. We can fix a larger negative number i_l as $log(x)$ is zero if $x < i_l$ and i_u as $log(x)$ is infinity if $x > i_u$

V. Automatic building of function processing pipeline

Although the design of processing pipelines for elementary operations is manual, the definition of the processing pipeline of an arbitrary compound function f is fully automated.

A. Step 1: A theoretical pipeline

The theoretical pipeline, if it exists, provides the composition of f in terms of elementary operations and captures the dependencies between these elementary operations. It is represented using a Directed Acyclic Graph (DAG) as illustrated in figure 2. The nodes in this graph represent an elementary operation or a function variable and the edges represent the dependencies between variables or operations that compose a function. If the DAG cannot be constructed (iterative and recursive functions), so the function cannot be calculated.

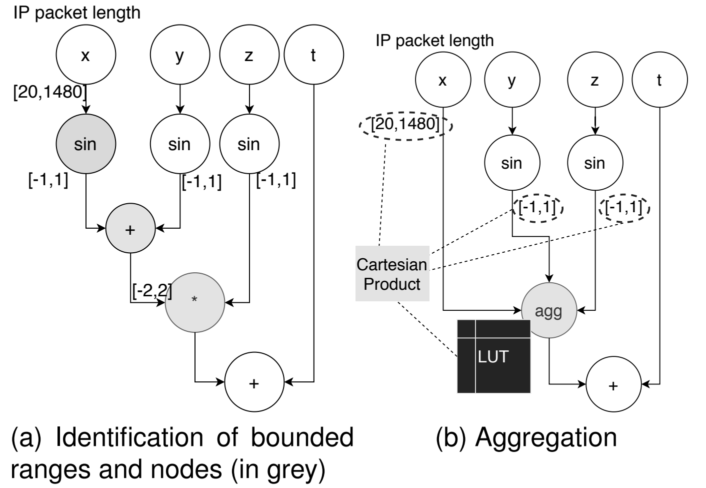
Variables can have contextual properties about their meaning. For example, a variable representing the network throughput cannot be higher than the bandwidth. Elementary operation nodes may also have mathematical properties as symmetry, a limited range (*e.g.* $sin(x) \in [-1, 1]$) or domain. Those properties are critical in determining if a node is bounded and when performing several optimizations in next steps.

Nodes representing state in the theoretical graph are placed before all computation nodes that are dependent on it. Stateful nodes are represented as registers on the switch, these are generally fixed in with (usually 32 bits wide) hence by default all values stored in registers have all values(this can change based on the representation used) that can be stored in 32 bits of data.

B. Step 2: Check for bounds on domain and range

The goal of this step is to infer the domain and range of each node in the theoretical pipeline when possible to identify bounded nodes. Although all types of nodes can have a bounded range for example, addition can be also bounded if its input are bounded, we qualify as bounded nodes only the functions and so exclude the variables. Bounded ranges (of variables and elementary operations) identified in step 1 are propagated recursively. More precisely, a node is said to be bounded only if it is a function whose its domain is a subset of the real set, $\mathbb{R}$. This is only the case if the ranges of all its parent nodes (including variables) are bounded. Bounded nodes result also in a bounded range and, consequently, their child nodes are bounded.

As a result, all adjacent bounded nodes form a directed acyclic subgraph, called a bounded chain, to be used for performing aggregation and other optimizations afterwards. Several bounded chains can exist. In figure 2(a), the sin operation is used multiple times and has a bounded range $[-1, 1]$ by definition. Also, x is a variable equivalent to the IP packet length bounded by the Ethernet MTU. So it is between 20 and 1480 bytes. As a result, $sin(x) + sin(y)$ is a bounded node. The multiplication is also bounded because the inputs are a sin function and the previous addition that is bounded. So the subgraph composed of sin, $+$ and $*$ in grey



(a) Identification of bounded ranges and nodes (in grey) (b) Aggregation

Fig. 2: Aggregation of bounded chains in $f(x, y, z, t) = (sin(x) + sin(y)) * sin(z) + t$ where x is the IP packet length.

is a bounded chain. This graph represents the bounded range of $sin(x) + sin(y)$ but it is not used. Only identified initial bounded ranges (of variables and elementary operations) are necessary for the next step (aggregation).

C. Step 3: High level aggregation

If a bounded chain is identified in the previous step, both input and output values of their aggregated equivalent function are limited to a certain domain and range respectively. So, an aggregated LUT, agg, representing the whole chain is preferred because the input keys can be carefully selected. This is similar to finding an equivalent bounded form of an elementary operation. In addition, aggregating multiple operations into a single LUT avoids all data manipulation which are necessary between each operations to manipulate real numbers (extraction from the PHV, normalization...).

Obviously, the number of entries in agg depends on the bounds of incoming edges. The incoming edges of the agg node are the ones from functions being aggregated. To avoid a loss in accuracy, the agg input keys are derived from the cartesian product of initial LUT keys. Assuming agg to have E in-going edge, each in-going edge e is bounded $[l_e, u_e]$ and originally associated with $\#[l_e, u_e]$ LUT entries, the input keys of the newly created LUT for agg is all possible combinations of all original keys so:

$$\#[l_{agg}, u_{agg}] = \Pi_{e \in E}[l_e, u_e]$$

where Π is the n-fold Cartesian product.

In Figure 2(b), the identified bounded chain of figure2(b) is replaced by a single node *agg*. This corresponds to the $agg(x, y, z) = (sin(x) + sin(y)) * sin(z)$ function. The input keys of the newly built LUT are derived from the bounded ranges of functions or variables used as input. In this example, the inputs are *x*, *sin(y)* and *sin(z)* which are bounded to $[20, 1480]$, $[-1, 1]$ and $[-1, 1]$ respectively. The number of entries in the new LUT is so $\#[20, 1480]$*$\#[-1, 1] * \#[-1, 1]$.

Taking into consideration all the possible combinations from the keys of the original LUTs avoids a loss in accuracy.

However, it can lead to a very large LUT which might need to be split into multiple stages. Doing such an aggregation is only viable if the number of stages to compute the aggregated function is not higher than the number of stages without aggregation. Otherwise, Stateful InREC does not aggregate the considered bounded chain. In that case, a smaller aggregation might occur. To do so, the sink (ending) node of the bounded chain is removed and the aggregation process is performed on the remaining part. This process is then applied recursively. In Figure 2(b), if the most global aggregation fails, the multiplication node is removed and the process tries to aggregate $sin(x) + sin(y)$ only.

```
1  G: DAG from  step 2
2  Bounded_List: List of bounded chains
3
4  foreach Boundedchain b in Bounded_List {
5    Get input nodes n_i for bounded chain
6    Get output nodes n_o for bounded chain
7    if (isAggregatable(n_i, n_o, b)) {
8       Set newNode as createNode(n_i, n_o,
      b)
9       Get number of stages #newNode using
      numberOfStages(newNode)
10      Get number of stages #b using
     numberOfStages(b)
11      }
12      if (#b > #newNode) {
13           substituteNewNode(G, newNode)
14           }
15      else {
16           Set reduced_b
     removeNodeFromBoundedChain(b)
17           addToBoundedList(reduced_b ,
     Bounded_List)
18           }
19  }
```

Listing 1: Identification and aggregation of bounding chains

The procedure 1 identifies the bounded chains and replaces them with a single aggregated node. As input, it uses the directed Acyclic graph generated from the previous step and also the list of bounded chains(**Bounded_List**). The procedure starts by iterating through the list of bounded chains provided as input and checks if the chain can be aggregated (using *isAggregatable* function line 7). The *isAggregatable* function uses the bounded chain b, input and output nodes to the bounded chain b to check for inconsistencies (infinity values, division by 0). If so lines 8-10 creates a new node(representing the aggregated node) and fetches the approximate[2] number of pipeline stages used on the switch, same for the bounded chain. If aggregation results in fewer stages then the new node is substituted into the graph. If not, the node belonging to the bounded chain b that has: 1) an edge from an input variable and 2) smallest domain is removed from the bounded chain b to create a new chain reduced_b(using function *removeNodeFromBoundedChain* line 16) and is then added to the list Bounded_list.

[2]This is an approximation calculated based in the individual elementary operations in the bounded chain, the exact number of stages depends on the target specific compiler used to compile the P4 code.

D. Step 4: Substitution of elementary operations and simplification

Remaining elementary operation nodes in the graph (nodes not aggregated) are substituted for their actual instructions in the form of lookup tables and action unit primitives that have been pre-defined in regards of their inputs because several variants can be implemented depending on them (see section IV). However, further simplifications can be still applied.

Also several elements are added for management, like mirroring mechanism for updating registers in the flow.

First, chained operations involving log and division are checked to see if they cannot be arithmetically simplified. For instance, a/b equals $2^{log(a)-log(b)}$. If the graph contains a subgraph corresponding to $log(\frac{a}{b})$, this is equivalent to $log(2^{log(a)-log(b)}) = log(a) - log(b)$.

Second, some elementary operation nodes can have incoming edges with bounds when they have not been aggregated. These bounds are the limited range of input values. In that case, all other values in the LUT can be discarded to save memory space.

We provide an example of the code generated in listing 2. It shows part of the P4 code that is substituted for the final addition operations in figure 2. The result from the lookup operation (agg node) is added with t variable in the graph at line 19. For example, since both values from the input node are always positive, the P4 code and the table keys do not account for it.

```
1  #shift the mantissa of the smaller number
2  by difference between the exponents
3  action sub_shift_action_1(bit<16>
     shifted_value) {
4          meta.shifted_1 = shifted_value;
5  }
6  table sub_shift_1 {
7  key = {
8      hdr.t[13:10] : exact;
9      hdr.agg[13:10] : exact;
10     hdr.agg[9:0] : exact;
11     }
12 actions = {
13     sub_shift_action_1;
14     }
15     size = 139269;
16 }
17 #Add shifted mantissa and the mantissa
     from the larger variable, compute the
     new exponent. put together the final
     float number.
18 action add_action_1(bit<10> res, bit<6>
     exp) {
19         meta.result_1 = exp++res;
20 }
21
```

```
22 table add_1 {
23 key = {
24     meta.res_1[10:0] : exact;
25     hdr.t[14:11] : exact;
26     hdr.t[10:10] : exact;
27   meta.shifted_1[10:10] : exact;
28     }
29 actions = {
30     add_action_1;
31     }
32     size = 139269;
33 }
```

Listing 2: Snippet of P4 code generated for the function mentioned in figure 2(b), in particular the last addition operation between the result from agg node and variable t, $t + agg$

Once the tables are created and installed onto the switch, the final step is to fill them at run-time. Finally, when computing the entries of a lookup table for a node (representing a function), there are 4 factors to consider: the intervals from the domain of the function being used, the frequency of sampling in this intervals, restrictions on the range of the function and properties due to the nature of the intervals being used. The steps are as follows:

- perform partial differentiation with respect to each independent variable for the function to detect slow or fast varying subranges (output values) using a threshold.
- fix a sampling frequency for each interval and obtain a set of samples with respect to each independent variable. A higher sampling frequency is used for fast varying ranges. The sampling frequency can be tuned based on targeted application-specific accuracy.
- create the keys of the newly created LUT from the Cartesian product of the sets obtained before to populate the LUT entries with these keys and the associated computed function value. It is similar to the aggregation optimization but on a single function here.
- prune the set of lookup table entries by removing keys that map to values considered as out of domain of its child node (given in step 1, when constructing the theoretical pipeline, in contextual properties of nodes).
- substitute all numbers in the pruned set with its equivalent floating-point representation and output the entries for the lookup table.

This procedure is performed using a high precision floating-point representation and is rounded to the nearest half-precision float number in the last step. The resulting table entries are encoded using the table entries paradigm provided by P4 for static elements.

E. Step 5: parallel processing

Parallel execution of independent operations reduces the number of stages needed and ensures maximum resource utilization in each stage while reducing the latency (directly depending of the number of stages to go through). Elementary operations that are commutative and occur consecutively can be executed in any order. This can be exploited to re-adjust the graph in such a way that consecutive commutative operations are independent, and merged later in a child node. For example, for $f(w, x, y, z) = w + x + y + z$, three stages are needed if an iterative add of x, y and z to w is performed to strictly follow the order of $+$ operators. Actually, this corresponds to apply a single addition per stage. However, computing $w + x$ and $y + z$ in parallel in the first stage before the final addition leads to two stages only. These patterns (arrangements of elementary operations) are predefined and compared against the graph to identify possible optimizations.

VI. Evaluation

A. Implementation

Stateful InREC is implemented as a python program producing a P4 program compiled then onto a Tofino switch using the BareFoot SDE. In that case, other optimizations can be done but are hardware specific. The elementary operations can use more compact and minimal action units. Several of the primitives used by elementary operations had table units to assist certain key operations, like variable bit right shift. To address alignment issues (mantissas are smaller than a 16 bit word), we use identity hash functions provided by Tofino.

B. Setup and metrics

Our setup consists of a Stordis BF2556X-1T-A1F switch[3] connected to IBM BladeCenter HS22 7870 servers with an Intel Xeon X5660 2.80 GHz and 1Gbps Broadcom BCM5709S NIC. For the evaluation, the function to be computed and its inputs are sent in a packet to the switch. The result is then sent back to the controller which performs the same computation on a commodity computer and compares the results.

Hence, the first metric to assess the accuracy of Stateful InREC is the (relative) error between the computed value by the switch and by one computed by the controller. Secondly, to assess its efficiency, the overhead implied by the computation is derived from the resource usage in the switch and the additional latency required for packet transmission. The evaluations are done with elementary operations tunes so that their maximum relative error is within 5%. This number is arbitrary and must be set according to a particular application in practice.

The Stordis switch uses a Tofino processor consisting of an ingress and egress pipeline of 10 stages each. Some of routines we use for elementary operations are based on the architectural limitations of Tofino. For example, values can only be right bit-shifted a constant number of places. Thus, right bit-shift operation has to be implemented using lookup tables with several action unit functions (one for each possible shift value) increasing the cost of implementing in Tofino. It also supports several extern functions in P4 including support for registers, symmetric hashing for calculating table keys and egress mirroring mechanisms. In general, Tofino is more restrictive than the open source simulator BMv2 [28].

[3]https://shop.stordis.com/switches/bare-metalwhite-box-switches/stordis-48x-25gbe-sfp28--8x-100gbe-qsfp28-switch-bf2556x-1t-a1f

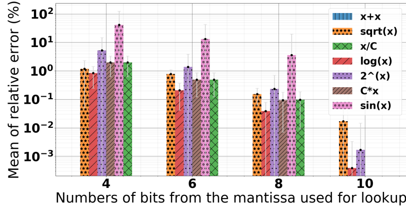


Fig. 3: Impact of the LUT key size on the relative error of elementary operations with C a constant.

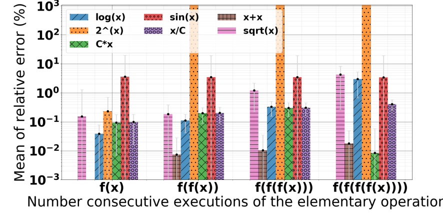


Fig. 4: The relative error when computing an elementary operation successively.

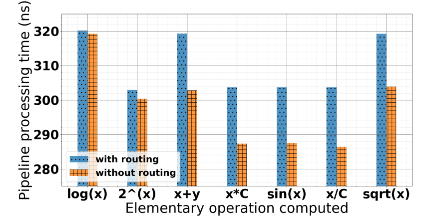


Fig. 5: Processing time by the switch for elementary operations.

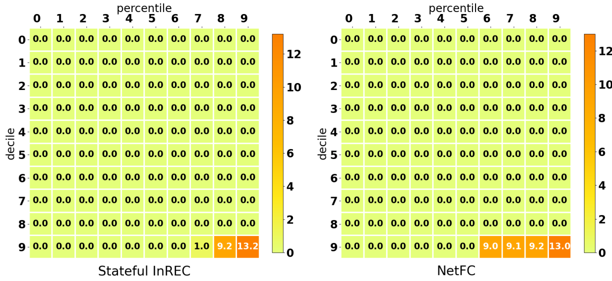


Fig. 6: Comparison of accuracy between NetFC and Stateful InREC when performing the addition operation

C. Error in a single elementary operation

In figure 3, we evaluate the error with respect to the memory used by the elementary operations. The exact amount of memory used depends on lengths of chosen keys and action values. For comparison, we took w (4, 6, 8 or 10 bits for match) bits representing the w most significant bits of mantissa for the lookup operations. The mantissa is thus truncated when $w < 10$ leading to a loss of accuracy but a lower LUT size. Thus a table contains 2^w entries.

For each elementary operation, the average relative error is computed while varying x between 0 and 10000 and w from 4 to 10 bits. The error decreases when w increases. This is especially true for operations with a high rate of changes like 2^x. To maintain an error below 5% w must be at least 8 for sin and 6 in all other cases. An ideal lookup table would contain all possible values of the mantissa and is represented by the case where $w = 10$. In that case, the error is due to other operations, like the addition operation, used after the lookup.

As described in Section IV, the addition is a lookup-assisted. It is done using the native bit-field addition operator but LUTs are used for performing bit-wise shift and computing the exponent values of the final result. Because these tables replace operations the switch does not support, w must be equal to 10 bits. For example, if shifting the mantissa by n bits (10 at most) is not possible, the addition cannot be processed further. For $w = 10$, the relative error is around 10^{-4} and is omitted in this figure. The value w directly impacts the size of LUT in SRAM that is discussed later in Section VI-F.

We also evaluated the difference in accuracy between NetFC [21][4] and Stateful InREC scheme over the same range mentioned above. We used the same metrics as used in the NetFC publication with the accuracy measured as follows:

$$accuracy = e^{-(\frac{|result - computedresult|}{|result|})}$$

The $result$ is computed using the python numpy2.1 on the setup mentioned in the previous section, this is compared with the $computedresult$.

Figure 6 plots the accuracy on a heatmap between NetFC and Stateful InREC . In this plot a cell (x,y) represents the logarithm of the total number of computed results whose accuracy is within the range $0.1*y+0.01*x$ and $0.1*y+0.01*(x+1)$. Stateful InREC has a similar accuracy when compared with NetFC with most of the computed values within $[0.98, 1]$. Since Stateful InREC uses a LUT based implementation for addition, with the primitive addition operator used to add the mantissas of both numbers and the remaining operations performed by a direct match on a LUT, the error is lower (order of 10^{-4}) than other elementary operations. However, NetFC does not allow a user to compose any compound function.

D. Error propagation

In Figure 4, we performed an elementary operation multiple times consecutively. The LUT size have been chosen such that the average relative error is less than 1% each time a single operation is performed and x varies between 10 and 2000.

As expected, the error increases with the number of times we perform the elementary operation except for the $sin(x)$ operation because its range is bounded and so the input values are restricted to $[-1, 1]$ after the first iteration. Obviously, the average relative error when performing an operation in a fixed interval is the same. This is different from the other operations where the result increases or decreases in magnitude, changing the average error observed at each iteration.

We can conclude that the error of an operation, when performing several iterations, depends on the properties of the underlying operations and has a maximum bound of 10% at 4 iterations. On average it is less than 1%, which is enough for most applications.

[4]implemented in python to mimic the proposed scheme scaling factor set to 1000

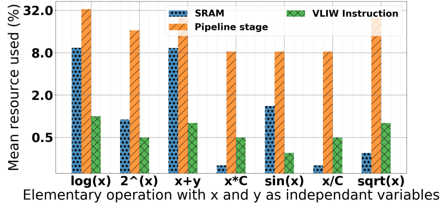


Fig. 7: Pipeline resources used by each elementary operation.

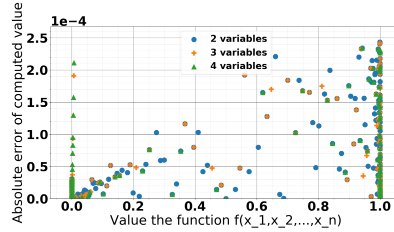


Fig. 8: Logistic regression error with 2, 3 or 4 independent variables.

Fig. 9: The relative error and true positive rate of the ARIMA model according to the timeslot size.

E. Latency overhead

Figure 5 shows the average processing time of the elementary operations. We compare 2 scenarios:

- without routing: once the operation is performed, the packet is simply sent back to its source port;
- with routing: once the operation is performed, the destination port is set using 3 lookup tables that use a combination of IP header fields as keys.

Assuming no routing, log requires the most processing time followed by the $sqrt$ and $+$. This is because the processing time is mostly impacted by the number of stages used in the pipeline. The log operation requires 4 stages in the pipeline while a single one is necessary for $*$, $/$ and sin.

The difference in processing time between both scenarios are smaller for log than for $*$,$/$ and sin because routing tables share the same pipeline stage as the LUTs for log. Moreover, routing tables use the TCAM memory available in the stages that is considerably faster and hence do not add much more overhead even in other cases. Especially, in the case of $*$, $/$ and sin, the operation only uses a single stage compared to the routing tables that take 3 stages, leading to a three-stages pipeline. An exception is the $+$ operation. It requires a significant amount of action units competing with those needed to perform actions of the routing tables pushing the tables to other stages with more available memory.

Nonetheless, the additional latency involved by Stateful InREC for a single operation is below 15ns (nanoseconds). Even if multiple operations are applied, this overhead is well below the end-to-end latency of packets whose order of magnitude is in milliseconds. Also, the worst case in Figure 5 is 320ns for $log(x)$ which is equivalent to 3,125,000 packets per second.

F. Ressource overhead

Figure 7 shows the average usage of SRAM[5], pipeline stages and Very Long Instruction Word (VLIW). The average value is computed across the total available quantity of a specified resource. The sizes of LUTs are adjusted in size such that the induced relative error is lower than 1%.

The VLIW instructions is the memory used to define action units and the instructions to be executed by them. $log(x)$ uses the most resources (33% of the total number of stages and 8% of the total SRAM available) in its most generic form. The bulk of the SRAM usage is similar of $x + y$, around 8%.

[5]The total SRAM size of the switch cannot be divulged due to a non disclosure agreement

Elementary operations relying on native operators or bit manipulation in their implementation, like $x + y$ and $log(x)$ (for performing addition), require more VLIW instructions. Most of elementary operations are fully lookup table based and have simple action units that are mostly assignment operations. Nonetheless, the capabilities of Stateful InREC (number of real-valued operations) are constrained by the number of stages and the available SRAM because the number of VLIW instructions remain low ($< 1\%$) in all cases. Thanks to its design, Stateful InREC reduces simultaneously the requirements on these resources as multiple operations initially put on different stages can be combined in a single LUT (high-level aggregation in step 3) and so in a single stage. Furthermore, the number of input keys is also optimized in step 4 to reduce the LUT size and so the SRAM used. The number of stages to be used is the most critical resource in the switch pipeline since only a small finite number is available and unused resources at a specific stage cannot be used by any other stage by switch design. Operations must be thus distributed among the stages to use as less as possible stages and above all by avoiding wasting resources, *i.e.* optimized to use most of available memory. Finally, parallel processing in step 5 reduces the number of stages. As a result, reaching a good precision is possible with a low overhead on resource usage.

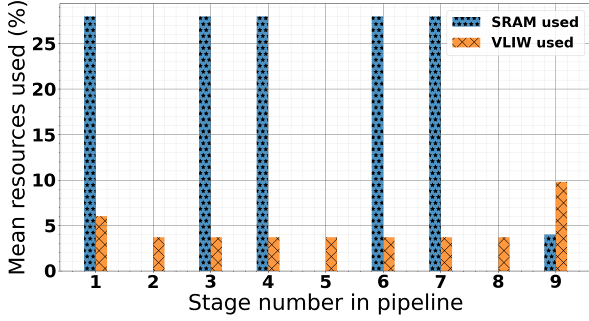
VII. Application to use cases

A. Logistic regression for device classification

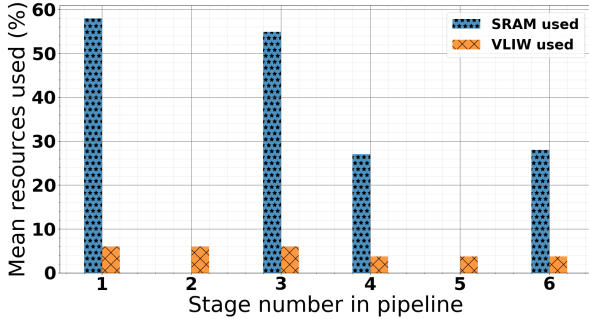
To show the viability of our approach, we implemented a logistic regression model that is mainly used for classification, for example to detect phishing [29] or malicious URLs [30]. Several of these applications are deployed on the edge of a network and require a low latency response time. A logistic regression model can be computed in its equivalent form as:

$$f(x_1, x_2, x_3, .., x_n) = \frac{1}{1 + 2^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n)}} \quad (1)$$

The training of the model, *i.e.* learning the β parameters, is pre-computed and only its application is run on this switch as this would be the case for a real deployment, *e.g.* to detect malicious traffic flows at the edge of networks. Equation 1 is derived from usual logistic regression but uses bases 2 rather



(a) Resource usage without Stateful InREC optimization



(b) Resource usage with Stateful InREC optimization

Fig. 10: Amount of SRAM and VLIW resources needed to implement a 4 variable logistical regression function with and without optimization. Stateful InREC's optimization reduces the overall number of stage used and furthermore enables parallelism of computational operation, reducing wastage in a given stage.

than base e and uses 4 variables with constants as $\beta_0 = 0.1$, $\beta_1 = 0.1$, $\beta_2 = 0.1$, $\beta_3 = 0.1$ and $\beta_4 = 0.1$.

As highlighted, the complexity of the computation depends on the number of variables but increasing their number from 2 to 4 does not impact the accuracy as noticed in figure 8 (error lower than 2.5×10^{-4}). For testing purpose, we classified 50 random data points in four dimensions either with Stateful InREC or with Scikit-Learn[6] in python with a common computer and verified that the output is the same. So, the very low error induced by Stateful InREC does not impact the final classification. Also, we performed the same test using 16 bits width fixed-point number scheme (with 9 bits for the whole number part and 7 bits for the decimal part). The largest number this scheme can represent is 511.984375, hence numbers larger than this results in an overflow. This makes it insufficient for use with the model above, since any computation in $\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4$ resulting in a number greater than 511.984375 will increase the error significantly. A solution could be to use a wider 32 bit fixed-point number.

The latency increases with the number of variables: 304.1, 314.64 and 320.3 nanoseconds in average for 2, 3 and 4 variables respectively. The higher difference between 2 and 3 variables is due to one more stage whereas the 4th variable calculation can be done in parallel. This confirms that our design choices to optimize the pipeline, including aggregation and simplification as described in section V, are valuable. For comparison, the naive pipeline as introduced in step 1 in section V-A is also applied. Figure 10 shows that the optimized pipeline uses three less stages than the naive one considering four variables. Actually, the three last stages performed in the naive pipeline have been combined in a single LUT thanks to our aggregation scheme. Figure 11 shows the final pipeline generated by Stateful InREC where the multiplication and addition operations are performed in the first 5 stages $(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4)$ and last stage contains the aggregated table for the function $1 \setminus (1 + 2^{-(input)})$ where $input$ is the result of computation from the first 5 stages. In practice, the number of stages in the switch we used limits the maximum number of variables for logistic regression to four for the naive pipeline whereas our technique allows to use up to six variables.

[6]https://scikit-learn.org/

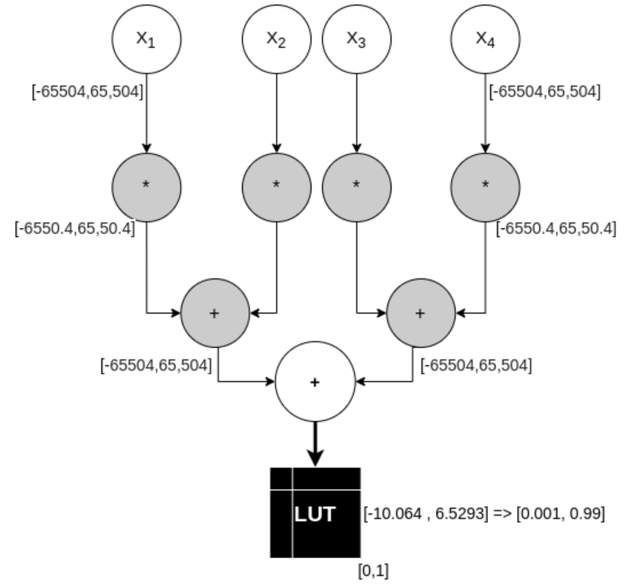


Fig. 11: The aggregated pipeline DAG for the logistical regression function implemented (equation 1). The first 2 addition operations are placed on the same stage (each level of the tree is placed on the same stage) in parallel and the functions $\frac{1}{1-2^{input}}$ is aggregated into a single lookup table.

Assuming the computation done in an external server connected to the switch, the overall latency is 4.733 milliseconds while it is reduced to 2.367 milliseconds in our case. So, the increased computation time due to the switch architecture in comparison to a modern computer is largely compensated by the link latency even with a single hop. Therefore, Stateful InREC can be used in practice to support advanced monitoring functions like logistic regression with a high accuracy and low latency.

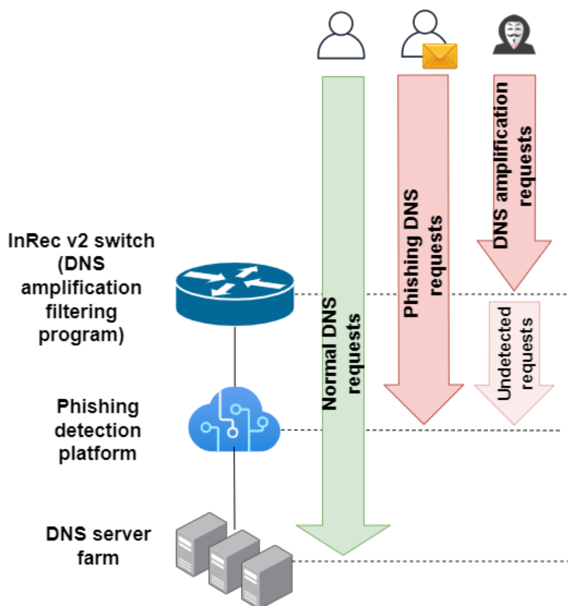


Fig. 12: In-network DDoS request filtering as support for an anti-phishing platform.

B. Detection of anomalies in DNS requests

1) Context: In a French ISP, a dedicated platform has been set to detect phishing emails by analysing the DNS traffic generated when a user accesses malicious links received by emails that result in sending DNS requests. This process heavily relies on filtering rules and machine-learning based models.

This system is heavily stressed due to the volume of malicious traffic (mainly DDoS attacks) along with genuine DNS queries being received. We are thus interested in rapid and basic filtering of such a traffic at the switch level. Therefore, our aim is not to build a perfect detection system but rather an accelerator which would be able to discard most of malicious traffic before it reaches the anti-phishing platform.

Figure 12 illustrates our approach. Stateful InREC is deployed as a front-end network equipment. Therefore, normal user requests are forwarded through the programmable switch to the phishing detection platform and subsequently to the DNS servers. Phishing related DNS requests are filtered by the phishing detection platform. Our solution filters earlier the DNS amplification attack related requests. As shown in the figure, the approximation of calculation of Stateful InREC does not guarantee a 100% filtering of these requests but the objective is to minimize the number of undetected requests.

2) In-network ARIMA model for DDoS detection: DNS Amplification attacks [31] are Distributed Denial-of-Service attacks where the attacker exploits public DNS resolvers and floods them with requests using spoofed IP address to overwhelm the real IP address (the victim) with the replies. The victim and its surrounding infrastructure is thus flooded by the latter with a risk of disruption. Indeed, the payload of DNS queries is relatively small compared to the response payload, leading so to qualify the used open DNS resolvers as amplifiers.

Our objective is to filter out DNS amplification requests, thereby allowing the system to focus primarily on phishing related traffic. Time-series models such as AR, ARIMA and FARIMA are frequently used for DDoS detection. [32], [33] use algorithms based on a ARIMA model to detect potential DDoS traffic. Our solution, consists of deploying a simplified ARIMA model on a programmable switch in front of the network to identify and drop DDoS traffic using a pipeline generated by Stateful InREC.

As highlighted in previous section, Stateful InREC acts as an accelerator by discarding as most of possible malicious traffic. Hence, our aim was not to choose the most effective model or algorithm for detecting malicious traffic but to select one that could fit on a switch in terms of resources and still with a good capability to discard malicious traffic.

In our case we use a simple ARIMA model (ARIMA(0,1,1) or simple exponential smoothing) to identify flows (a set of DNS requests sent by the same IP address) that are part of an amplification attack to the server. Our intention is to allow flows that are considered safe by the model to be forwarded and to drop all others. The main indicator we used is the number of bytes in a flow. The model is described as follows:

$$S_t = \alpha * S_{t-1} + (1 - \alpha) * X_t$$

X_t is the cumulative size of the packets received in the current time window
α is a smoothing factor determined by ***method of least squares***

The values of S_{t-1} are as per the model supposed to indicate the cumulative packet size in the time slot t. So, for each packet received, X_t is updated. If it exceeds a threshold, $X_t > f_t \times S_{t-1}$ compared to the calculated value, the packet is dropped.

3) Packet mirroring for state update: The ARIMA model is computed with a recursive function. On a RMT like switch, stateful elements called registers must be exploited. However, we remind that the computation of a function is spread over multi-stages. In contrast, as mentioned in section II-C, access to each register is strictly localized to a pipeline stage. Since data flow within the pipeline is uni-directional and stages cannot be traversed twice it is impossible to perform the following sequence of operations: (1) fetch a value from a register, (2) perform computations using this value across multiple stages and (3) update the register with the result.

To address this problem, Stateful InREC uses packet mirroring. For implementing the ARIMA model, two registers are used R_1 and R_2 for each monitored flow. R_1 contains the value of the previous iteration, S_{t-1} while R_2 contains the cumulative byte count of the current time window (stored in half-precision floating point format), X_t which is updated for each received packet. Indeed, the ARIMA model itself is updated at a fixed interval resolution using the total byte count at the end of the time window. So, two different processes are implemented: when DNS requests are received and when the time window ends.

Upon the reception of a DNS request, the process works as follows:

- S_{t-1} and X_t values is read from R_1 and R_2 respectively.
- The size in bytes of the current packet is converted to half-precision floating using a single lookup table and is added to X_t.
- If $S_{t-1} > T$, the packet is dropped. The comparison is made by a single lookup table that uses S_{t-1} from register R_1 as the match key and performs either a drop rule or assigns a port to forward the resultant traffic too. Independently of that step, the process continues because the ARIMA model must be updated.
- X_t is copied to a temporary header and then mirrored using the mirroring mechanism at the egress part of the pipeline. Mirroring is a function offered by the Tofino switch we used. It creates an exact copy of a packet but regarded as a separate packet with its own metadata.
- The mirrored packet is so recirculated from the start of the pipeline, where the corresponding register(s) (in our example R_2) is updated from the value(s) in the temporary header fields and the original packet is forwarded to its destination uninterrupted.

For time-slot or time window of the time-series, the ARIMA model is updated. The start of a new time window in the time-series function is triggered by a beacon packet sent from the controller to the switch called a window packet, which resets all relevant registers in the switch for the new window. Indeed there is no timer mechanism we can rely on in the switch.

When Stateful InREC receives the beacon, the process works as follows:

- S_{t-1} and X_t values is read from R_1 and R_2 respectively.
- S_t is computed according from S_{t-1} and X_t. This step performs 2 multiplications operations $temp_1 = \alpha * S_{t-1}$ and $temp_1 = (1 - \alpha) * X_t$ followed by the addition $temp_1 + temp_2$ operation. The multiplications are performed in parallel in a single stage, followed by the addition operation that uses 4 stages in the pipeline.
- Using packet mirroring as before, the new value S_t is now stored in R_1 and will be used for future computation in the upcoming time-window.

Both the register and the update method (replication + re-circulation) are placed at both ends to encapsulate the rest of the pipeline. This implementation ensures that the original packet processing is never interrupted. However, the update method requires duplication and re-circulation that does not guarantee state consistency between more recent computed value and the new value read for future computations. Indeed a new packet can be interleaved between the previous packet and its duplicate. It would thus entail an approximation in the computation and a higher error evaluated in our experiments. We refer to this problem as time-slotting in the rest of the paper. However, because of the uni-directionnality of the stage pipelining, any stateful function, even simple as the ARMIA model we propose, must rely on this method.

4) Scenario description: To evaluate our solution, we used ddosflowgen[7], an open source tool, to generate synthetic traffic representative of a DNS amplification attack. We captured 4 hours worth of DNS requires in our lab network to serve as the noise dataset for ddosflwogen to generate traffic for DNS query flooding from different sources.

As highlighted in table II, we so consider 15 malicious hosts and 40 benign hosts. As explained, our aim is to reduce the load of the anti-phishing system rather. Hence, the traffic is overloaded with malicious requests in order to force Stateful InREC to perform numerous computations and so possibly a large number of errors due to the time-slotting issue. Also, we have tested the effectiveness of a function pipeline build using Stateful InREC and deployed in-network when compared to running the same function on an external server.

Number of attack source IP addresses	15
Number of benign source IP addresses	40
Number of malicious DNS requests	340505
Number of benign DNS requests	1290
Average inter-arrival time of requests	0.2615 milliseconds

TABLE II: The number of malicious and benign traffic used in our evaluation

5) Error due to time-slotting: As highlighted, the start of a new time-slot in the time-series function is triggered by a beacon packet sent from the controller to the switch

Figure 9 shows the relative error and the corresponding true positive rate (details provided in the next section) between the in-network computed value and the value computed by a commodity hardware (the controller). The stream of traffic with exactly the same payload is sent, the network latency between the controller and the switch is kept at 2ms and the function is reset for each window size tested. The relative error is noticeably high at very low window sizes and decreases as the window size is increased. By reducing the window size, the chances of the beacon packet not arriving on time increases drastically, especially when approaching the value of the latency between the control and the switch. This implies that packets arriving between the end of a window and the beacon packet will be counted as belonging to the previous window. This results in an error in the computed value. However, with 100ms this error is extremely low and will be used in the next experiment, showing it provides a good accuracy to filter out malicious traffic. It is worth reminding that DNS requests are processed individually at line-rate to check if they fit the ARIMA model. So the 100ms time window does not delay their processing and their eventual filtering. Furthermore, for the 100ms window we tried using a controller to compute the value S_t. But the delay between sending the register values to the controller and receiving the update increases the systems reaction time as a whole. Also we think, one of the major advantages of in-network computing is the fast reaction times especially in time sensitive networks.

6) Detection accuracy: This test aims at evaluating the accuracy of the in-network ARIMA model to predict attacks when compared with the same model deployment on an external server. We have tuned the threshold value to ensure that no false positive are generated by the ARIMA model (with the highest possible true positive rate) when implemented with python. It results in the value 275.5 for our test traffic. Thus,

[7]https://github.com/GaloisInc/ddosflowgen

it compares the performance of an in-network deployment using a pipeline built by Stateful InREC to one running on an external server using python, excluding the effectiveness of the underlying model itself. It is worth mentioning that even an external server misclassified 11.87% of the flows. However, since our objective is to compare our approach against a deployment on a server, the results for the in-network deployment are calculated by its deviation from the results of the external server (taken so as our ground truth), which is thus partially erroneous.

The resulting false-positive rate is 0.05% for Stateful InREC with only a single benign flow classified as malicious. The true positive rate is 66%, with two thirds of malicious of DNS requests filtered upstream by the anti-phishing platform. As explained, our goal is to preprocess and filter a maximum of DNS requests while keeping intact the benign traffic. The unfiltered malicious requests will be analysed by the anti-phishing platform as it was the case without our solution. This highlights that Stateful InREC can be used to complement other advanced systems or middle boxes in the network.

Indeed, only one third of the malicious traffic remains and so must be then filtered out by the external server. The inability for Stateful InREC to reach a higher level of true positive rate is due to the absence of a rounding scheme as mentioned in previous sections, leading to error in the computed function. The accuracy decreases even more for larger time windows. Despite a lower error due to time-slotting, the error in the computation itself increases because (1) with more packets, more computations must be performed leading to accumulate errors and (2) the computed real number value is is near the end of the floating-point range because of the cumulative packet sizes.

To confirm this second assumption, we performed the same test with packet size set to 1 (assuming header sizes are also excluded). So, the computed value remains in a lower range. In figure 9, the true positive rate increases when the window size increases. Globally, the computed values remains relatively small in the domain of floating point computing. Hence, the error incurred due to the lack of rounding is also smaller. In that case, the main contribution to the error factor is due to the beacon packet updates. As a conclusion, finding the right trade-off to reach a satisfactory accuracy may depend on several factors whose impact may vary depending on the inputs of the function (*e.g.* packet size). Application knowledge is so necessary to tune well the configuration of time-slotting for example.

7) Reaction time: A major advantage of an in-network solution is the increase in reactivity we can gain in comparison with the use of a commodity hardware or server, which the advantage is its higher accuracy.

We thus assess the reaction time to the DNS amplification attack between an in-network Stateful InREC generated pipeline and a controller-based reaction. The time here is the number of packets received before the stream is dropped (send a batch of N packets and measure the number of packets received). With the controller, the reaction time equals the time needed for a digest message to be generated and sent to the controller and the controller to program a drop rule on the switch. The test assumes that both deployments are able to detect an attack and that the traffic sent is carefully filtered in an equivalent manner. The reaction time with the in-network switch is 260% faster on average. The bulk of the delay with a controller-based system is mostly due to the communication overhead as expected.

8) Processing time: We compare the processing time of benign DNS requests with and without the Stateful InREC's pipeline deployed. The processing time is evaluated considering all modules from when the packet arrives at the switch until it is processed by the DNS servers as illustrated in figure 12. This is why only benign DNS requests are considered, the malicious ones are filtered before reaching the DNS servers. In parallel, we thus count the number of malicious DNS requests properly identified and dropped by the in-network ARIMA model-based system.

The total time taken to process 341795 DNS queries only the phishing detection system is approximately 17.1 minutes. Deploying the ARIMA model on Stateful InREC results in approximately 259764 malicious packets being dropped at the switch, reducing the overall process time to 4 minutes, a reduction of 75.8%. The addition of the switch itself contributes a negligible amount of 10ms of extra processing time. Thus, Stateful InREC reduces the load of the phishing platform which leverages more complex processing and so its global processing time. Leveraging the speed of a switch's processing pipeline for detecting malicious traffic filters out several packets that otherwise have to be processed by the phishing system at almost no cost (delay in a few nanoseconds at switch level). Therefore, Stateful InREC can handle and perform an analysis of any DNS request in comparison with the phishing technique which rely on sampling to be scalable.

VIII. Related Work

A. In-network computing

Multiple works propose to leverage in-network computing capabilities. Monitoring applications that use measurement constructs like counters and sketches are one of the most popular use cases [34], [35] but state machine representations have also been proposed [36], [37]. Other examples target in-network key-value store applications [11]. Sonata [38], is a telemetry system that uses programmable switches in tandem with stream-processors to implement user queries. Sonata can be used for implementing a query detecting DNS amplification attacks by counting the total number of packets in a given query window, but cannot perform floating point operations in-network without reliance on the onboard switch stream processor.

In the scope of this paper, various methods have been proposed to perform real valued operations. Naveen Kumar *et al* [12] implemented RCP and calculate fair-rate using in-network division.

In [17], the authors offload an entropy model for DDoS detection to the dataplane, using a combination of sketches and LUTs to perform real valued operations. Recently, [18] focused on entropy calculation.

While all these proposals provide methods to optimize specific functions, they do not consider the general case of any

real-valued function as Stateful InREC promotes. Furthermore, none of the work considers floating point real numbers. In [39], the authors explore the possibility of deploying machine learning classification algorithms in-network, hoping to one day have switches perform some of the task being done on servers. They highlight the lack of floating point operations in current hardware. Adding hardware accelerator for machine learning has been also proposed [40].

A few proposals implementing floating point elementary operation computation have been produced. In [41], they use a 43 bit fixed point number to represent a float but these solutions do not make it easy to put together pipelines to compute a complete function. FPISA [42] decomposes floating point addition into multiple steps and maps them to stages in the pipeline. This solution implement various arithmetic operations but constructs pipelines for a function as a whole. The methods proposed in these papers can be incorporated into Stateful InREC as predefined elementary operation routines.

NetFC [21] enables in-network floating point arithmetic elementary operations by decomposing it into a series of optimized logarithmic LUTs. However, NetFC is not able construct an optimized pipeline for a compound function but implements individual arithmetic operations. Thus, unlike Stateful InREC ,NetFC does not optimize with respect to the context of the function being deployed and its properties, for example variables being added could have extrinsic properties that restrict its values within a certain range.

B. *Pipeline Design and Resource Optimization*

One major challenge of InREC was to build efficient pipelines for real valued operations. Magellan [43] takes a high level program and builds a compact pipeline that is optimized for a given hardware. [44] looks at the advantages of a greedy approach compared to a linear programming approach for optimizing a pipeline. Stateful InREC uses a combination of techniques to create pipelines but, most of all, is specific to floating point numbers. Thus, specific optimizations like aggregation of operations or reducing the LUT size with bounded operations can be performed unlike the general case. While our work is focused on optimizing the processing on a single switch, the literature is vast about the decomposition and optimal placement of a program. SNAP [45] uses xFDD for decomposing and analyzing dependencies of state variables in a program and further optimally places them on the dataplane by solving a mixed integer linear program. Merlin [46] uses service chains and optimizes traffic using an MILP (Mixed-integer linear programming). Stateful InREC is complementary to these approaches as it brings the possibility to perform real valued operations in a switch, so the constraints of those operations (resource needed and achieved accuracy) could be integrated into a global optimization problem. Due to resource struggle, there are several papers that propose adding external memory [47] and, in some cases [48], with custom functions.

IX. Limitations and remaining challenges

Elementary operations implemented by Stateful InREC use between 1 and 4 stages to be computed. This is relatively a large in regards to the total available stages in a switch. Frequent operations like addition require four stages per execution. This can rapidly result in resource exhaustion if a function contains multiple additions. Elementary operations cannot be optimized to use less resources because the structure of floating point numbers does not align with the structures provided by the switch store and each operation using a floating point number requires normalization and de-normalization. Identifying a representational scheme for real-values that have an equivalent range as that of floating point numbers is the key for implementing resource conservative elementary operations.

Recursive and stateful functions are a common class of functions used on the network (e.g time-series functions). As shown in this paper, updating registers makes the problem of concurrency being exacerbated when dealing with high bandwidth traffic. Research in programmable switch architecture is headed in the general direction with newer architectures like dRMT [24] defining a common resource pool that is accessible across stages. More flexible architectures are of paramount importance when implementing these types of functions.

Stateful InREC is currently designed to operate on a single switch. In a network with several programmable switches, the power of potential computation is not exploited yet. However, enabling Stateful InREC to deploy a function across several switches creates several challenges in itself. Identifying the optimal set of resources in the network, ensuring atomic operations are a few of such challenges. This is a direction for future enhancement of Stateful InREC.

X. Conclusion

Stateful InREC is a new approach to support real-valued functions on RMT based switches including recursive function over time series, which require a mechanism to synchronize a state value across multiple stages of the RMT pipeline. Our approach cleverly combines native operations and LUTs to construct a minimal switch specific pipeline guaranteeing an acceptable loss of accuracy in the computing. This pipeline is expressed as P4 logic and sent to programmable hardware switches. The evaluation on a Tofino switch shows that reaching a relative error below 5% or even 1% is possible with a low amount of resources making Stateful InREC a viable approach to support complex functions and algorithms. The use-case implementing ARIMA model shows highlight the capacity of Stateful InREC to serve as an accelerator for complex processing which cannot be executed onto a switch.

In future work, we plan to release Stateful InREC as an open-source software and investigate how more complex functions calculation which cannot fit in a single switch pipeline can be decomposed over multiple switches. We will also investigate how to minimize the error based on the available resources.

References

[1] Arista, “Arista 7170 multi-function programmable networking.” [Online]. Available: https://www.arista.com/assets/data/pdf/Whitepapers/7170_White_Paper.pdf

[2] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[3] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "Openstate: Programming platform-independent stateful openflow applications inside the switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, april 2014.

[4] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *ACM Symposium on SDN Research (SOSR)*, 2016.

[5] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[6] A. C. Lapolli, J. Adilson Marques, and L. P. Gaspary, "Offloading real-time ddos attack detection to programmable data planes," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019.

[7] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *Proceedings of NDSS*, 2020.

[8] M. Dimolianis, A. Pavlidis, and V. Maglaris, "A multi-feature ddos detection schema on p4 network hardware," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. IEEE, 2020, pp. 1–6.

[9] J. Woodruff, M. Ramanujam, and N. Zilberman, "P4dns: In-network dns," *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019.

[10] A. Sapio, M. Canini, C.-y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," in *KAUST*. KAUST, 2019.

[11] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "Incbricks: Toward in-network computation with an in-network cache," *ACM SIGOPS Operating Systems Review*, vol. 51, april 2017.

[12] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation," in *14th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, Mar. 2017.

[13] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An untold story of middleboxes in cellular networks," *SIGCOMM Comput. Commun. Rev.*, vol. 41, Aug. 2011.

[14] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: Enabling innovation in middlebox deployment," *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011.

[15] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014.

[16] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the network be the ai accelerator?" in *Proceedings of the 2018 Morning Workshop on In-Network Computing*. Association for Computing Machinery, 2018.

[17] Â. C. Lapolli, J. A. Marques, and L. P. Gaspary, "Offloading real-time ddos attack detection to programmable data planes," in *IEEEInternational Symposium on Integrated Network Management*. IEEE, 2019.

[18] D. Ding, M. Savi, and D. Siracusa, "Estimating logarithmic and exponential functions to track network traffic entropy in p4," in *IEEE/IFIP Network Operations and Management Symposium*, 2020.

[19] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *SIGCOMM Conference*. ACM, 2013.

[20] M. Jose, K. Lazri, J. François, and O. Festor, "Inrec: In-network real number computation," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021, pp. 358–366.

[21] P. Cui, H. Pan, Z. Li, J. Wu, S. Zhang, X. Yang, H. Guan, and G. Xie, "Netfc: enabling accurate floating-point arithmetic on programmable switches." IEEE, 2021.

[22] B. E. Hansen, "Time series analysis james d. hamilton princeton university press, 1994," *Econometric Theory*, vol. 11, no. 3, p. 625–630, 1995.

[23] IEEE, "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, 2008.

[24] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "drmt: Disaggregated programmable switching," in *SIGCOMM Conference*. ACM, 2017.

[25] C. d. B. S.D. Conte, *Elementary Numerical Analysis: An Algorithmic Approach Updated with MATLAB*, ser. Classics in Applied Mathematics. SIAM-Society for Industrial and Applied Mathematics, 2018.

[26] A. networks, "Aps networks bf6064x-t." [Online]. Available: https://www.aps-networks.com/wp-content/uploads/2021/07/210712_APS_BF6064X-T_V04.pdf

[27] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *NSDI*. USENIX Association, 2015.

[28] P4lang, "p4lang behavioral-model." [Online]. Available: https://github.com/p4lang/behavioral-model

[29] S. Garera, N. Provos, M. Chew, and A. D. Rubin, "A framework for detection and measurement of phishing attacks," in *Proceedings of the 2007 ACM Workshop on Recurring Malcode*. Association for Computing Machinery, 2007.

[30] V. Anandkumar, "Malicious-url detection using logistic regression technique," *International Journal of Engineering Business Management*, vol. 9, Dec 2019.

[31] D. C. MacFarland, C. A. Shue, and A. J. Kalafut, "Characterizing optimal dns amplification attacks and effective mitigation," in *Passive and Active Measurement*, J. Mirkovic and Y. Liu, Eds. Springer, 2015.

[32] S. M. Tabatabaie Nezhad, M. Nazari, and E. Avazkonandeh Gharavol, "A novel dos and ddos attacks detection algorithm using arima time series model and chaotic system in computer networks," *IEEE Communications Letters*, vol. 20, 2016.

[33] A. H. Yaacob, I. K. T. Tan, S. F. Chien, and H. Tan, "Arima based network anomaly detection," *2010 Second International Conference on Communication Software and Networks*, 2010.

[34] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 561–575.

[35] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 576–590.

[36] A. Laraba, J. François, I. Chrisment, S. R. Chowdhury, and R. Boutaba, "Defeating protocol abuse with p4: Application to explicit congestion notification," in *IFIP Networking 2020*, 2020.

[37] G. Bianchi, M. Welzl, A. Tulumello, F. Gringoli, G. Belocchi, M. Faltelli, and S. Pontarelli, "Xtra: Towards portable transport layer functions," *IEEE Transactions on Network and Service Management*, vol. 16, no. 4, pp. 1507–1521, 2019.

[38] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: query-driven streaming network telemetry," in *SIGCOMM Conference*. ACM, 2018.

[39] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. Association for Computing Machinery, 2019.

[40] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *International Symposium on Computer Architecture (ISCA)*. ACM, 2019.

[41] M. M. J. K. W. B. Agrawal, "Forwarding element data plane performing floating point computations," 2021, uS Patent US10986042B2. [Online]. Available: https://patents.google.com/patent/US10986042B2/en

[42] Y. Yuan, O. Alama, J. Fei, J. Nelson, D. R. K. Ports, A. Sapio, M. Canini, and N. S. Kim, "Unlocking the power of inline Floating-Point operations on programmable switches," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 2022.

[43] A. Voellmy, S. Chen, X. Wang, and Y. R. Yang, "Magellan: Generating multi-table datapath from datapath oblivious algorithmic sdn policies," in *ACM SIGCOMM Conference*, ser. SIGCOMM '16. ACM, 2016.

[44] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *12th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, May 2015.

[45] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," *ACM SIGCOMM Conference*, 2016.

[46] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A language for managing network resources," *IEEE/ACM Transactions on Networking*, vol. 26, 2018.

[47] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, “Efficient measurement on programmable switches using probabilistic recirculation,” *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018.

[48] B. Anwer, T. Benson, N. Feamster, and D. Levin, “Programming slick network functions,” in *Symposium on Software Defined Networking (SDN) Research*. Association for Computing Machinery, Inc, Jun. 2015.

Matthews Jose Received a diplôme d’ingénieur in computer science from École Pour l’Informatique et les Techniques Avancées. In 2018, he started work on his Ph.D. at the university of Lorraine collaborating with Orange Labs and the RESIST team at Inria. His current research interests are in programmable dataplanes, in-network computing and real-value computation on network switches.

Dr. Kahina Lazri obtained her Ph.D. in Computer Science from the Sorbonne Paris Nord University, in December 2014. She joined Orange as Researcher in 2015 where she is involved in research projects related to the field of virtualisation, fast networking, and programmable data planes.

Dr. François Jérôme obtained his Ph.D. in Computer Science from the University of Lorraine, France in December 2009. He was then appointed as research associate at the University of Luxembourg. He is now research scientist at Inria in the RESIST team. His main research areas are focused on the use of data analytics techniques for security and also its coupling with network softwarization. In 2019, he received the IEEE Young Professional award in Network and Service Management.

Olivier Festor is Professor in Computer Science at the University of Lorraine and Director of TELECOM Nancy, the Graduate School in Computer Science and Engineering. He is active for more than 25 years in the Network and Service Management scientific community. His research interest are in Network Security, Monitoring, Programming and Configuration, domain in which he published more than 150 papers in the major conferences and journals in the field. Olivier was awarded the IEEE COMSOC Network Operations & Management Technical Committee *Dan Stokesberry award* in 2019.