University of Padova

Department of Mathematics "Tullio Levi-Civita"

*Master Thesis in Cybersecurity*

# Designing a privacy preserving solution on programmable switches

*Supervisor*
Prof. Mauro Conti
University of Padova

*Co-supervisor*
Dr. Eduard Marin Fabregas
Dr. Salvatore Signorello
Telefonica Research, Spain

*Master Candidate*
Cristian Coreggioli

*Student ID*
2005839

*Academic Year*

2023-2024

# Abstract

The use of programmable data planes in network research has led to great innovation in the last decade. With programmable data planes, a network administrator can customise and monitor the lowest-level behaviour of network devices via standard open-specific APIs. This offers great flexibility and control to the people operating the network. Research demonstrates that many applications can be transferred from servers to devices such as switches or network cards, taking advantage of the data plane programmability offered by these devices.

Along that line of research on programmable data planes, this work proposes the implementation of a privacy-preserving mechanism in the network data plane, leveraging the flexibility of programmable switches and the expressiveness of the domain-specific programming language P4. We choose Differential Privacy (DP) as the target privacy-preserving technique, hence, orienting our data plane design to DP-based mechanisms. Mainly, we show how to implement and assemble Floating Point (FP) operations on P4 targets to build a differential privacy mechanism working on a vector of input elements.

Even though data plane programmability is a very active research area, at present, we are not aware of any other work in the literature that presents the implementation of in-switch differential privacy technique. We believe that our initial effort sheds some important light on the challenges and trade-offs to devise such in-network functionality.

# Contents

# Listing of figures

# Listing of tables

# Listing of acronyms

| | |
|---|---|
| **ACL** | Access Control List |
| **ALU** | Arithmetic Logic Unit |
| **API** | Application Programming Interface |
| **CDF** | Cumulative Distribution Function |
| **CPU** | Central Processing Unit |
| **DDoS** | Distributed Denial of Service |
| **DP** | Differential Privacy |
| **FL** | Federated Learning |
| **FP** | Floating Point |
| **KB** | Kilo-Byte |
| **LoC** | Lines Of Code |
| **LUT** | Look-Up Table |
| **M/A** | Match/Action |
| **NOS** | Network Operating System |
| **NPU** | Network Processing Unit |
| **ONOS** | Open Network Operating System |
| **OS** | Operating System |
| **PISA** | Protocol Independent Switch Architecture |
| **PSA** | Portable Switch Architecture |
| **SDN** | Software Defined Network |
| **SRAM** | Static Random Access Memory |
| **STD** | Standard Deviation |
| **TCAM** | Ternary Content Addressable Memory |

# 1 Introduction

In recent years, with the surge of novel data plane programming technologies, research enabled by programmable networks has evolved considerably. The concept of programmability within the network data plane denotes the capacity of a network device to provide access to its fundamental packet processing logic to the control plane via a standardized Application Programming Interface (API), enabling its efficient and comprehensive reconfiguration [1]. This boosts innovation in networks, as network administrators and/or operators can more rapidly reconfigure the behaviour of network devices according to their specific requirements. In fact, they can implement new functionalities just by using the device's programming interface, avoiding to wait for the usual longer production cycles of network devices to roll out new functionalities.
Through data plane programming, researchers could show that network devices can do way more than only forward packets, they prove to be suitable also for "in-network computing tasks", where application-level logic can be offloaded to network devices (e.g. switches, NICs). For example, Zilberman et al. [2] explore the potential of programmable switches for in-network classification, by mapping machine learning models to match-action pipelines. Jung et al. [3] propose a novel in-switch Access Control List (ACL) system to assist an autonomous defence mechanism based on static and dynamic ACL rules. Another example proposed by Liu et al. [4] is a switch-native approach to detect volumetric DDoS by running detection and mitigation functions entirely on switches.
Even though data plane programmability is a very active research area, at present there are not many attempts to leverage this technology to assist privacy-preserving techniques for network

applications. As a consequence, this project aims to leverage the capabilities of programmable data plane devices and the P4 language to explore the design of an in-network privacy-preserving mechanism. Differential privacy is widely used in contexts where privacy-preserving data is crucial. For example, it is used in distributed machine learning applications to increase the privacy of training data and nowadays it is common to apply Machine Learning (ML) to sensitive training datasets.
We target differential privacy as a class of privacy-preserving techniques for our work. We design a P4 program that implements a DP mechanism for a switch target, detailing the implementation challenges and the design trade-offs. We also perform a preliminary evaluation on a software switch target that sheds further light on the design of such a P4 program.

The rest of the document is structured as follows: Chapter 2 provides the necessary background to understand the overall project; Chapter 3 presents our core contribution, that is, the design of a differentially private technique on a P4-progammable switch target, illustrating the related design challenges and the details of our proof-of-concept implementation; Chapter 4 reports the evaluation of our P4 implementation of the targeted differential privacy mechanism; Chapter 5 concludes this document by presenting a few ideas for future work on this project.

# 2
# Background

This chapter provides the main background knowledge necessary to understand the overall project. First, it describes the main characteristics of the Software-Defined Network (SDN) paradigm in Section 2.1, to outline architecture and differences compared to traditional computer networks. Then, it focuses on the data plane layer of SDN, as it is the crucial part of this thesis work, by going from a fixed to a programmable data plane. Afterwards, Section 2.2 introduces P4, a high-level language for data plane programming, outlining its strengths and weaknesses. Finally, Section 2.3 illustrates differential privacy by providing a mathematical definition, basic techniques and a use-case application.

## 2.1 The SDN paradigm

The idea of this project builds upon a relatively recent network paradigm called SDN. To fully understand its impact, it is useful to start by looking at how networks were operated and managed in the past.

### 2.1.1 Traditional networks

Historically, network devices were structured as *vertically integrated solutions*, including everything from the underlying hardware to the Operating System (OS) running on that hardware, to the application itself. Those devices have proprietary control, data and management planes.

Moreover, the control plane is highly tied to the data plane.

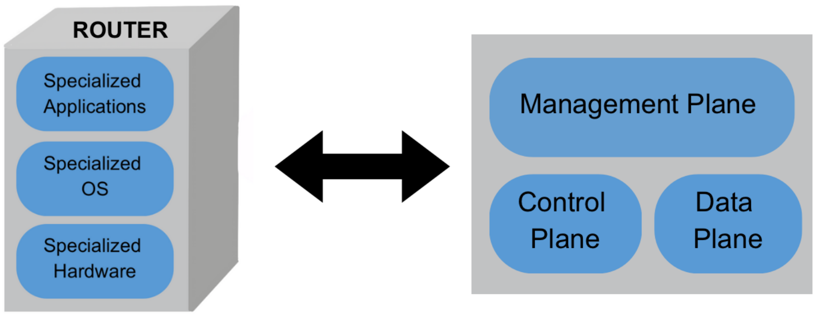


**Figure 2.1:** Traditional router

For example, let us look at a router as a network device (Figure 2.1). The router's control plane can be seen as the specialized OS that takes decisions on which path and protocols the packets have to use to reach the destination. The process of creating a routing table and drawing the network topology are examples of control plane functions. Instead, the data plane forwards packets from one interface to another according to the control plane logic.
A tight binding between control and data plane results in reducing flexibility, hindering innovation and evolution of the networking infrastructure, as a network administrator must configure each network device separately using low-level and vendor-specific commands. Moreover, with this approach is very difficult to add new functionalities to devices [5].

### 2.1.2 Software-Defined networks

SDN aims to transform the market from hardware-centric and static networks to software-defined and highly programmable networks. A simple definition of SDN is: *"A network in which the control plane is physically separate from the data plane, and a single control plane controls several forwarding devices"* [5]. The main contributions can be summarized into four main pillars:

1. Decoupling control and data planes. In practice, two data structures need to be maintained: the control plane maintains a *routing table* with the information to select the best route while the data plane maintains a *forwarding table* optimized for fast packet processing (Figure 2.2). This disaggregation makes it possible for different parties to be responsible for each plane and it implies the need for a well-defined *forwarding abstraction*, that defines a way for the control plane to instruct the data plane. This separation

should be codified in an open interface and the most common one is called *Open Flow*. One consequence of such separation is that data plane components (like switches) become simple packet-forwarding devices with all intelligence implemented via software by the control plane. In principle, this disaggregation means that a network operator should be able to purchase their control plane from vendor X and their data plane from vendor Y.

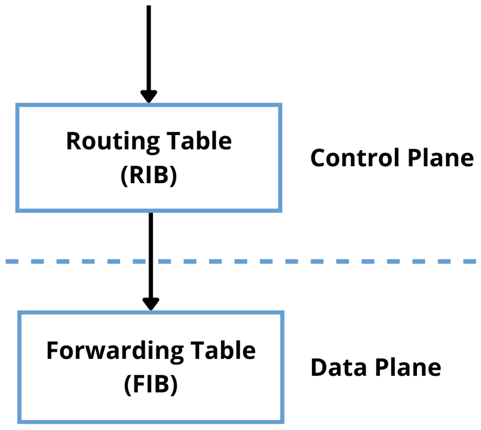


**Figure 2.2:** Control and Data planes separation

2. *Flow-based* forwarding decisions. The *Open Flow* interface introduces a new way to specify the forwarding behaviour, via *Flow rules* and *Flow tables*. A "flow" can be defined as a sequence of packets from source to destination and, in the SDN context, all packets of a flow receive identical service policies at the forwarding device. A *Flow rule* is a Match-Action pair such that any packet that matches the first part of the rule should have the associated action applied to it.
   For example, a simple flow rule might specify that any packet with destination address "D" must be forwarded to output port "I". Each switch then maintains a *Flow table* to store the set of flow rules the controller has passed to it (Figure 2.3).
   In flow-programming, decisions can be based on multiple factors like IP, MAC, and TCP port. It allows more flexibility, as rules can be defined in a more granular way, compared to a destination-based approach where decisions are based only on address destination;

3. Moving the controller logic to an external entity called *SDN controller / Network OS* (NOS). The NOS is a software platform that runs on a commodity server and provides the essential resources and abstractions to facilitate the programming of forwarding devices based on a logically centralized abstract network view. Logically centralized means

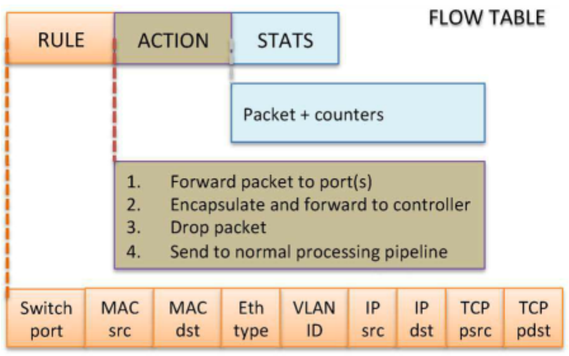


**Figure 2.3:** Flow table

that a globally unique data structure is maintained in the controller, however, it may be implemented in a distributed way over different servers. This is important both for scalability and availability. Therefore, its purpose is similar to the one of a traditional operating system.

4. The network is programmable through a set of software applications (Control Apps) that express the desired network behaviour without actually being responsible for implementing it (Figure 2.4).

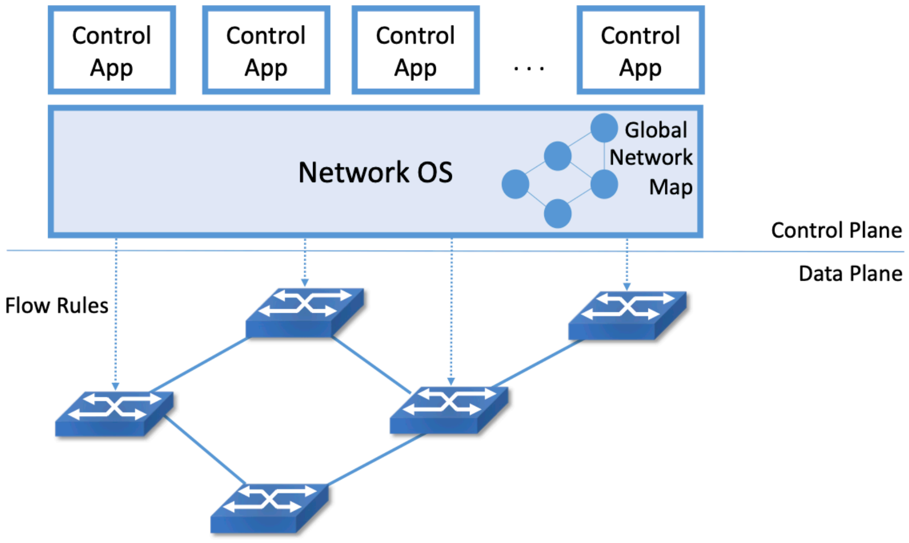


**Figure 2.4:** SDN workflow

Figure 2.5 illustrates those two kinds of networks side by side. Each device in the traditional network (on the left) has its control plane and data plane, while in the SDN one the control plane is separated from each device and placed into one logically centralized entity, called *SDN Controller*. By opening vertically integrated, closed network devices, SDN creates opportunities for innovation that would not otherwise be available. Moreover, by opening up interfaces, it shifts the control from vendor-specific to the network operator, as such interfaces are "public" and can be modified using APIs [6].

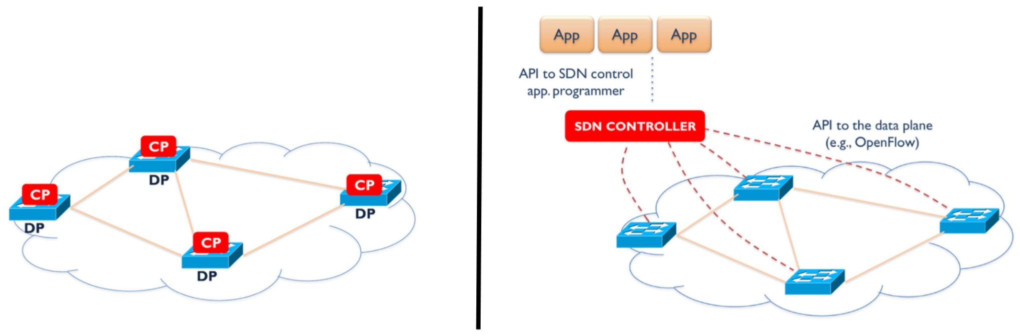

CP
DP
App
App
App
API to SDN control app. programmer
SDN CONTROLLER
API to the data plane (e.g., OpenFlow)

**Figure 2.5:** Traditional VS Software-Defined Network

### 2.1.3 SDN ARCHITECTURE

The Figure 2.6 shows a high-level system design of an SDN architecture on the right with the separation of the network planes on the left [5].

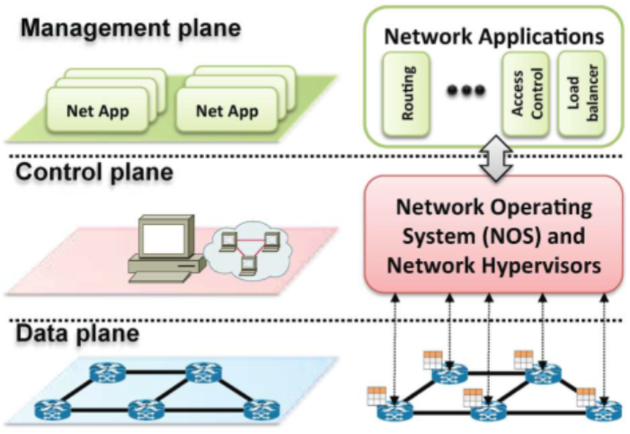


**Figure 2.6:** SDN layers

A given SDN network is composed of three planes:

1. Management plane: a set of applications (called Network / Control Apps) that manage and configure the overall network behaviour. Essentially, a management application defines the policies, which are appropriately translated by the NOS and sent to forwarding devices. Examples of applications are routing, firewalls and load balancers;

2. Control plane: it is called *SDN Controller / Network OS*. The NOS is a platform for configuring and controlling a network of switches. It runs off-switch as an external entity. Central to this role is the responsibility for monitoring the state of those switches (for example detecting port and link failures), maintaining a global view of the topology that reflects the current state of the network, and making that view available to any interested network application;

3. Data plane: it is composed of a network of forwarding devices which are responsible to forward network packets based on instructions received from the control plane. It plays a critical role in the efficient and reliable forwarding of network traffic based on received policies and instructions. The initial SDN concept assumed fixed data plane functionalities, however the research in this field has recently evolved. The next section explains this evolution in more detail.

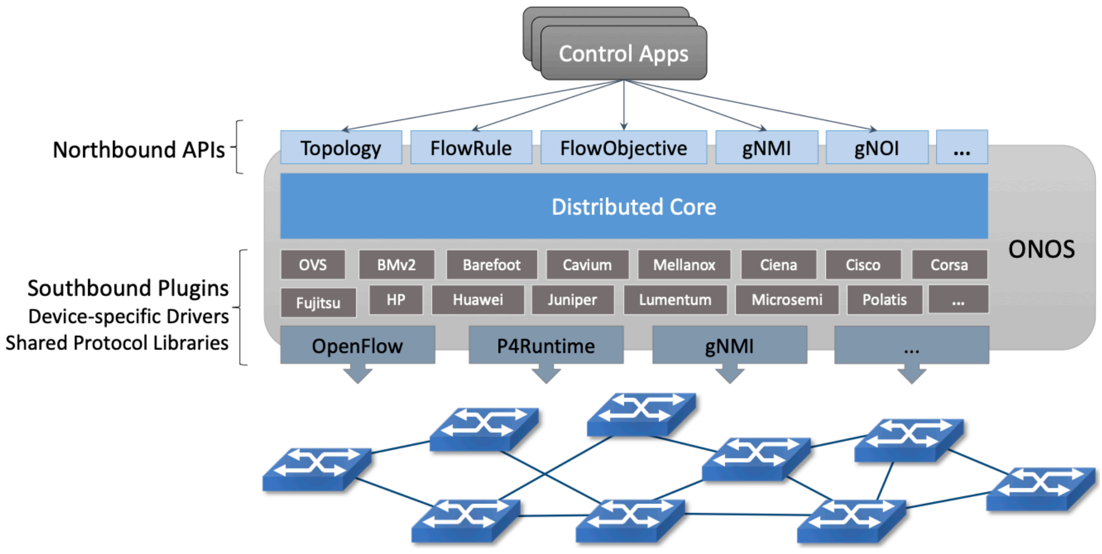


**Figure 2.7:** ONOS

As a practical example, Figure 2.7 illustrates the open-source SDN controller platform *Open Network Operating System (ONOS)*. ONOS maps the desired behaviour of control applications onto the configuration instructions that need to be loaded onto each switch in the network. It is also important to consider that information flows both "down" and "up" through ONOS. The principal components of an SDN controller like ONOS are:

- Northbound Interfaces: a collection of interfaces used by "control apps" to stay informed about the network state and to control the network data plane. The union of all northbound APIs must be sufficient to configure, operate, and control the network. These APIs allow developers to build custom network applications and services on top of the ONOS platform;

- Distributed Core: ONOS is designed as a distributed system, with a cluster of controller nodes working together to provide scalability and fault tolerance. This distributed architecture allows ONOS to handle large-scale network deployments with high availability. It is responsible for managing the network state and notifying applications about relevant changes in that state;

- Southbound Interfaces: multiple southbound protocols are supported, including OpenFlow, NETCONF, and P4Runtime. They allow to control a wide range of network devices from various vendors.

### 2.1.4 Programmable Data Plane

With the introduction of the first generation of SDN, networks had programmable control plane but fixed data plane functionalities. Still, it was a huge improvement compared to traditional networks of the past. However, recently the network community further improved the architecture of SDNs by introducing programmable data planes.
This section focuses on programmable switches and their main characteristics [5]. In the SDN context, switches are commonly called *bare-metal switches*. They are pure hardware devices as they do not have any pre-installed Operating System (OS). They can be called also *open switch*, the difference is that the latter has an OS pre-installed even if it is not tied to the hardware as in a *traditional switch*. It is equivalent to buy a package with a bare-metal switch and an operating system at the same time. The interconnected set of switches is the underlying hardware for SDN.
Figure 2.8 gives a high-level schematic of a *bare-metal switch*. It is made of the following components:

1. Network Processing Unit (NPU): It is a merchant silicon switching chip optimized to parse packet headers and make forwarding decisions. In the Figure 2.8 the NPU is a combination of ASIC-based forwarding pipeline that implements a series of match-action tables and SRAM-based memory to buffer packets while they are being processed. It is

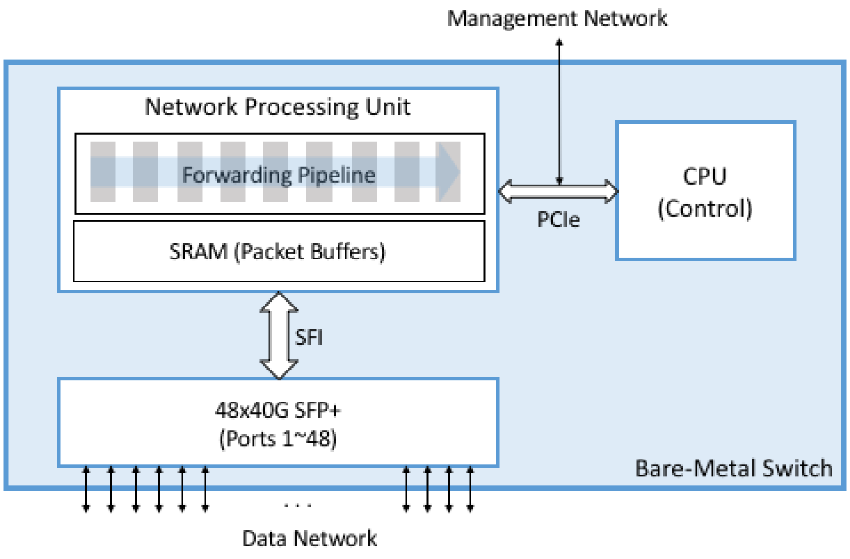


**Figure 2.8:** Bare-metal switch

important to point out that the ASIC pipeline is designed to perform specific networking tasks efficiently, such as packet switching, forwarding table lookups, and quality of service (QoS) enforcement and it is "vendor-specific";

2. Central Processing Unit (CPU): It is a general-purpose processor that controls the NPU and it is where the Switch OS is running. It is the part that exports the API that allows the NOS to control the data plane;

3. Other commodity components that make this all practical like transceiver modules that take care of all the media access details.

Look now at the *forwarding pipeline* of the NPU in detail, since it is a crucial element in programmable switches. High-speed switches use a multi-stage pipeline to process packets. This architecture divides the execution of a packet processing task into several sequential stages, with each stage performing a specific subset of operations on the input data before passing it to the next stage. Each stage operates concurrently with other stages, allowing for parallel processing and improving overall throughput. In practice, it means that multiple packets can be processed at the same time. The main distinction in how a given NPU implements this pipeline is whether the stages are fixed-function or programmable:

- **Fixed-function pipelines**: the switch has a pre-defined packet processing logic whose processing functions are implemented and cannot be modified or reprogrammed by the network administrator. While *flow rules* are general enough to say what forwarding behaviour the controller wants to program into a switch, switches do not necessarily have the capacity to implement that functionality efficiently. As networks evolve, it is reasonable to think that switches would implement new functionalities. However, to incorporate new features, traditional switches must be re-designed. This process can be very long (it generally takes some years);

- **Programmable pipelines**: they allow a network administrator to define and customize the packet processing behaviour using a high-level programming language such as P4. The primary benefit lies in its ability to provide a significant degree of flexibility, enabling network operators to implement new features, protocols, and optimizations as needed. These pipelines can easily adapt to network changes and can be updated remotely without requiring the deployment of new hardware.

As stated previously, the switch can be built using different ASICs. Because of this, a general way to represent the pipeline called *abstract pipeline* is required, together with a definition of how the abstract pipeline maps onto the physical pipeline (that is the vendor-specific ASIC). Such *abstract pipeline* can be specified using the dedicated programming language P4. Some standard architectural models are used for this purpose (PISA, V1 model, PSA, etc.). The introduction of such an architectural model enables the portability of the same forwarding pipeline (P4 program) across multiple targets (switching chips) that support the corresponding architecture model. Therefore, programmable switches are a huge step forward from a few years ago as now it is possible to program both control and data plane layers of an SDN system.

## 2.2 The P4 language

This section introduces P4, a high-level language for programmable switches used in this project. It is a language for *Programming Protocol-independent Packet Processors* firstly introduced in 2014 [7]. The main goals in designing P4 stated in the original paper are:

1. **Switch reconfigurability**: the ability to change and add new functionalities via software once the switch is already deployed. The main idea is to prevent redesigning new hardware (a long and expensive process) once novel functionalities must be implemented on the switch;

2. **Protocol independence**: Switches should not be tied to any specific network protocols (like Ethernet, IPV4, etc.) and any specific packet format. Rather, the programmer should be able to specify custom protocols and custom packet formats;

3. **Target independence**: Programmers should be able to write P4 programs independently of the underlying hardware. Indeed, the programmer does not need to know the details of the underlying switch as a compiler will be responsible for the translation from the P4 program into the actual target switch low-level code.

### 2.2.1 Abstract model

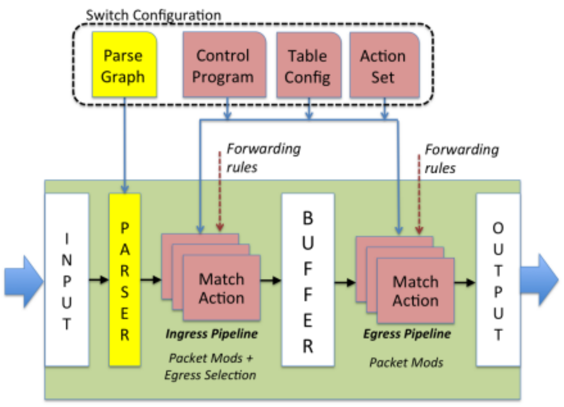


**Figure 2.9:** P4 Abstract model

The Figure 2.9 shows the original *abstract model* proposed by the P4 original paper that generalizes how packets are processed in different forwarding devices (like Ethernet switches, load-balancers, routers) and by different technologies (like fixed-function switch ASICs, NPUs, software switches, FPGAs). Using such a model as a reference, the authors of the language could more easily devise some syntax and programming model for P4 that would allow programmers to create target-independent programs and compilers to map programs to a variety of different forwarding devices. The above model is controlled by two main types of operations:

- **Configure** operations: program the parser, set the order of Match-Action (M/A) stages, and specify the header fields processed by each stage;
- **Populate** operations: add (and remove) entries to the M/A tables that were specified during configuration phase.

The packet processing workflow for such a model is the following: arriving packets are first handled by the parser. The parser recognizes and extracts fields from the packet header. The extracted header fields are then passed to the Ingress pipeline. There are two distinct M/A pipelines, one for the Ingress and another for the Egress pipeline. The difference is that during Ingress process tables determine the egress port and the queue into which the packet is placed, while during Egress process the tables form per-instance modifications to the packet header (for multi-cast copies). Once the packet goes through both Ingress and Egress pipelines, it is recomposed by the Deparser and sent in output. Packets can carry additional information between stages, called metadata, which is treated identically to packet header fields.

### 2.2.2 Protocol-Independent Switch Architecture

At an architectural level, the programmable pipeline is often referred to *Protocol Independent Switching Architecture (PISA)* as shown in Figure 2.10.

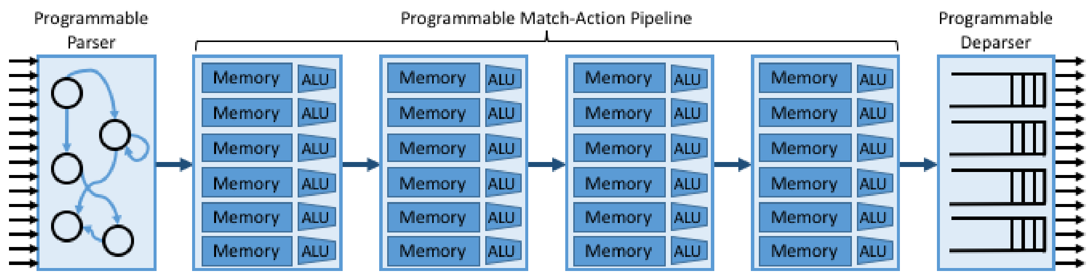


**Figure 2.10:** PISA architecture

PISA has three main components:

1. **Parser**: it identifies and extracts header fields from specific locations in the packet;
2. **Match-Action pipeline**: it is a sequence of Match-Action (M/A) units each of which is programmed to match and act upon one or more identified header fields. The memory blocks shown in the Figure 2.10 are typically built using a combination of SRAM and TCAM. The difference is that TCAM is more expensive and power-hungry than SRAM, but it supports wildcard matches.
   The ALU component is responsible for implementing "actions". Possible actions include modifying specific header fields (like decrementing a TTL), incrementing or clearing various counters internal to the switch, and setting user/internal metadata;
3. **Deparser**: it reconstructs the representation for each packet from all the in-memory header fields produced as an output by the M/A pipeline.

Writing low-level target-specific code for a switch architecture is usually not an easy task, however, the desired packet processing behaviour of a PISA switch can be expressed through a high-level program written in P4. Then, a vendor-specific compiler is responsible for generating the equivalent low-level program.

Together with PISA, another very famous architecture is the *V1 model* shown in Figure 2.11. This is the architecture used in practice in this project. The main components are: Parser, Ingress pipeline, Checksum verification, Traffic manager, Egress pipeline, Checksum update, and Deparser.

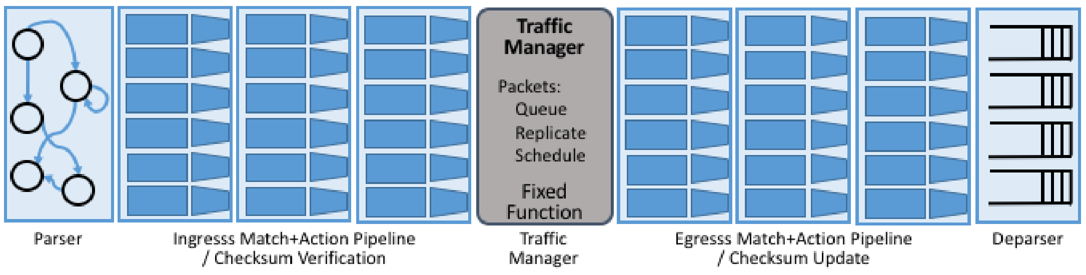

Traffic Manager
Packets:
Queue
Replicate
Schedule
Fixed Function
Parser
Ingresss Match+Action Pipeline / Checksum Verification
Traffic Manager
Egresss Match+Action Pipeline / Checksum Update
Deparser

**Figure 2.11:** V1Model switch architecture

In practice, every P4 program starts with a P4 template with a declaration of the above components. The following P4 code shows only the final block, which is the "main" function that specifies all components to be pulled together to build a complete switch pipeline.

```
/* Switch */
V1Switch(
    MyParser(),
    MyVerifyChecksum(),
    MyIngress(),
    MyEgress(),
    MyComputeChecksum(),
    MyDeparser()
) main;
```

## 2.2.3 Structure of a P4 program

A P4 program contains definitions of the following key components [7]:

- Packet Headers: each protocol header is specified by declaring an ordered list of field names together with their widths, similar to a C structure;

- Parsers: A parser is defined as a state machine that analyzes packet headers to identify and extract protocol fields. The extracted field values are sent to the M/A tables for processing;

- Tables: they are the core mechanism to perform packet processing. They are considered the core primitives. Tables are defined to match specific packet headers or fields and take corresponding actions based on the matches. A table can be seen as a key-action pair with pre-loaded entries by a controller. Then, during execution, once a specific field matches the key of a table entry, the corresponding "action" is triggered;

- Actions: P4 supports the construction of complex actions from simpler protocol-independent primitives. Actions can include forwarding, dropping, modifying headers, or invoking custom processing logic;

- Control blocks: Once tables and actions are defined, a control block determines the order of M/A tables that are applied to a packet. It is similar to a C function (without loops).

### 2.2.4 Language Specification and Current Version

The P4 language is developed by the *p4.org* consortium [8]. The current version is the $P4_{16}$. It is a relatively simple, statically-typed programming language, with a syntax based on C, designed to express computations on network packets. This language is designed to describe data-path packet processing logic targeting high-throughput networking devices, but it has several constraints [9]. For example:

- P4 does not offer support for recursive functions (no loops);

- P4 does not support dynamic memory and pointers. Resource consumption can be statically estimated (at compile-time);

- Only basic arithmetic operations are supported (no floating point).

In summary, P4 is a dedicated programming language for programmable switches, where the main switch functionality is to forward packets. Therefore, P4 can be used to implement more sophisticated in-switch operations but they must cope with the above limitations.

## 2.3 Differential privacy

This section introduces *Differential Privacy* a formal mathematical framework to ensure the privacy of individuals' data in the context of statistical analysis of aggregated datasets. First, it outlines the high-level intuition and purposes, then provides a more formal definition of the framework and explains some of the differentially-private techniques mostly used nowadays. Finally, the application of differential privacy to a practical use case is analyzed.

### 2.3.1 Intuition and definition

The Cambridge dictionary defines *privacy* as:

*"someone's right to keep their personal data and relationships secret"*

*Differential Privacy* is a formal mathematical definition of privacy. In the simplest setting, an algorithm that analyzes a dataset with sensitive information as an input and computes statistics about it, is said to be *differentially private* if, by looking at the output, one cannot tell whether any individual's data was included in the original dataset or not. In other words, the output of the algorithm hardly changes when a single individual joins or leaves the dataset.
The mathematical theory of *Differential Privacy* (DP) guarantees that anyone viewing the result of a differentially private analysis will make the same inference about any individual's private information. DP guarantees are summarized in [10]:

- It protects an individual's information as if his information were not used in the analysis at all, that is, the outcome of a differentially private algorithm is approximately the same whether the individual's information was used or not;
- It mathematically ensures that using an individual's data will not reveal any personally identifiable information, or even whether the individual's information was used at all.

More formally, the mathematical definition of *Differential Privacy* provided by Dwork et al. [11] is the following:

Let $\mathcal{A} : \mathcal{X} \rightarrow \mathcal{O}$ be a randomized algorithm. $\mathcal{A}$ is $(\varepsilon, \delta)$-differentially private, if for any $o \subseteq \mathcal{O}$, and for any $D_1, D_2 \in \mathcal{D}$ s.t. $D_1 \sim D_2$:

$$\Pr[\mathcal{A}(D_1) \in o] \leq e^{\varepsilon} \cdot \Pr[\mathcal{A}(D_2) \in o] + \delta \tag{2.1}$$

Where $\varepsilon \geq 0$, $\delta \in [0,1]$ are called **privacy parameters** and the relation $D_1 \sim D_2$ means that $D_1$ and $D_2$ are **neighbours**.

## Neighbours

The concept of "**neighbour datasets**" allows to quantify how much the output of an algorithm changes when a single individual's data is added or removed from the dataset. Two datasets $D$ and $D'$ are said to be "neighbouring" if they differ by at least one individual's data entry. Formally, let $D = (x_1, x_2, ..., x_n)$ and $D' = (x'_1, x'_2, ..., x'_n)$ be two datasets, where $x_i$ and $x'_i$ represent the data entries for individual $i$ in datasets $D$ and $D'$, respectively. Then, $D$ and $D'$ are considered neighbouring if there exists an index $j$ such that $x_i = x'_i$ for all $i \neq j$ and $x_j$ and $x'_j$ differ by at least one unit. This means datasets $D$ and $D'$ are nearly identical except for one individual's data entry.

## Privacy parameters

The combination of $(\varepsilon, \delta)$ determines how strict or loose the DP guarantees are:

- $\delta$ is the probability of information accidentally being leaked, often referred to as the privacy parameter for "unusual events" or "worst-case scenarios". Lower values indicate stronger privacy guarantees, and typically, $\delta$ is set to a very small value such as $\delta < 1/n$ where $n$ is the database size. Based on the $\delta$ value, we may have [12]:
  - *pure DP*: the value of $\delta = 0$;
  - *approximate DP*: the value of $\delta > 0$ and it is consequently written $(\varepsilon, \delta)$-DP.

- $\varepsilon$ is commonly called *privacy loss parameter*. It quantifies the extent of the deviation between two output queries. Differential privacy requires only that the output of queries computed on different datasets remain approximately the same, that is, it permits a slight deviation between the two output values. Let us consider Figure 2.12 where there are two equal input datasets, where $D_1$ contains a row with the element X while $D_2$ does not. We compute the same query for the two datasets and the output results can differ at most for $\varepsilon$ [10].
  Choosing a value for $\varepsilon$ can be thought of as tuning the level of privacy protection required. A smaller value of $\varepsilon$ results in a smaller deviation between $output_1$ and $output_2$, therefore associated with stronger privacy protection but less accuracy.
  For example, when $\varepsilon = 0$, the **Analysis 1** mimics the **Analysis 2** perfectly, for all the elements in the datasets. Yet when $\varepsilon$ is set to a small number such as $\varepsilon < 2$, the deviation

between **Analysis 1** and **Analysis 2** will be small, providing strong privacy protection. Other ranges used in practice are $\varepsilon \in [2, 10]$ for relax but possibly meaningful privacy and if $\varepsilon > 10$ is almost equivalent to have no privacy;

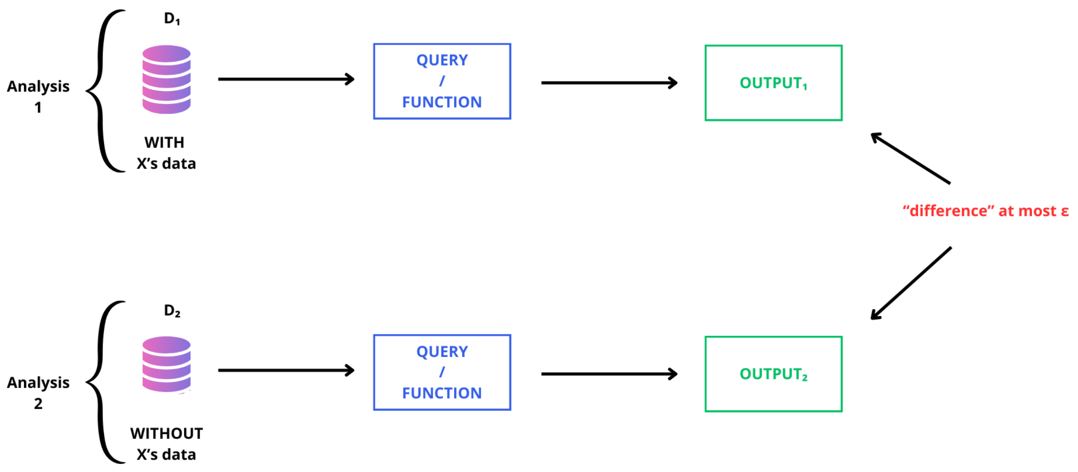


**Figure 2.12:** $\varepsilon$ privacy loss parameter

### 2.3.2 Basic techniques

After describing the mathematical framework behind differential privacy in Section 2.3.1, this section describes some of the main techniques used to achieve DP in practice, namely *Laplace* and *Gaussian* mechanisms [13]:

- **Laplace**: The Laplace mechanism is a method for achieving differential privacy by adding carefully calibrated noise to the output of a function (or query). It is often used in scenarios where the function being computed involves the aggregation of sensitive data. The following points illustrate the mechanism:

    1. *Function evaluation*: A function $f$ takes a dataset $D$ as input and produces a real-valued output $f(D)$. The goal is to release the output of this function in a differentially private manner. This function could be any computation or statistical operation performed on the dataset, such as counting the number of individuals satisfying a certain condition, calculating the mean or sum of a numerical attribute, etc;
    2. *Sensitivity calculation*: It measures how much the output of the function can change when a single individual's data is added or removed from the dataset. It is

denoted as $\Delta f$. Mathematically, it is defined as (where $D_1$ and $D_2$ are **neighbours**):

$$\Delta f = \max_{D_1, D_2} \|f(D_1) - f(D_2)\| \tag{2.2}$$

3. *Noise addition*: As stated in the mechanism's name, Laplace-distributed noise must be added to the function output. The Laplace distribution is characterized by its Probability Density Function (PDF), which is given by:

$$\text{Laplace}(x|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right) \tag{2.3}$$

   where $\mu$ is the location parameter (mean) and $b$ is the scale parameter (related to the spread of the distribution). The larger the value of $b$ is, the more spread out the noise will be;

4. *Privacy guarantee*: Precisely, the final noisy output is given by:

$$f(D) + \text{Laplace}(0, \Delta f/\varepsilon) \tag{2.4}$$

   where $\Delta f/\varepsilon$ represents the scale of the Laplace distribution, and it is chosen such that as $\varepsilon$ decreases (providing more privacy), the scale of the noise increases. Therefore, the amount of noise added to the output is determined by the sensitivity of the function $\Delta f$ and a privacy parameter $\varepsilon$.

In summary, the Laplace mechanism first generates Laplace-distributed noise and, afterwards, adds such noise to the output of a function to achieve differential privacy.

- **Gaussian**: The Gaussian mechanism follows a similar process to the Laplace one. However, the noise added to the function's output $f(D)$ is sampled from a Gaussian distribution. A Gaussian distribution is characterized by the following PDF:

$$\text{Gaussian}(x|\mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.5}$$

  where $\mu$ is the mean and $\sigma$ is the standard deviation. The final noisy output will be:

$$f(D) + \text{Gaussian}(0, \sigma^2) \tag{2.6}$$

  Note that in the Gaussian mechanism, the noise is scaled by the standard deviation parameter $\sigma^2 = \frac{(\Delta f)^2 \cdot 2 \cdot \log\left(\frac{1.25}{\delta}\right)}{\varepsilon^2}$. Similarly to Laplace, the amount of noise is determined by the sensitivity of the function $\Delta f$ and the privacy parameters $\varepsilon$, $\delta$.

The Figure 2.13 shows at a high level the process for achieving DP using either the *Laplace* or the *Gaussian* mechanism. There is a dataset $D$ containing sensitive information about individuals. Then, a general query (or function) is applied to the entire dataset, for example computing the mean. The noise is generated according to the chosen mechanism, that is, Laplace or Gaussian. Finally, the generated noise is added to the output previously computed by the function $f(D)$ to produce the final output, called "noise output" in Figure 2.13.

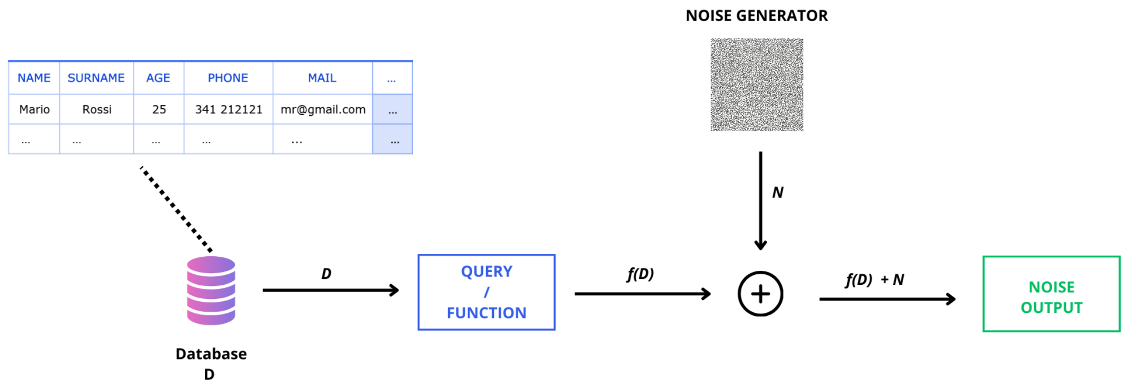


**Figure 2.13:** DP general workflow

### 2.3.3 Applications

Differential Privacy (DP) can be applied to a wide range of applications across various domains where privacy-preserving data analysis is crucial. For example, in the context of healthcare applications sensitive data are used, so privacy should be protected. Or yet, these days it is very common to apply machine learning to sensitive training datasets. In fact, this section presents federated learning, a framework where it is very relevant to protect the privacy of the training data through techniques like DP.

#### Privacy leakage in Federated Learning

Federated Learning (FL) is a machine learning approach where a single model is trained in a decentralized way across multiple clients. Instead of collecting data in a central repository, the core idea of FL is to train multiple models locally by each client, without exchanging their training data. One of the main motivations for performing distributed training is exactly to protect the privacy of each client's training data, as in some applications such data can contain sensitive information (e.g. healthcare, finance, transportation, etc.) [14].
We illustrate a typical FL workflow with one server and multiple clients in Figure 2.14:

1. The server broadcasts the "base model" to each client;

2. Each client trains the “base model” with its local data in parallel and sends only *model parameters* to the server, while keeping their training data private;

3. Once the server has received all models from clients, it aggregates those to build a unique model, that becomes the new “base model”.

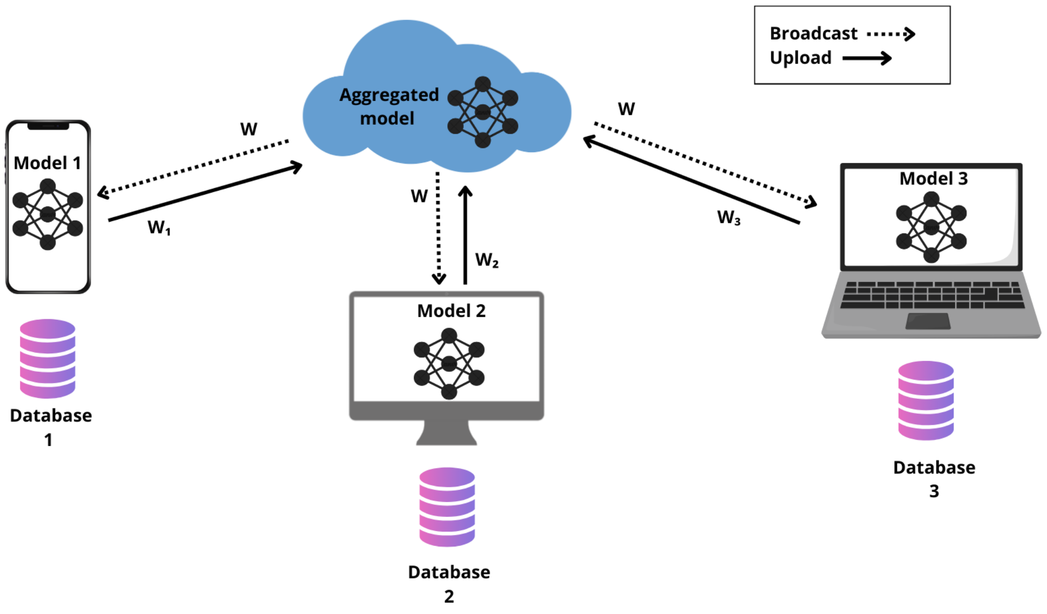


**Figure 2.14:** Federated Learning workflow

The process is repeated several times, as the collaborative training requires many iterations between clients and the server to build the final model.
Even if, in FL private data do not leave clients, researchers have proved that it is still possible for an adversary or a curious observer to infer information on the training data. This type of attack is called *Membership Inference Attack* and it allows an adversary to infer information about the training data only by observing the model updates exchanged by the clients with the server. Various variants of the *Membership Inference Attack* have already been proved, Suri et al. (2023) [15] presents a subject membership inference attack while Zari et al. (2021) [16] presents a passive membership inference attack.

Therefore, only keeping training data private for clients is not sufficient to achieve a strong privacy guarantee in a federated learning system. For this reason, further privacy mechanisms must be applied to protect training data from the above attacks. One mechanism used to further protect the privacy of training data is to use differential privacy on the gradient updates [17]. In this way, the information carried by model updates is not linked to specific clients, even if an adversary may be looking at it.

# 3 Our contribution

This project aims at leverage the capabilities of programmable switches to implement differential privacy directly in the network data plane. We are not aware of any other work, at present, in the literature that presents the implementation of in-switch privacy technique. This chapter presents our core idea for a differentially private technique on a P4 switch, together with the related challenges and the implementation details. The chapter starts by describing at high-level the targeted Differential Privacy (DP) technique, then it presents the main implementation challenges and how those have been addressed. Finally, the chapter describes how the target DP technique can be implemented in P4.

## 3.1 Targeted Differential Privacy Mechanism

This section explains the specific mechanism that we have selected for an implementation of differential privacy on a P4 programmable switch. Let us consider the SDN model described in figure Figure 3.1. The workflow to implement DP in P4 is the following: the client sends a vector of parameters to the switch, the switch parses the vector, it performs the DP operations on its parameters and it sends a differentially private vector out to a server application. To perform the DP operations, the switch must have the relative look-up tables populated. They are needed to perform FP arithmetic directly on the switch. Therefore, the SDN controller is responsible for populating each required look-up table once the switch has just started.
Let us now focus on the operations for DP that must be implemented in P4 on the target switch.

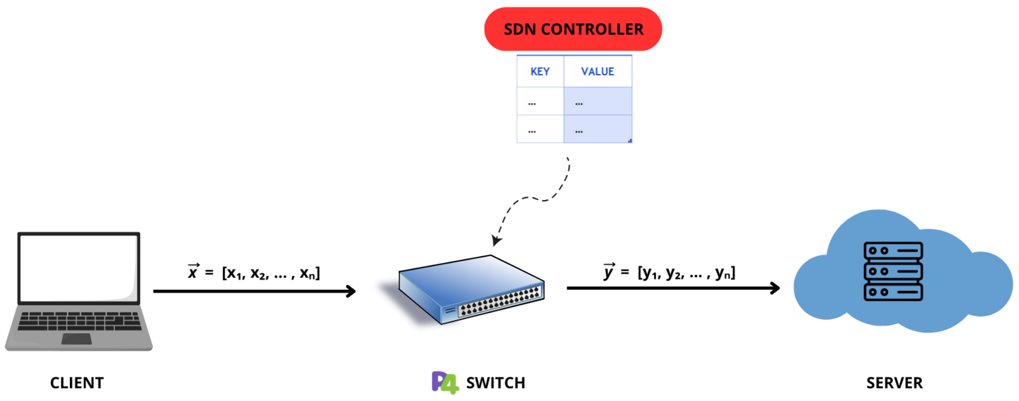

SDN CONTROLLER
KEY
VALUE
...
...
...
...
$\vec{x}$ = [x₁, x₂, ... , xₙ]
$\vec{y}$ = [y₁, y₂, ... , yₙ]
CLIENT
SWITCH
SERVER

**Figure 3.1:** Basic SDN model

We consider input vectors of floating point numbers, assuming this could be the case with *"model parameters/gradient updates"* for Machine Learning (ML)-based applications. Given an input vector ($\vec{x}$) of limited size, the switch executes the following operations, in sequence, to produce a differential private output vector ($\vec{y}$) of the same size:

1. **Clipping the input vector**: "Clipping" is a standard mechanism to limit the magnitude of a set of real values. For example, bound gradients during a training process is a widely adopted technique in ML. Gradients too large may, in fact, lead to instability of the model. This is often done by scaling down the gradients if their norm exceeds a certain threshold [18].

   Given the input vector $\vec{x} = [x_1, x_2, ..., x_n]$, where $n$ is the number of parameters, the switch does the following:

   (a) It computes the clipping coefficient $C_{true}$ as the L1 norm of the input vector:

   $$C_{true} = \|x\|_1 = \sum_{i=1}^{n} |x_i| \tag{3.1}$$

   (b) It computes the clipping coefficient $C_{clip}$:

   $$C_{clip} = min(1, C/C_{true}) \tag{3.2}$$

   where $C$ is the clipping threshold constant. It can be considered as a free hyper-parameter;

   (c) It generates the clipped vector $\vec{x}_{clipped}$:

   $$\vec{x}_{clipped} = \vec{x} * C_{clip} \tag{3.3}$$

2. **Adding noise to the Clipped vector**: This consists of generating a noise vector ($\vec{n}$) according to pre-defined distribution, which is typically *Gaussian/Laplace*. Then, the pre-computed noise is added to the clipped vector to obtain the differential private vector $\vec{y}$:

$$\vec{y} = \vec{n} + \vec{x}_{clipped} \tag{3.4}$$

   As a final result, the switch receives an input vector of parameters $\vec{x}$ and outputs the correspondent differentially private vector $\vec{y}$.

## 3.2 Design Challenges

We had to address the following four main challenges to implement the targeted DP mechanism on a P4 programmable switch.

1. **Dealing with input vectors of variable length**: the P4 language does not have any built-in support for working with vectors of any type. Assuming working with vectors of variable length size also increases the difficulty of implementing the targeted operations on the switch, since P4 does not support loops or recursive functions. We detail about this challenge in Section 3.3;

2. **Noise generation**: the P4 programmable target must be able to generate noise according to a certain distribution. This is required to obtain the values to be added to the clipped vector. We detail about this challenge in Section 3.4;

3. **Floating Point implementation**: the input vector contains floating point values. However, the current P4 language specification and targets do not support floating point types and arithmetic. We detail about this challenge in Section 3.5;

4. **Sequential FP operations**: the main challenge in implementing the targeted differential privacy mechanism described in Section 3.1 is to let the switch perform a pre-defined sequence of floating point operations in the correct order without using loops. This greatly increases the complexity of the P4 program, lookup tables and memory consumption. We detail about this challenge in Section 3.6;

## 3.3 Dealing with input vectors of variable length

In this section, we describe how we generate the P4 program based on the number of input elements. In our targeted DP mechanism, we assume the input vector to be of arbitrary size, consequently, when changing the size of the input vector, the P4 program must change accordingly. Being a language most targeting high-throughput packet processor targets, the P4 language does not provide a built-in type for *vectors* and it does not support loops as many other conventional high-level programming languages do. To store *vectors* in P4 we declare a header with a custom number of fields, where each field is used to store every single element contained in the input vector (an example is in Figure 3.2). In our program, we use fields of 16 bits as input numbers following the IEEE half-precision format. Each field in P4 must be manually written inside a given header and it cannot contain loops. Therefore, we can write a set of P4 instructions for a specific input vector, based on its size, but if we want to use a vector with a different size, we must re-write part of the P4 program.
Assume to use a custom P4 header to store each parameter of the input vector, as shown in the following P4 code (vector of size 2):

```
// Custom header for INPUT VECTOR:
header param_t {
    bit<16> x1;
    bit<16> x2;
}
```

**Figure 3.2:** P4 custom header for an input vector of 2 parameters

As the P4 language does not provide loops, if we decide to use an input vector of four parameters, we must manually change the previous code snippet to become the following:

```
// Custom header for INPUT VECTOR:
header param_t {
    bit<16> x1;
    bit<16> x2;
    bit<16> x3;
    bit<16> x4;
}
```

**Figure 3.3:** P4 custom header for an input vector of 4 parameters

Therefore, it is difficult to adapt a P4 program based on how many elements of the input vector are received within an incoming packet, since that would require to use variable length loops into the control flow of a P4 program.

To overcome this limitation, we use a template engine called *Jinja2* [19]. *Jinja2* is a template engine for Python that simplifies the process of generating code dynamically. It provides a flexible way to combine template code with dynamic data, allowing network administrators to efficiently generate custom programs tailored to specific requirements. The general workflow is to create a template in some programming language (like P4 or Python) that contains placeholder variables and that follows a specific syntax. The placeholders represent the variable number of vector elements that will be replaced with actual values when rendering the template at compile time. Therefore, the final program is generated with the custom data provided at compiled time.

For example, the P4 code in Figure 3.2 and Figure 3.3 have been dynamically generate using the following template code:

```
// Custom header for INPUT VECTOR:
header param_t {
    {% for i in range(1, num_parameters + 1) %}
    bit<16> x{{ i }};
    {% endfor %}
}
```

**Figure 3.4:** P4 custom header's template

Figure 3.4 shows a P4 template with placeholders based on *Jinja2* syntax. The placeholder is represented by the *num_parameters* variable. Its value is set at compiled time and can be changed to generate different final programs. In fact, if *num_parameters*= 2 we obtain the code in Figure 3.2 while if *num_parameters*= 4 we obtain the code in Figure 3.3.

The above example demonstrates how we handle a variable number of parameters in our P4 switch program using this technique. The same technique is also used to dynamically generate a set of P4 instructions, according to input vector size. As an example, consider the "clipping" phase of the DP mechanism described in Section 3.1 where the first step is to compute the L1 norm of the input vector. To compute the L1 norm in P4 we use the same technique, that is, we write the template in Figure 3.5 that will generate a custom number of P4 instructions based on the value of *num_parameters*.

In Figure 3.5, line 3 stores a single parameter in a metadata field while line 4 sets the number to positive by changing the most significant bit to 0 (in the IEEE half-precision format the most significant bit expresses the sign of the number).

```
// STEP 1: Compute the Clipping coefficient "C_true" as L1 norm of the INPUT VECTOR:
{% for i in range(1, num_parameters + 1) %}
meta.mainm.x{{ i }}abs = hdr.param.x{{ i }};
meta.mainm.x{{ i }}abs[15:15] = 0b0;            // set to positive
{% endfor %}
```

**Figure 3.5:** Compute absolute values in P4

The above examples demonstrate how to customize a P4 program in an agile way at compiled time, without having to rewrite each time considerable parts of it.

## 3.4 Noise generation

The targeted DP technique requires applying noise generated from a specific distribution, typically *Gaussian/Laplace*. P4 targets like *bmv2* have the ability to generate noise in the data plane, e.g., through the use of dedicated engines for the generation of random numbers. The main drawback with these engines is that is not possible to control the distribution of the produced values, therefore we cannot produce noise values that follow a Gaussian or a Laplace distribution through those. For this reason, the noise required by the target DP mechanism must be generated differently.
We can control the noise distribution more easily by pre-computing its values offline with software running at the control plane level and then loading them into a data plane table to be used at run time. The P4 program will load the required noise values into metadata fields from a match-action table and then will add those to the clipped input vector elements. This process allows us to control the underlying noise distribution and to fulfil the differential privacy requirements. For this work, we designed a simple incremental counter mechanism to select different noise values per each parameter. A better mechanism should be adopted to update the table values after a certain time, to avoid introducing and repeating patterns into this process.

## 3.5 Floating Point implementation

As previously explained, Floating Point (FP) operations are not natively supported by P4 targets, therefore as part of this work, we investigated possible techniques to perform floating point arithmetic on P4-programmable targets. Table 3.1 summarizes the state-of-the-art techniques for performing FP arithmetic on P4-programmable targets:

| **Technique** | **Description** | **Pros** | **Cons** |
|---|---|---|---|
| *NetFC [20]* | It uses the IEEE half-precision format to represent FP values and a "Look-Up Table" (LUT) method to implement FP operations. It focuses on FP addition, FP multiplication and FP division. | - LUT approach amenable to match-action based architectures.<br>- Tables can be used across the various FP operations.<br>- Implementation of all of the required FP operations.<br>- Pseudo-code available. | - LUTs cannot represent every possible value, the final result will be approximated.<br>- Several tables per operation are required.<br>- No public code is available for each operation. |
| *InREC [21]* | It uses the IEEE half-precision format to represent FP values and the LUT approach to implement FP operations. They implement the following elementary operations: $\log(x)$, $2^x$, $x/y$, $\sin x$, $\sqrt{x}$, $x+y$. | - LUTs provide fast matching.<br>- The technique is tested on a real Tofino switch. | - No pseudo-code available.<br>- No public code available.<br>- The implementation of each elementary operation is not detailed. |

| **Technique** | **Description** | **Pros** | **Cons** |
|---|---|---|---|
| *Unlocking the Power of Inline Floating-Point Operations [22]* | It implements standard FP arithmetic for IEEE-754 numbers. They propose "FPISA" to implement FP computations in P4. "FPISA" breaks each floating point value into *exponent* and *signed mantissa* and stores them separately in different pipeline stages. | - They proposed a variant technique called "FPISA-A" to run on existing Tofino switches. | - The full technique cannot be implemented on PISA-like switches.<br>- Not all the FP operations are illustrated.<br>- No public source code available. |
| *In-Network Fractional Calculations [23]* | It is based on Fixed-point encoding of real numbers. It uses Taylor polynomials to approximate a function and the degree of the polynomial to specify accuracy. Taylor coefficients are pre-computed offline and stored in the P4 program. | - It shows how to encode complex functions like $\pi$, cos, log, exp.<br>- The code is publicly available.<br>- The accuracy of the operations is tuneable by modifying the degree of the polynomial. | - Not amenable for an implementation on a PISA-like switch target. |

**Table 3.1:** State-of-the-art techniques to implement FP arithmetic in P4

As reported in Table 3.1, none of the related work made available an open-source code of the P4 implementation of their technique that we could leverage for our work. In the end, we decided to implement the technique proposed in the *NetFC* paper for the following main reasons:

- Their technique can be applied to perform all of the FP operations we need for our DP technique, namely, addition, multiplication and division;

- They provide pseudo-code for the floating point operations;
- Their technique shows good approximation results on average.

### 3.5.1 The original *NetFC* algorithms for FP arithmetic

*NetFC* implements FP arithmetic operations using a "Look-Up Table" (LUT) method. This method translates a FP operation into a sequence of table lookups. Being match-action tables a construct available on P4-programmable target switches, this method fits well with that target architecture.

**FP MULTIPLICATION / FP DIVISION:**

Let us focus the description of the algorithm on the multiplication operation since similar steps are followed to perform division. Assuming $x$ and $y$ to be two non-zero IEEE half-precision numbers, the multiplication can be expressed as:

$$x * y = \pm 2^{i+j} \tag{3.5}$$

where $i = \log_2(|x|)$ and $j = \log_2(|y|)$. This new computation can be performed through the two following tables:

1. *"log2 T"*:
   $$\text{VALUE} = \lfloor \log_2(|\text{KEY}|) \rfloor$$
   where KEY is a 16-bit FP number and VALUE is a 16-bit Integer. This table is used to retrieve the values of $i$ and $j$;
2. *"exp2 T"*:
   $$\text{VALUE} = 2^{\text{KEY}}$$
   where KEY is a 16-bit Integer computed as $(i + j)$ and the VALUE is a 16-bit floating point number.

The entire pseudo-code for the NetFC's FP multiplication is shown in Algorithm 3.1. *NetFC* first parses an input packet to obtain the two operands $x$, $y$ and checks if they are equal to 0. Afterwards, it looks up the *"log2 T"* to find out the value of $i$ and $j$, whose sum is used to detect corner cases. After that, $(i + j)$ is used as KEY to look up on the *"exp2 T"* and retrieve the final

**Algorithm 3.1** *NetFC* floating-point multiplication

```
Input: $p$, an input data packet.
 1: parser floating-point number $x, y$ from $p$
 2: if $x \equiv 0$ or $y \equiv 0$
 3:     return $0$
 4: end if
 5: get $i = \lfloor \log(|x|) \rfloor, j = \lfloor \log(|y|) \rfloor$ by "logTable"
 6: $\text{sign}(x * y) = \text{XOR}(\text{sign}(x), \text{sign}(y))$
 7: $n = i + j$
 8: if $n > 15$                                  {Corner case: overflow}
 9:     return $\infty$
10: else if $n < -24$                            {Corner case: underflow}
11:     return $0$
12: else
13:     get $|x * y| = 2^n$ by "expTable"
14:     set the sign bit
15: end if
```

result of the FP multiplication. The sign of the result is determined by XOR-ing the two input signs.

We have illustrated only the algorithm for the FP multiplication, but that can be easily adapted to perform the FP division operation. The division algorithm is based on a set of transformations, similar to the ones applied for the multiplication, resulting in the following equation:

$$x/y = \pm 2^{i-j} \tag{3.6}$$

**FP ADDITION:**

The addition operation is performed through a different algorithm that requires the use of more look-up tables. Having two positive IEEE half-precision FP numbers $x$ and $y$, the addition can be rewritten as:

$$x + y = 2^{i+\log_2(1+2^{j-i})} \tag{3.7}$$

where still $i = \log_2(|x|)$ and $j = \log_2(|y|)$. We define the following table:

1. *"miT"*:

$$\text{VALUE} = \lfloor \log_2(1 + 2^{\text{KEY}}) \rfloor$$

where the KEY=$(j - i)$ and both KEY & VALUE are 16-bit Integers.

The above addition equation only holds for positive operands, while it becomes the following if negative operands are considered:

$$x + y = \pm 2^{i + \log_2(\pm 1 + \pm 2^{j-i})} \tag{3.8}$$

Two more definitions of the *"miT"* table are necessary to perform this operation also considering negative operands:

1. *"mi2T"*:
$$\text{VALUE} = \left\lfloor \log_2(1 - 2^{\text{KEY}}) \right\rfloor$$
2. *"mi3T"*:
$$\text{VALUE} = \left\lfloor \log_2(-1 + 2^{\text{KEY}}) \right\rfloor$$

where the KEY is always given by $(j - i)$ and both KEY & VALUE are 16-bit Integers.

**Algorithm 3.2** *NetFC* floating-point addition

**Input:** $p$, an input data packet.

```
 1: parse floating-point operands $x, y$ from $p$.
 2: if $x$ (or $y$) $\equiv 0$
 3:     return $y$ (or $x$)
 4: end if
 5: get $i = \lfloor \log_2(|x|) \rfloor, j = \lfloor \log_2(|y|) \rfloor$ by "logTable".
 6: $n = j - i$.
 7: if $n > 15$                                   {Corner case: overflow}
 8:     return $y$
 9: else if $n < -15$                             {Corner case: underflow}
10:     return $x$
11: else
12:     select "miTable" based on Table 3.2.
13:     get $m = \lfloor \log(|1 \pm 2^n|) \rfloor$ by "miTable".
14:     $k = i + m$.
15:     get $|x + y| = 2^k$ by "expTable".
16:     set sign bit according to Table 3.2.
17: end if
```

The *NetFC* algorithm to perform FP additions in P4 requires 5 different table lookups. The pseudo-code is reported in Algorithm 3.2. It should be noted that by considering both positive and negative operands, subtraction can be seen as a special case of the same algorithm. The process starts with parsing an incoming packet to obtain the two operands $x, y$ and check if they are 0. Then, it retrieves the values $i$ and $j$ by looking up the *"log2T"* table. After processing corner cases (lines 7-10), it selects which *"miTable"* to look up based on a decision table (Table 3.2) to obtain the value $m$. The value $m$ is summed to the $i$ value to obtain the KEY of the *"exp2T"*. The final result is obtained by looking up the *"exp2T"* table, while the sign of the result is determined by the decision table Table 3.2.

| $x > 0$ | $y > 0$ | $\|x\| > \|y\|$ | **Formula** | **"miTable"** | **Sign** |
|---|---|---|---|---|---|
| T | T | T | $2^{i+\log(1+2^{j-i})}$ | *miT* | + |
| T | T | F | $2^{i+\log(1+2^{j-i})}$ | *miT* | + |
| T | F | T | $2^{i+\log(1-2^{j-i})}$ | *mi2T* | + |
| T | F | F | $-2^{i+\log(-1+2^{j-i})}$ | *mi3T* | - |
| F | T | T | $-2^{i+\log(1-2^{j-i})}$ | *mi2T* | - |
| F | T | F | $2^{i+\log(-1+2^{j-i})}$ | *mi3T* | + |
| F | F | T | $-2^{i+\log(1+2^{j-i})}$ | *miT* | - |
| F | F | F | $-2^{i+\log(1+2^{j-i})}$ | *miT* | - |

**Table 3.2:** *NetFC* decision table

The decision table is needed to process both positive and negative floating point values. The more general FP addition equation (3.8) covers eight possible situations which are decided by the following conditions: 1) $x > 0$; 2) $y > 0$; 3) $|x| > |y|$. All situations are enumerated in such a decision table. The last two columns in the table show respectively which *"miTable"* to apply and the sign of the final result.

```
// Custom "Control block" that performs the FP MULTIPLICATION & FP DIVISION:
//      - if the "div" bit is set to 1  =>  execute FP DIVISION code
//      - if the "div" bit is set to 0  =>  execute FP MULTIPLICATION code
control FPdiv_mult(inout metadata meta, in bit<1> div) {
    ...
}

// Custom "Control block" that performs the FP ADDITION:
control FPaddition(inout metadata meta) {
    ...
}
```

**Figure 3.6:** Control blocks declaration in P4

### 3.5.2 Our P4 implementation of *NetFC*

Since the P4 implementation of the NetFC algorithms for FP arithmetic is not publicly available, our implementation of those algorithms follows the description and the pseudo-code available in the original paper [20]. This section describes our P4 program for performing FP arithmetic, based on the NetFC algorithms.
Our P4 program implements FP addition, FP division and FP multiplication using custom control blocks declared as in Figure 3.6. A "control block" is a P4 language construct that provides a structured way to define the control flow and packet processing logic, enabling network administrators to specify the behaviour of P4 switches in a flexible way. Through look-up tables and custom logic, it allows the manipulation of packet header and metadata fields. Typically, a control block declares tables and actions, then in the internal *apply()* block it specifies a custom logic that uses the previously defined tables to perform the required operations. The definition of these control blocks is included in a dedicated file called **FPoperations.p4**. FP operations are implemented on a separate file to keep the main P4 program more readable and organized. The program is structured as follows:

1. It declares a *metadata* structure with a set of fields required to store intermediate values for each operation. "Metadata" refers to the additional information associated with the packet that is not part of the packet header. It is typically used to store intermediate results, context information, or any other data that needs to be shared or manipulated during packet processing. In our program, each field is carefully created with a specific size and it is used to store values computed during internal operations;

2. It creates a custom *Control* for FP multiplication and FP division. The same control can be used for both operations as their implementation codes are similar. In fact, we use a 1 bit variable called "div" as a control argument to select between multiplication and division. It allows us to execute different parts of codes according to the selected operation. It is illustrated in line 4 of Figure 3.6.
   The purpose of this control is to define tables, actions and logic to perform both operations according to Algorithm 3.1. The defined tables are the *"log2 T"* and the *"exp2 T"*. As an example, the Figure 3.7 illustrates the *"log2 T"* table declaration where the KEY is a FP number stored in *"meta.tempmeta.x"* and the corresponding VALUE is provided by the controller and it will be store in the *"meta.tempmeta.i"* field through the "geti" action. The matching type used for this table is called "ternary" and it allows matching on a value with three possible states for each bit: 0, 1, or "do not care" (often represented as *). We configure the ternary match such that the last four bits of our FP number are "do not care" bits. This is done on purpose to avoid the situation in which we do not have a match due to some internal rounding. It is important to highlight that a "ternary"

```
//                              "log2T" Table
action geti(int<16> val) {
    meta.tempmeta.i = val;
}
table log2T {
    key = {
        meta.tempmeta.x: ternary;
    }
    actions = {
        geti;
        NoAction;
    }
}
```

**Figure 3.7:** *log2* table declaration

match is more expensive in terms of resources compared to an "exact" match.
Then, in the *apply()* block, we manually configured the following corner cases: $0/0 = Nan; x/0 = +inf; x/-0 = -inf; 0/x = 0; 0*0 = x*0 = 0*x = 0$ and we implement the logic of Algorithm 3.1.

3. It creates a custom *Control* for FP addition as shown in line 9 of Figure 3.6. As the addition operation requires more look-up tables and a different logic, it is implemented in a separate control. The control's structure is the same as in point 2, it declares tables, actions and the logic of Algorithm 3.2. In this case, three more tables are required, therefore the defined tables are: *"log2T"*, *"miT"*, *"mi2T"*, *"mi3T"*, *"exp2T"*. The declaration of each table follows the structure of Figure 3.7. Then, in the *apply()* section of this control we first consider the following corner cases: $0 + 0 = 0; 0 + y = y; x + 0 = x; (-x) + x = 0; x + (-x) = 0$ and then we implement the logic of Algorithm 3.2 using previous tables.
   The choice of the correct *"miTable"* is based on a decision table (Table 3.2) and it is implemented in P4 with a sequence of *if-else* statements.

To perform a single FP operation, we create an instance of our custom FP control block inside a main program called **main.p4**.
As an example, in Figure 3.8 we show how to perform an addition. We instantiate the control block for addition in line 6, then, on the *apply()* block we use that instance to perform the addition operation between $x1$ and $x2$ (lines 9-13).
For any of the implemented FP operations to work, the P4 tables in our program must be filled with the required entries. For that purpose, our implementation also includes some controller logic inside the program **mycontroller.py** that initialises the P4-target's tables with the required

```
/***     INGRESS PROCESSING       ***/
control MyIngress(inout headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_meta) {
    // Create one instance of Addition's custom Control block:
    FPaddition() add1;

    apply {
            // ADDITION between "hdr.param.x1" and "hdr.param.x2"
            meta.tempmeta.x = hdr.param.x1;
            meta.tempmeta.y = hdr.param.x2;
            add1.apply(meta);
            hdr.param.add_res = meta.tempmeta.finresult;
            ...
```

**Figure 3.8:** Perform a FP addition in P4

values. Therefore, the role of the controller is to first create the required tables and, then, load them into the switch. More precisely, to implement FP division and multiplication our program defines and uses the following two LUTs:

1. *"log2T"*:
$$\text{VALUE} = \lfloor \log_2(|\text{KEY}| * k) \rfloor$$
2. *"exp2T"*:
$$\text{VALUE} = 2^{\frac{KEY}{k}}$$

While for the FP addition, our program defines and uses the following three tables:

1. *"miT"*:
$$\text{VALUE} = \left\lfloor \log_2(1 + 2^{\frac{KEY}{k}}) * k \right\rfloor$$
2. *"mi2T"*:
$$\text{VALUE} = \left\lfloor \log_2(1 - 2^{\frac{KEY}{k}}) * k \right\rfloor$$
3. *"mi3T"*:
$$\text{VALUE} = \left\lfloor \log_2(-1 + 2^{\frac{KEY}{k}}) * k \right\rfloor$$

Each table includes a scaling factor $'k'$, which is not strictly mandatory, however, it allows to increase the accuracy of the final result by increasing the impact of the decimal fraction of $log2(|x|)$, otherwise reduced by the floor function. The NetFC paper [20] shows that as the scaling factor increases so does the accuracy of their results. For example, they show in the paper they can reach very high accuracy values (99%) by using a scaling factor k=1024.

To build tables, the network administrator defines which KEYs to use for each table and, through the execution of the five above equations, the correspondent VALUEs are generated. Once the controller creates the tables, it is ready to load them into the P4 switch. To do that, a connection between the controller and the switch is created using specific APIs, as shown in line 2 of Figure 3.9. At this point, specific commands can be used to populate each table.
For example, Figure 3.9 shows the controller code to populate the *"log2T"*.

```
1  // Connect to the P4 switch:
2  controller = SimpleSwitchThriftAPI(9090)
3
4  // Ternary mask:
5  ternary_mask = '0xFFF0'
6
7  // Populate the "log2" Table:
8  //                   KEYS are stored in "key_l_hex" as HEX_16_bits format
9  //                   VALUES are stored in "values_l_hex" as HEX_16_bits format
10 for i in range(len(values_l_hex)):
11     key = key_l_hex[i]
12     value = values_l_hex[i]
13     // add one entry
14     controller.table_add('log2T', 'geti', [key+'&&&'+ternary_mask], [value])
```

**Figure 3.9:** Populate the *log2* table

The "table_add" command is used to add entries into a match-action table of a P4 switch and it requires the following arguments:

- Name of the table to which you want to add an entry (e.g. "log2T");
- Name of the action to perform when a matching entry is found (e.g. "geti");
- Values to match against in order to determine whether an entry applies (as we use ternary match the key is combined with a ternary mask). In our example, we use line 5 of Figure 3.9 to set the last four bits as "do not care bits";
- Parameters to pass to the action when it is invoked (e.g. value).

The code in Figure 3.9 assumes to have the set of KEYs and VALUEs stored into *"key_l_hex[]"* and *"values_l_hex[]"* respectively in a hexadecimal format, as it is required by the switch's "table_add" command. Then, through the for loop, it adds one entry at a time to the *"log2T"*.

The same switch's command is used to populate the other tables.

In summary, up to a maximum of five lookup tables are created to perform a single FP operation between two operands into our P4 program.
The next section explains how our program can be used to execute multiple operations in parallel as well as to obtain the sequence of operations required to execute the targeted DP mechanism.

## 3.6 Sequential FP operations: DP in P4

Once we were able to execute the single FP operations in a P4 program, we moved to the implementation of the required sequence of operations to achieve Differential Privacy (DP). The exact process is described in Section 3.1. The DP technique requires performing a sequence of FP operations in a predefined order.
The core P4 program that implements our DP mechanism is **main.p4**, while the individual FP operations are implemented in the separate **FPoperation.p4** program described in Section 3.5.2. More in detail, **main.p4** does the following main steps:

1. The *parameters* of the vector to operate on are passed to the switch through a custom header. We assume FP input values following the IEEE half-precision format. The Section 3.3 illustrates how vectors are stored in P4;

2. For each FP operation between two numbers, our core program includes a correspondent control block instance. It is important to understand that each instance creates a separate set of tables with distinct names. Therefore, when we create multiple instances we are essentially creating multiple sets of tables, each associated with its specific instance. The total number of FP operations (*TOT_FP_OPERATIONS*) required can be computed with the following equation (assuming $n$ is the number of elements of the input vector):
$$TOT\_FP\_OPERATIONS = (n * 2 - 1) + n + 1 \tag{3.9}$$
$$(n * 2 - 1) \Rightarrow \text{N° of } \textit{FP additions}$$
$$n \Rightarrow \text{N° of } \textit{FP multiplications}$$
$$1 \Rightarrow \text{N° of } \textit{FP division}$$
Since the predefined sequence of operations is dependent on the number of input elements, a P4 template with placeholders is used to create the final P4 program following the technique described in Section 3.3;

3. Loading the value of the clipping constant from a dedicated table into some metadata field, this value is configured by our controller program and provided by the network administrator.

```
//----------------------    "store_C_constant" Table   ------------------------//
action storeConst(bit<16> const_val) {
    meta.mainm.c_const = const_val;
}
table Clip_constT {
    key = {
        standard_meta.ingress_port: exact;
    }
    actions = {
        storeConst;
        NoAction;
    }
}
```

**Figure 3.10:** Clipping constant table declaration

The code snippet in Figure 3.10 presents the relative table declaration in P4. We create a table named *"Clip_constT"* and we use as KEY the port on which the packet enters the switch. Once a packet enters the switch there will be a match, therefore, the action named *"storeConst"* is executed and it stores the value provided by the controller in the metadata *"meta.mainm.c_const"*.

4. Executing the required operations described in Section 3.1, in sequence, to apply differential privacy. Namely, in the *apply()* block of the Ingress control of our main p4 program the following operations are executed:

   (a) **Clipping the input vector**:

      i. Compute absolute values for all elements of the input vector. The Figure 3.5 shows the relative P4 code to compute absolute values for each input parameter;

      ii. Add the absolute values in parallel, with a sequence of partial FP addition operations, finally obtaining $\overline{C_{true}} = ||x||_1 = \sum_{i=1}^{n} |x_i|$. The set of P4 operations is carefully generated to exploit the internal "parallelism". In P4, certain operations can be parallelized as long as there is no hard dependency between them. In our case, by carefully designing the sequence of additions, some of them can be executed in parallel. As an example, let us consider the P4 code in Figure 3.11 to perform the sum among four elements. The *"add1"* (between $x1 - x2$) and the *"add2"* (between $x3 - x4$) are not dependent, so they are executed in parallel, while *"add3"* depends on partial results. Then, partial results are added to obtain the final *C_true* value (lines 13 - 18).

```
1  // Parallel sums of absolute values:
2  // x1 + x2
3  meta.tempmeta.x = meta.mainm.x1abs;
4  meta.tempmeta.y = meta.mainm.x2abs;
5  add1.apply(meta);
6  meta.mainm.temp1 = meta.tempmeta.finresult;
7  // x3 + x4
8  meta.tempmeta.x = meta.mainm.x3abs;
9  meta.tempmeta.y = meta.mainm.x4abs;
10 add2.apply(meta);
11 meta.mainm.temp2 = meta.tempmeta.finresult;
12 // temp1 + temp2
13 meta.tempmeta.x = meta.mainm.temp1;
14 meta.tempmeta.y = meta.mainm.temp2;
15 add3.apply(meta);
16 meta.mainm.temp3 = meta.tempmeta.finresult;
17 // final result
18 meta.mainm.ctrue = meta.mainm.temp3;
```

**Figure 3.11:** Parallel sums of absolute values

The parallelism speeds up the computation, since, multiple operations can be executed simultaneously;

iii. Get the value of the $C$ constant from a metadata field and find the value of $C/C_{true}$ by performing a single FP division;

iv. Find the value of $C_{clip} = min(1, C/C_{true})$ with an *if* statement;

v. Execute $n$ FP multiplications to obtain $\vec{x}_{clipped} = \vec{x} * C_{clip}$;

(b) **Adding noise to the Clipped vector**:

i. Compute the noise vector ($\vec{n}$) offline in a python program, based on *Laplace* distribution;

ii. Perform $n$ FP additions to obtain the final noise output $\vec{y} = \vec{n} + \vec{x}_{clipped}$.

Since the P4 language does not support loops, each vector operation has been unrolled in our program.

5. The final vector $\vec{y}$ is emitted into a custom header in the output packet.

The functionality of **mycontroller.py** is very similar to the one described for the single FP operations. It still creates the five different look-up tables, in the same way, required to perform

the FP operations. However, since the number of tables varies with the size of the input vector, the controller program must populate the respective number of tables.
Let us consider the code snippet in Figure 3.12 where our **main.p4** executes two FP additions. Multiple FP operations can be executed by simply instantiating the corresponding control block multiple times (lines 6-7). Then, the first addition between $x1 - x2$ is performed in lines 10-14 and the second addition is shown in lines 15-19. As described in Section 3.5.2, our FP addition requires 5 different look-up tables. It follows that, in this example, the controller needs to populate the following ten tables: *"add1.log2T"*, *"add1.miT"*, *"add1.mi2T"*, *"add1.mi3T"*, *"add1.exp2T"*, *"add2.log2T"*, *"add2.miT"*, *"add2.mi2T"*, *"add2.mi3T"*, *"add2.exp2T"*.

```
1  /***     INGRESS PROCESSING       ***/
2  control MyIngress(inout headers hdr,
3                  inout metadata meta,
4                  inout standard_metadata_t standard_meta) {
5      // Create two instances of Addition's custom Control:
6      FPaddition() add1;
7      FPaddition() add2;
8  
9      apply {
10             // ADDITION between "hdr.param.x1" and "hdr.param.x2"
11             meta.tempmeta.x = hdr.param.x1;
12             meta.tempmeta.y = hdr.param.x2;
13             add1.apply(meta);
14             hdr.param.add1_res = meta.tempmeta.finresult;
15             // ADDITION between "hdr.param.x3" and "hdr.param.x4"
16             meta.tempmeta.x = hdr.param.x3;
17             meta.tempmeta.y = hdr.param.x4;
18             add2.apply(meta);
19             hdr.param.add2_res = meta.tempmeta.finresult;
20             ...
```

**Figure 3.12:** Sequence of FP additions in P4

# 4 Evaluation

This chapter presents the evaluation of our P4 implementation of the floating point arithmetic and of the targeted differential privacy mechanism. To begin with, it outlines the general network model used for our experiments, in Section 4.1. Afterwards, it describes the experiments performed for each floating point operation reporting accuracy results and presenting an analysis of the resources required by the related P4 programs, in Section 4.2. Finally, it presents the evaluation of the targeted differential privacy mechanism in Section 4.3.

## 4.1 Experimental set-up

The experiments of the following sections are based on the SDN model shown in Figure 3.1 which emulates a simple network scenario to develop and test a P4 program. For our experiments, the scenario was emulated on an Ubuntu virtual machine using the **P4 UTILS** tool [24] that allows to create and test virtual networks composed of P4-programmable switches. P4 UTILS is based on the following main components:

1. *Mininet*: It is a network emulation framework that can efficiently virtualize nodes (hosts and switches) in a network by exploiting Linux kernel features. It is an interesting tool to develop and test Software-Defined Networks (SDN) including P4-programmable switches [25];

2. *Behavioral-model*: It is a software switch for rapid prototyping and testing of P4 programs. The software is the P4 reference software switch commonly referred to as the

Behavioral model or *bmv2*. This switch provides an environment where developers can build, debug and test P4 programs without the need for specialized hardware [26];

To build the SDN model in Figure 3.1 for our evaluation, we have written a Python program called **network.py** that, based on P4 UTILS APIs, specifies the following network configuration:

- A P4-programmable switch and a program to be loaded into it;
- Client and server hosts;
- Links by placing hosts in the same sub-network;
- The SDN controller program;

First, **network.py** creates and configures the desired network. Then, it compiles and loads the switch's program. After the switch is started, it runs the SDN controller program to load tables on the P4 switch and it ends by opening a *Mininet* command line (CLI). By interacting with CLI, host terminals can be opened to execute custom scripts.

## 4.2 On the Floating Point Operations

This section describes the testing environment and presents the evaluation results for our P4 implementation of the floating point arithmetic techniques described in Section 3.5.2. It focuses on FP addition, FP division and FP multiplication.

### 4.2.1 Set-up

We use the **network.py** program, described in Section 4.1, to create the base network. The FP operations are defined in the P4 program called **FPoperations.p4** (as described in Section 3.5.2). The controller program (**mycontroller.py**) is the SDN controller logic responsible to populate the match-action tables on the switch. In order to perform a single FP operation, we use a general P4 program called **main.p4** (in Section 3.5.2 a code snippet shows how a FP operation is expressed in P4) responsible for instantiating the relative control block and performing the selected FP operation. To test the P4 program with our FP operation implementation, we use two Python scripts, one to simulate a client application and another one to simulate a server application:

- *"client.py"*: Python program that creates and sends to the switch an arbitrary number of custom network packets. Packets are built with the "Scapy" library [27] and have the following custom data (Figure 4.1):

    1. $X$: It is an IEEE half-precision floating point number that represents the first operand of a given FP operation;
    2. $Y$: It is an IEEE half-precision floating point number that represents the second operand of a given FP operation;
    3. *RES*: It is the output of $X\ op\ Y$ in P4, where *op* can be addition / multiplication / division. It is initialized at zero and it will be written the switch with the result of the operation.

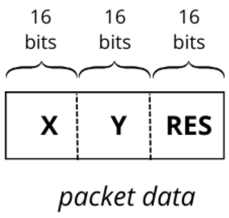


**Figure 4.1:** Packet format of test packets for single FP operations

- *"server.py"*: Python program, also based on "Scapy", used to sniff packets coming from the P4 switch and store the values of the operands and of the result of the operation in packets for further analysis.
  By taking input operands $X$ and $Y$ from the received packet, the server program computes the same FP operation in Python to obtain the *exact result*. This is the target value we use to assess the accuracy of our output result in P4.
  Following the evaluation in the *NetFC* paper [20], we use the same formula to compute the *accuracy* metric. Assuming $EXACT$ to hold the result of the operation computed in Python and $APPROX$ to hold the result of the operation computed in P4, we compute our *accuracy* metric as follows:

$$EXACT = Python(X\,op\,Y)$$

$$APPROX = P4(X\,op\,Y)$$

$$accuracy = e^{-\frac{|EXACT-APPROX|}{|EXACT|}} \tag{4.1}$$

  Accuracy values are in $[0, 1]$, where a higher accuracy indicates that the $EXACT$ value is closer to the $APPROX$ value. It is easy to see that an *accuracy* value of 1 is an optimal

result since it means that the two measurements are the same.
To store the above values, the server creates a data frame with the following five columns and saves it as a CSV file:

1. **X**: first input operand;
2. **Y**: second input operand;
3. **P4 result**: output result computed in P4;
4. **Python result**: output result computed in Python;
5. **Accuracy**: value computed using equation 4.1.

Then, we compute the average accuracy reported in our plots as a single value obtained by averaging the values in the *accuracy* column.

### 4.2.2 Experiments

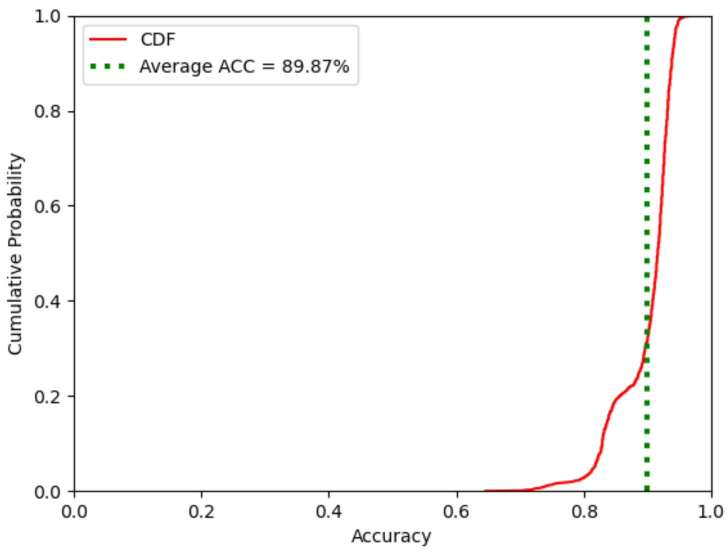
The input data for our experiments is taken from a single dataset of ten thousand IEEE half-precision floating point numbers in the range $[-2, +2]$, generated following a uniform distribution. The range was arbitrarily chosen to include both negative and positive numbers. To build a complete list of value pairs:

1. We first select 1000 elements from the dataset to be used as the first operand ($X$) in the packet;
2. We then select another 1000 elements from the dataset to be used as a second operand ($Y$) in the packet.

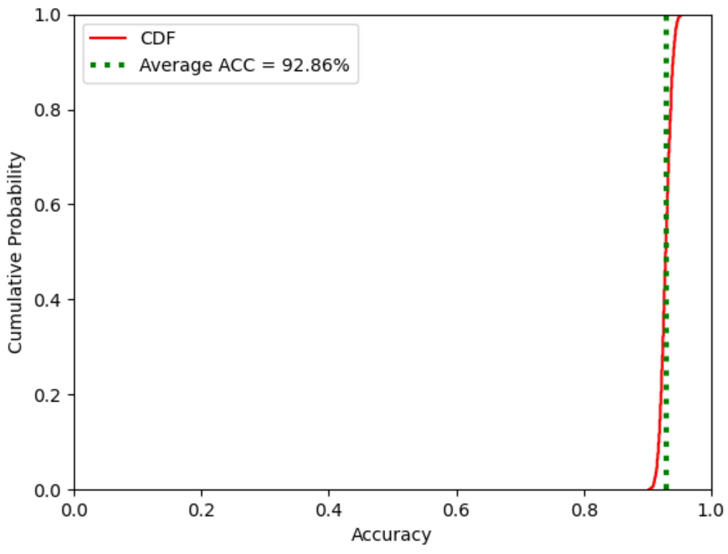
The **single experiment** consists of 1000 FP operations. For each FP operation, we repeat the single experiment ten times, each time repeating the operations in the above points 1 and 2 to feed different input data to each experiment.

### 4.2.3 Results

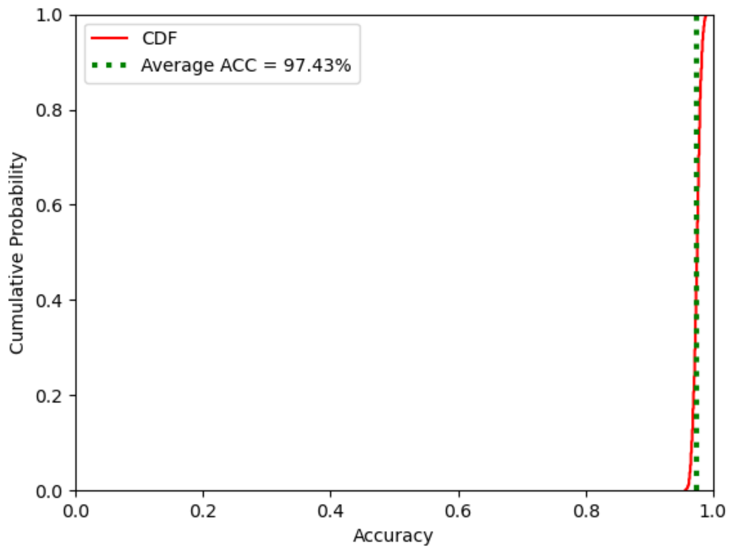
For every FP operation under analysis, we run ten experiments with random data and we store the results into a CSV file (the format of the CSV file was described in Section 4.2.1). In the end, we run a total of 10000 computations per each FP operation and so we obtain a CSV file with the same number of entries.



**(a)** Cumulative Distribution Function of the FP addition



**(b)** Cumulative Distribution Function of the FP multiplication



**(c)** Cumulative Distribution Function of the FP division

**Figure 4.2:** Accuracy's plots

The data stored in such files are used to create the **accuracy plots** reported in Figure 4.2. These plots show the "average" Cumulative Distribution Function (CDF) for the accuracy values. The "average" means we are plotting the CDF of 1000 accuracy values, each value obtained by averaging the corresponding results of the ten different experiments for that FP operation. Figure 4.2 shows the results of each considered FP operation in three different plots. Figure 4.2 (**a**) plots the CDF of the FP addition, Figure 4.2 (**b**) plots the CDF of the FP multiplication and Figure 4.2 (**c**) plots the CDF of the FP division. The X-axis shows *accuracy* values in the range $[0, 1]$ and the Y-axis represents the cumulative probability. The vertical dotted line represents the average accuracy among all $x$ data points.
By comparing the average accuracy obtained with each operation, we can see that the division achieves the highest accuracy values ($97\%$) after the multiplication ($92\%$), while the addition has the lowest average accuracy ($90\%$). Moreover, in the case of the addition the measured accuracy values are spread out over a wider range compared to the other two operations.
It can also be observed a certain similarity of the CDFs of the division and multiplication. The technique to implement those two operations is indeed very similar, contrary to the algorithm used to compute the addition operation where more lookup tables and processing steps were required.
Despite our results being good overall, that is, average accuracy values equal to or greater than

90%, they still looked worse than the ones reported in the original *NetFC* paper [20]. We acknowledge that we were not able to replicate their exact same results (above 99% accuracy) and we think that this may be due to different factors in the experimental settings, e.g., different datasets, different range of values stored in the lookup tables, different scaling factors. Unfortunately, the authors of NetFC did not provide all of the necessary data and details to replicate their experiments with our implementation.

### 4.2.4 Resource Analysis

This section reports an analysis of the memory resources which are required to store the lookup tables for executing our FP arithmetic algorithms. We consider 16 bits to store both IEEE half-precision and integer numbers in P4. We can think of a P4 table as a KEY-VALUE pair, where each KEY and VALUE can be either a FP number or an integer. Therefore, we consider 32 bits the size of one row for each of the five lookup tables required. Based on that table entry size, we build Table 4.1 to show the total memory usage by each look-up table, where the table sizes are obtained by multiplying the number of rows by the row's size. The largest table is the *log2T* while the other tables have considerably lower similar sizes.

| | **log2T** | **exp2T** | **miT** | **mi2T** | **mi3T** |
|---|---|---|---|---|---|
| **TOTAL SIZE (KB)** | 80,0 | 1,47 | 1,60 | 0,8 | 0,8 |

**Table 4.1:** Single lookup table sizes

Knowing the size of each lookup table required, we can compute the total memory requirements for each FP operation as shown in Table 4.2. The last two columns show, respectively, the total number of tables required and the corresponding memory size.

| | **N° log2T** | **N° exp2T** | **N° miT** | **N° mi2T** | **N° mi3T** | **TOT N° TABLES** | **TOT SIZE (KB)** |
|---|---|---|---|---|---|---|---|
| *addition* | 2 | 1 | 2 | 2 | 2 | 9 | 167,88 |
| *mult / div* | 2 | 1 | - | - | - | 3 | 161,47 |

**Table 4.2:** Memory usage for single FP operations

In summary, a total of 9 tables and 167,88 KB of memory are needed to execute a single FP addition while 3 tables and 161,47 KB of memory are needed to execute a single FP division. It is worth noting that the single table size is based on the number of entries in each table.

In our experiments, we do not consider the entire range of representable values for each table, as we faced a table size limitation on the *bmv2* software switch. For the sake of time, we could not investigate this issue further in the context of this work and increase the number of entries that each P4 table could store on *bmv2*, rather we limited the number of table entries to 1024 in our experiments.

## 4.3 On the Differential Privacy Mechanism

This section describes the experiments and results related to the targeted differential privacy mechanism explained in Section 3.6.

### 4.3.1 Set-up

The testing environment is created using the same **network.py** program described in Section 4.1. Each FP operation is implemented using the control blocks defined in the **FPoperations.p4** file, which was described in Section 3.5.2. In this setting, the number of tables that the controller (**mycontroller.py**) needs to populate depends on FP operations to be performed. Multiple copies of the same table are required to perform multiple FP operations per packet with our P4 program.
The core P4 program that implements the DP technique is **main.p4**. It is generated starting from a P4 template with placeholders, as the pre-defined sequence of operations is modified according to the number of input parameters. The structure of the main P4 program was explained in Section 3.6. Similarly to the experiments of Section 4.2, we use the following two scripts to simulate client and server applications:

- *"client.py"*: Python program that creates an arbitrary number of packets by leveraging the "Scapy" library [27]. Each packet contains a flexible number of parameters, configurable by the administrator. To ease the generation of the corresponding P4 code in our "main.p4" program, we assumed the number of parameters to always be a power of two. As it can be seen from Figure 4.3 where the packet format is illustrated, the packets generated by *"client.py"* also carry some fields to store the noise values. The choice to carry noise values in the packets has been driven by the need to replicate more easily the computation performed with P4 in Python for comparison. This does not invalidate the more sound solution proposed in Section 3.4 to introduce noise directly on the P4 target through the P4 program.

- *"server.py"*: a Python program (based on "Scapy") used to sniff packets coming from the P4 switch and store the values of the vector for further analysis. Every packet received

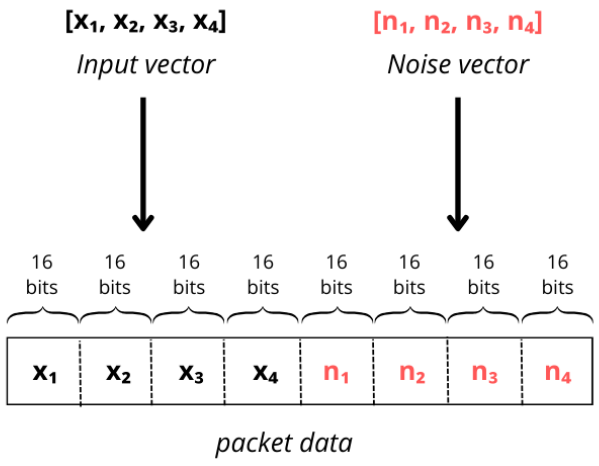


**Figure 4.3:** Packet format of test packets for the DP targeted mechanism, example with an input vector containing 4 parameters.

carries two subsequent vectors: the first vector contains the differentially private values and the second contains the noise values.
Whenever the server receives a packet, it computes the same sequence of DP operations in Python. This result is considered the "target result" and it will be used, together with the P4 result, to compute the *accuracy* metric (using the same formula as in Section 4.2).
To store both the received and computed vector values, the server creates a data frame with the following five columns and writes it as a CSV file:

1. **Input vector**: each input parameter of a given input vector;
2. **P4 result**: the result of the DP operations computed in P4;
3. **Python result**: the result of the DP operations computed in Python;
4. **Accuracy**: a value computed using equation 4.1 for each couple of P4 and Python results;
5. **Average accuracy**: a single value obtained by averaging the *accuracy* column.

### 4.3.2 Experiments

We used the same dataset described in Section 4.2.1 to run these experiments. Remember that our dataset contains ten thousand FP values uniformly distributed in the range $[-2, +2]$. We use the following approach:

1. For each experiment, we randomly select $n$ values from the dataset, where $n$ is the number of input parameters.

We repeat step 1 for 1000 times, which is equivalent to pick 1000 input vectors of size $n$ from our dataset. An analogous process is executed for every $n \in [2, 4, 8, 16, 32, 64]$.

### 4.3.3 Results

Each CSV file created by the server contains an *accuracy* column that holds values in $[0, 1]$. This measures the discrepancy between the output of our DP mechanism applied with the approximations introduced by P4 and the output of the equivalent operations performed in Python. Higher *accuracy* values mean that our P4 code approximates well the targeted FP operations. The optimal result corresponds to an accuracy value of 1.

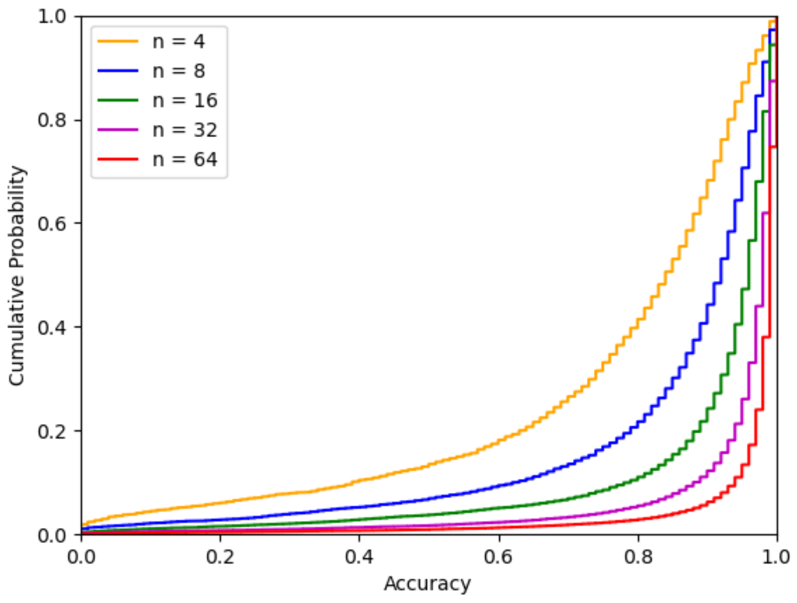


**Figure 4.4:** Impact of different input vector size $n$ on the accuracy of the DP mechanism

Surprisingly, from our experiments in Figure 4.4, we see that an increasing number of parameters produces better accuracy results on average. For the sake of time, we could not investigate deeply these results within this work, nonetheless, we are aware of the following factors that may have contributed to these results: the choice of a specific clipping constant and our dataset for experiments.

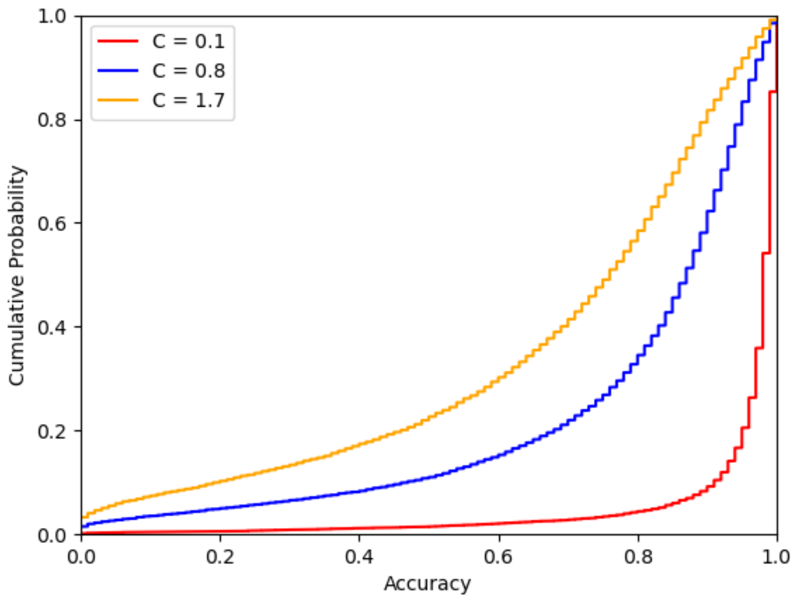


**Figure 4.5:** Impact of different values of the clipping constant $C$ on the accuracy of the DP mechanism

With regards to the clipping constant in our DP mechanism, we have run some experiments to understand how this may have affected our accuracy results. The clipping constant is a free hyperparameter used during the "clipping" phase of the differential privacy mechanism. To show the impact of the $C$ constant on our results, Figure 4.5 plots the Cumulative Distribution Function (CDF) of the accuracy in three different experiments. The three experiments use the same set-up and the same number of input parameters but they vary the $C$ value used for clipping the input vector parameters. We consider the three following arbitrary values following for the clipping constant: $[0.1, 0.8, 1.7]$. Each curve in Figure 4.5 represents the accuracy CDF for each of the three experiments. From these experiments, it looks that lower values of $C$ (e.g. $C = 0.1$) produce better accuracy while increasing the value of the clipping constant (e.g. $C = 1.7$) produces less accurate results. Figure 4.5 demonstrates the importance of the $C$ value in our differential privacy mechanism. Therefore, the clipping constant $C$ represents a parameter to be carefully tuned according to the ML model and the dataset used.

### 4.3.4 Resource Analysis

This section analyzes the memory usage of our P4 program when running the target differential privacy technique. Differently from Section 4.2 where single Floating Point (FP) operations were tested, the differential privacy mechanism requires performing a varying number of FP operations, based on the number of input parameters $n$. The total number of FP operations can be computed with equation 3.9 in Section 3.6. Based on that equation, Table 4.3 shows how the memory size increases with increasing the number of parameters and, as a consequence, the number of required FP operations. For each row the table reports the number of input parameters, the correspondent number of FP operations (computed with equation 3.9) and the total memory required, the last can be computed referring to the data in Table 4.2.

| N° Input parameters | TOT FP operations | TOT memory size (KB) |
|---|---|---|
| 2 | 6 | 123,99 |
| 4 | 12 | 254,39 |
| 8 | 24 | 515,19 |
| 16 | 48 | 1036,79 |
| 32 | 96 | 2079,99 |
| 64 | 192 | 4166,39 |

**Table 4.3:** Memory usage of our P4 program varying the size of the input vector

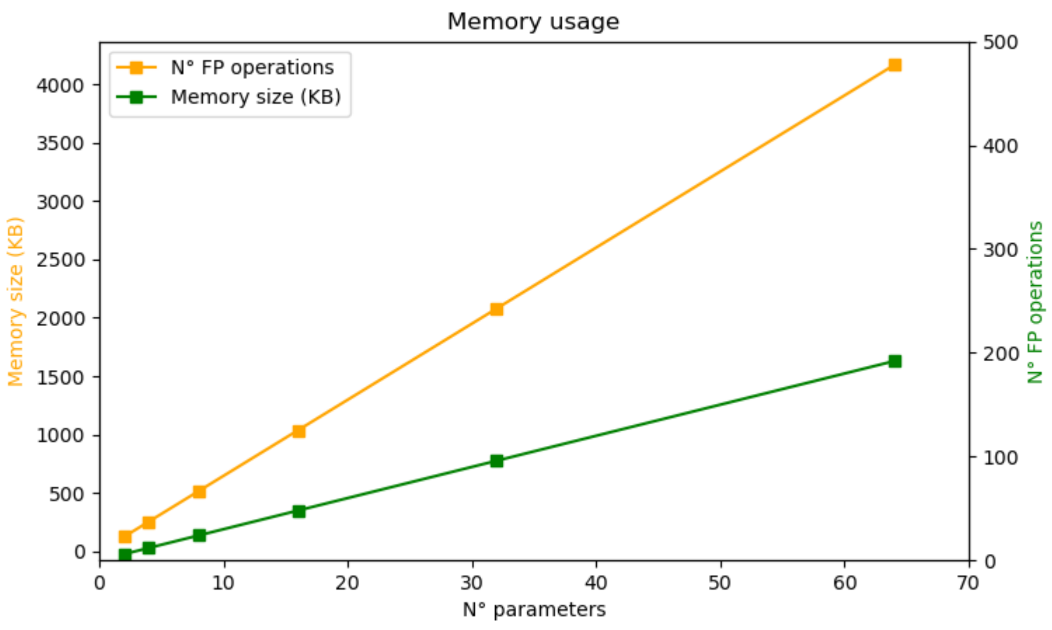


**Figure 4.6:** Switch's memory usage varying the number of input parameters

Figure 4.6 shows the linear increase of the memory required to store our P4 program when increasing the number of parameters to deal with into the input vector. 64 was the highest number of parameters we could successfully compile in our P4 program during our tests. Despite compiling successfully, that program size was already not easily managed by the software switch *bmv2*, so we could not run any experiment with that program. To better consider these results, remember that a P4-programmable switch (e.g., Intel's Tofino [28]) usually has a few tens of MB of memory available, which is also shared by several switch functions.

Another metric that shows the increasing complexity of our P4 program with an increasing number of input parameters in the input vector is the size of the P4 program itself.

| **N° Input parameters** | **Size of the P4 program** | **Size of the compiled program** |
|---|---|---|
| 2 | 381 | 14236 |
| 4 | 427 | 29795 |
| 8 | 519 | 61023 |
| 16 | 703 | 123479 |
| 32 | 1071 | 248391 |
| 64 | 1807 | 498215 |

**Table 4.4:** Sizes of the P4 source code and of the relative compiled program by varying the number of input parameters

Table 4.4 compares the sizes of the generated P4 program and its relative compiled JSON file. The first column of Table 4.4 shows the number of input parameters, the second one indicates the lines of code (LoC) of the generated P4 program and the third one presents the LoC of the compiled JSON file. The JSON file is the compiled program generated by the P4 compiler. The LoC for the JSON file is significantly higher compared to the LoC of the P4 program as in the compiled program, each "control" is very likely codified independently, therefore, the size of the compiled program continues to increase linearly as reported in Figure 4.7.

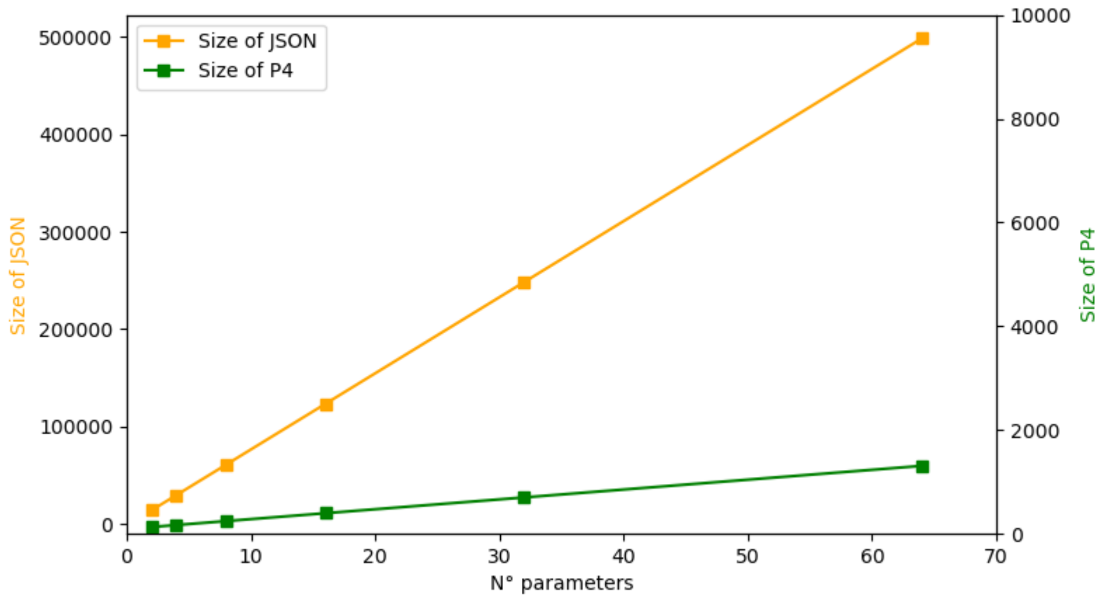


**Figure 4.7:** Memory usage of our P4 program varying the size of the input vector

# 5
# Conclusion

This work focused on the implementation in the network data plane of the targeted privacy-preserving technique. We tried to leverage the capabilities of programmable switches and the P4 language to implement a Differential Privacy (DP) technique in the switch's data plane. The targeted DP technique requires performing a sequence of Floating Point (FP) operations in a predefined order. Namely, our DP technique is based on two main steps: *clipping* and *adding noise*. We implemented a P4 program for the software switch *bmv2* that performs exactly those operations, assuming a vector of floating point numbers in input.

One of the main challenges we tackled was to implement floating point operations on the switch's target, as the P4 language does not support FP data types and arithmetic. In the literature, we could find several works proposing techniques to implement FP operations in P4, however, those do not make their source code publicly available. Therefore, after a careful analysis of those techniques, we embarked on the implementation of a specific technique, *NetFC*, only based on the description available in the original published work [20]. By following that information, we were able to implement FP addition, FP multiplication and FP division in P4.

To evaluate our targeted differential privacy mechanisms, we compared the differentially-private values produced by our P4 program with the values produced with an equivalent Python program. Based on the outputs of these two programs, we compute an accuracy value that shows how the two measures differ. With our implementation of that technique, we obtained accuracy values between 90% and 97%, while the original *NetFC* work reported higher accuracy results (99%). We believe that such discrepancy can be attributed to various factors in our ex-

perimental setting like the input dataset, the entries in the look-up tables and the scaling factor used. Unfortunately, the authors of *NetFC* did not provide enough details about their experimental setup for us to replicate precisely their experiments.
Finally, we analysed the impact of varying the size of the input vector on the size of our P4 program and the required memory. And we have also measured the impact of the clipping constant on the overall accuracy of our DP technique.

## 5.1 Future work

We hereby list a set of avenues that could be worth exploring with future work on this topic.

- In our experiments with the software switch *bmv2*, we hit a maximum number of entries per table, after 1024 entries, *bmv2* considered a table "full". Larger tables would allow to run experiments using wider ranges of numbers, possibly leading to higher accuracy results.

- Increase the scaling factor "*k*" of the *NetFC* technique to further improve *accuracy*, as shown in original paper. By increasing "*k*", more table entries are needed and, consequently, this increases the memory requirement of the program too.

- Our work targeted the *bmv2* software switch for rapid prototyping and testing. However, it is definitely important to assess better the practical relevance of such a P4 program to target other platforms (like a hardware switch). This may require changing considerable portions of our code as real switches can have different constraints compared to the *bmv2* software switch we used.

# References

[1] H. N. Zaoxing Liu, "Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches," *USENIX*, 2021.

[2] N. Z. Zhaoqi Xiong, "Do switches dream of machine learning? toward in-network classification," *ACM*, 2019.

[3] S. K. Changhun Jung, "A scalable and dynamic acl system for in-network defense," *ACM*, 2022.

[4] R. B. Oliver Michel, "Theprogrammabledataplane: Abstractions, architectures, algorithms, and applications," *ACM*, 2021.

[5] P. C. O. Vachuska and Davie, *Software-Defined Networks: A Systems Approach*. Systems Approach LLC, 2022.

[6] D. K. F. M. V. R. P. E. Veríssimo, "Software-defined networking: A comprehensive survey," *IEEE*, 2015.

[7] P. B. D. D. G. Gibb, "P4: Programming protocol-independent packet processors," *ACM*, 2014.

[8] https://p4.org/.

[9] C. D. Mihai Budiu, "The p4_16 programming language," *ACM*, 2017.

[10] A. W. Kobbi Nissim, Thomas Steinke, "Differential privacy: A primer for a non-technical audience," *Harvard University*, 2017.

[11] Dwork, "Differential privacy," *Springer*, 2006.

[12] J. U. Thomas Steinke, "Between pure and approximate differential privacy," *Cornell university*, 2015.

[13] A. R. Cynthia Dwork, *The Algorithmic Foundations of Differential Privacy*. the essence of knowledge - now, 2014.

[14] P. M. Mammen, "Federated learning: Opportunities and challenges," *arXiv*, 2021.

[15] D. P. Anshuman Suri, Pallika Kanani, "Subject membership inference attacks in federated learning," *arXiv*, 2023.

[16] G. N. Oualid Zari, Chuan Xu, "Efficient passive membership inference attack in federated learning," *arXiv*, 2021.

[17] J. L. Kang Wei, "Federated learning with differential privacy: Algorithms and performance analysis," *Cornell university*, 2019.

[18] X. C. Xinwei Zhang, "Understanding clipping for federated learning: Convergence and client-level differential privacy," *ARXIV*, 2021.

[19] https://jinja.palletsprojects.com/en/3.1.x/.

[20] H. P. Penglai Cui, "Netfc: Enabling accurate floating-point arithmetic on programmable switches," *ICNP*, 2021.

[21] K. L. Matthews Jose, "Inrec: In-network real number computation," *IEEE*, 2021.

[22] O. A. Yifan Yuan, "Unlocking the power of inline floating-point operations on programmable switches," *USENIX*, 2022.

[23] R. A. Shivam Patel, "In-network fractional calculations using p4 for scientific computing workloads," *ACM*, 2022.

[24] https://nsg-ethz.github.io/p4-utils/index.html#.

[25] https://mininet.org/.

[26] https://github.com/p4lang/behavioral-model.

[27] https://scapy.net/.

[28] https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html.

# Acknowledgments

These past four years of university have been characterized by numerous challenges and they have been far from easy. I started my master's degree with the COVID-19 pandemic, which profoundly altered the landscape of education by the shift to online learning. The transition to remote learning presented its own set of obstacles, depriving me of the opportunity to engage in traditional face-to-face interactions with classmates and professors.
Despite my initial intent to conduct a thesis project abroad in a traditional setting, circumstances led me to pursue this project almost entirely remotely. Throughout the duration of this thesis, I had the privilege of collaborating with talented individuals and gaining insights into the research methodologies employed within a well-known Spanish company. I am grateful for the opportunity to work towards the publication of a scientific paper.
Regrettably, especially the final phase of this project has posed significant challenges to my peacefulness, stemming from various factors including the nature of my chosen topic and its complexity. I would like to extend my appreciation to my supervisors for their guidance and support throughout this seemingly endless process. Nevertheless, this experience has led to unexpected consequences, and, I think that in conjunction with the outcomes of academic endeavours, it is paramount that serenity remains a guiding principle.
I extend my gratitude to my unwavering perseverance, which has been fundamental in ensuring that I did not falter on such a crucial step in my career. Additionally, I am deeply indebted to my family for their support, particularly my mother and my "dad", as well as my girlfriend for her constant presence by my side. Furthermore, I am thankful to all individuals who have played a role in supporting me in achieving this milestone.