UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Towards Privacy-Preserving Network Verification of Inter-Domain Software-Defined Networking

Dissertation presented by
Arnaud DETHISE

for obtaining the Master's degree in
Computer Science and Engineering

Supervisor(s)
Marco CANINI

Reader(s)
Olivier BONAVENTURE, Ramin SADRE, Marco CHIESA

Academic year 2016-2017

Acknowledgments

I would like to thank Marco Chiesa for his feedback and advice in the redaction of this thesis as well as his help with the techniques used in this work,

Marco Canini for his support of the research leading to this thesis and during my stay at KAUST,

Daniel Demmler for his work on ABY,

and David Yeh, Ruiqi Qiu and Doreena Chen from the Visiting Student Research Program office for the opportunity to travel and work at KAUST.

Contents

List of abbreviations and definitions

Abbreviations

AS	Autonomous System
BGP	Border Gateway Protocol
DIB	Deflection Information Base
GMW	SMPC protocol proposed by Goldreich, Micali and Wigderson
IP	Internet Protocol
IXP	Internet eXchange Point
M-XOR	Masked XOR
OT	Oblivious Transfer
RIB	Routing Information Base
SDN	Software-Defined Networking
SDX	Software-Defined Internet eXchange Point
SGX	Software Guard Extensions
SMPC	Secure Multi-Party Computation
TEE	Trusted Execution Environment
TPP	Trusted Third Party
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Definitions

- **Deflection**: A router that sends a packet through an outgoing interface that is not consistent with the next-hop of the best BGP route.
- **Share**: Encrypted data in SMPC.
- **IXP member**: Autonomous System that is customer of an IXP operator and able to exchange traffic with other members of this IXP. Because all members are ASes, and, within an IXP, all ASes are members, those terms are equivalent in the context of this thesis.
- **Match rule**: In an SDN rule of the form $match \rightarrow action$, we refer to the *match* part as a *match rule*.
- **Distinct rules**: SDN match rules whose set of commonly-matched packets is empty.
- **Candidate rule**: The rule held by the *client* party. See Chapter 3.
- **Installed rules**: The rules held by the *server* party. See Chapter 3.
- **Online time**: Time required to perform computation operations that depend on the inputs.
- **Setup time**: Time required to perform computation operations that are independent from the inputs (such a generating multiplication triples) and that can be pre-computed.

List of Figures

List of Tables

Part I

Introduction

The approach to network configuration referred as Software-Defined Networking (SDN) has gained popularity in the recent years thanks to the higher flexibility it provides to network operators. SDN opens the path for easier configuration, leading to fine-grained routing, better security and increased scalability.

Originally implemented for data center networks, academia and industry are now trying to bring the benefits of SDN to inter-domain routing. The SDN approaches promise better traffic engineering through load balancing and application-specific peering, enhanced security, and a higher level of automation by using simple interfaces.

Yet, naïve deployment of SDN on the Internet is dangerous as the expressiveness of BGP for configuring the network is significantly more limited than the expressiveness of SDN, which allows fine-grained rules to deflect traffic away from BGP's default routes. Most notably, this mismatch may lead to incorrect forwarding behaviors such as forwarding loops and blackholes, ultimately hindering SDN deployment at the inter-domain level.

In this thesis, we make a first step towards verifying the correctness of inter-domain forwarding state for a network running both BGP and SDN. Our focus is ensuring inter-domain forwarding loop freedom while keeping the SDN rules private, as said rules comprise confidential routing information and operators are reluctant to share them.

An important class of entities in inter-domain routing are Internet eXchange Points. They distribute traffic between large numbers of Autonomous Systems, which operate physical networks. Given their central role in interconnecting many networks and their importance in bringing popular content closer to the users, Internet eXchange Points are a compelling place to focus our work and provide solutions that benefit many network operators at once. In particular, Software-Defined Internet eXchanges (SDX) bring SDN capabilities to their customers networks without requiring said customers to invest in an SDN-capable infrastructure.

In an SDX, members install *outbound* SDN rules to select the peer they send traffic to. Sending traffic through a route different from the best BGP route causes a *deflection* of the traffic from its intended route. Multiple such deflections may cause the creation of persistent forwarding loops whenever the deflections occur in the same path and traffic is forwarded in a loop.

In order to prevent this behavior, we devised a simple yet powerful primitive that gives operators the possibility to verify whether two rules overlap, i.e. whether the set of packets both rules affect is non-empty, while keeping the actual rules private.

We propose an efficient implementation of this primitive by using recent advancements in Secure Multi-Party Computation and we then leverage it as the main building block for designing a system that detects Internet-wide forwarding loops among any set of SDN-enabled Internet eXchange Points. The implementation is publicly available as a Git repository at `https://bitbucket.org/adethise/disjoint-flow-match`.

This thesis is organized as follows: first, we provide the necessary background on the basic concepts needed to understand this thesis in Chapter 1, as well as existing solutions for related problems in Chapter 2.

After that, we will discuss the core of the problem, i.e. what objectives are we trying to achieve in Chapter 3, followed by a description of how we designed our privacy-preserving technique, alongside an implementation and extensions in Chapter 4.

Finally, we see how to build applications with the constraints imposed by SMPC (Chapter 5) and how to optimize them while accounting for the reduced quantity of information available (Chapter 6). We then present experimental results in Chapter 7 to show that our approach is feasible, and finally discuss possible future works in Chapter 8 before concluding.

Part II

Content

Chapter 1

Background

In this chapter, we provide a background on the main relevant concepts and technologies used in the Internet as a support for understanding the contributions of this thesis.

We start with a description of Internet routing and BGP, the classical and *de facto* standard routing protocol. This will serve as a reminder of how routes are computed and shared between operators, as well as establishing some definitions used in this document.

After that description of the classical approach to inter-domain routing, we follow up with a short introduction to the Software-Defined Networking (SDN) approach, its properties and a broad description of the OpenFlow architecture, a popular implementation of SDN.

We will then move on to the description of what Internet Exchange Points (IXP) are, and why they are important and play a growing role in today's Internet. In particular, we describe how IXPs can provide SDN capabilities to Autonomous Systems through the development of Software-Defined Internet Exchange Points (SDXes).

After that, we will briefly explain the problem of network verification, presenting several important properties that are sought by network operators. We discuss some questions that need to be answered to verify the consistency of the configuration in a network such as the absence of forwarding loops.

Finally, we will give some background about the Secure Multi-Party Computation model. This includes the problems it aims to solve and the guarantees SMPC provides. As SMPC is the fundamental privacy-preserving building block upon which we build our inter-domain network verification primitive, we also go over two protocols for secure computation.

1.1 Internet routing: the Border Gateway Protocol

The Border Gateway Protocol (BGP) [40] is used to configure and exchange the routes between Internet addresses. It is a *control plane* protocol (i.e. it is one of the elements used to compute and configure the content of the forwarding tables on routers). The control plane is often compared to the *data plane*, which is the actual forwarding of traffic towards its destination.

This section quickly describes the BGP mechanism. We will focus on the information that is relevant to this thesis, so some parts of the protocol are omitted.

Internet Protocol (IP) addresses are aggregated by their prefix. Network operators exchange addresses such as 10.0.0.0/24, representing all addresses of the form 10.0.0.x.

In BGP, an Autonomous System (AS), which is a network entity operating a physical network, announces to its neighbors the prefixes for which it can forward data plane traffic. Those route announcements have two parts: the announced prefix, and the *AS path*, that is, the ordered list of ASes through which traffic will go before reaching its destination.

When an AS receives a BGP route announcement from its neighbor, it stores that route in a local Routing Information Base (RIB). The operator then sorts the routes for each prefix by

order of preferences and select the best one. Other routes are not used, but they are kept in the RIB in case the best route is withdrawn by the neighbor.

Once the best route has been selected and installed in the control plane, the operator announces that route to its neighbors (after adding its own identifier, because it is now on the path).

There are two main reasons to announce the AS numbers of all the networks in the path. First, it is an indication of the efficiency of the path, because data plane traffic following a longer path takes more time to reach its destination. And second, it lets operators detect and prevent forwarding loops, because an AS receiving a path where its own number is already present will discard that route.

1.2 Software-Defined Networking

The main objective of Software-Defined Networking is to allow anyone to program the data-plane through an open standardized (vendor-independent) protocol. Routers and switches have both a control and data plane, and those planes used to be tightly coupled. SDN creates the possibly to decouple them, dividing the network in simple commodity switches that only forward packets (data plane) and a (logically centralized) controller to compute the forwarding state (control plane).

Enforcing this separation between control plane and data plane allows the network operators to easily express detailed forwarding behavior and rely on an SDN controller to ensure that the switches in the network are configured appropriately even in a dynamically changing topology.

The flexibility provided by SDN allows operators to tune the forwarding behavior depending on the type of traffic that is being exchanged and the state of the network, a technique called traffic engineering. For example, a network operator with multiple peers (i.e. other networks that accept traffic from this network) might want to use application-specific peering (selecting the destination peer based on Layer 4 header information, e.g., TCP port 80 for HTTP traffic) or load balancing (splitting the traffic into subflows that are sent to different peers).

In the following sections, we define the SDN components that make up the control plane (the policies) and the data plane (the switches), as well as the component connecting them, the SDN controller.

SDN is typically deployed through the OpenFlow protocol [35] which is the assumed implementation in this thesis. However, Software-Defined Networking is rather an approach than a technology and not strictly defined by OpenFlow.

Policies

The SDN routing policies are defined as match-action rules. The match part defines a subspace of the network traffic, called a *packet flow*. The selection of packets belonging to the flow is based on packet headers. Rules can match fields from layer 2 to layer 4, such as the source and destination MAC addresses, source and destination IP addresses, protocol (TCP, UDP...), ports, VLANs, etc.

The action part of the rule describes how the packets are processed. The possible actions include forwarding the packet to an ongoing interface, rewriting the packet header, and dropping the packet. Later in this thesis, we will focus exclusively on the more common case of forwarding actions.

An example of SDN policies is shown in Figure 1.1. Those rules match the port (HTTPS, SSH, HTTP) and IP addresses to forward traffic towards the outgoing interface of peer routers C, D and E. In this example, packets will be forwarded through the outgoing interface to C if their destination port (`dPort`) is 443 or 22, the outgoing interface to D if their destination port

is 80 and their source IP address (`sIp`) starts with 40.0.0.x, and through E if their destination IP address (`dIp`) is exactly 10.0.0.1.

```
dPort=443 -> fwd(C)
dPort=22 -> fwd(C)
dPort=80 && sIp=40/24 -> fwd(D)
dIp=10.0.0.1/32 -> fwd(E)
```

Figure 1.1: Example of SDN rules

In addition to installing fine-grained rules manually on switches, the network operator can define high-level SDN policies on abstractions rather than on the physical entities themselves [26, 36]. For example, the network operator might want to forward the packet to a remote peer through multiple switches, but will write a single policy to achieve this. A *controller* then translates the policy according to the physical topology of the network.

Controller

Various SDN controllers exist, from open ones to proprietary. Popular ones include Ryu [5], Open vSwitch [38] and POX [29].

To simplify the configuration of an SDN network, SDN controllers provide operators with an abstraction that simplifies network management. That abstraction ought to be simple and high-level, with the possibility to easily reach destinations even if their location is distant in the network covered by SDN switches.

A popular such abstraction is to see a network of switches as "one big switch" [28]. The switch is connected to multiple peers and can exchange traffic between all of them, even though the real topology has multiple hops between each peer. This abstraction allows installing policies composed of many rules over a network covering a wide area, and defers to the controller the task of installing appropriate low-level rules on each switch automatically.

For scalability and availability reasons, SDN controllers are only logically centralized while they may be implemented as distributed systems.

Switches

As mentioned above, SDN-enabled switches use the OpenFlow protocol to communicate with the controller and receive new rules to install in their forwarding table. When a packet enters the switch, its header is matched against the rules currently installed in the table. If no match is found, the packet will be sent to the controller for analysis or dropped. Optionally, a new rule can be installed on the switch to manage other packets with the same header.

Controller designs also consider the physical limitation of commodity switches. The switches, being hardware based in order to process packets at line rate, only have a limited capacity. The size of their forwarding table is bounded, and they might only be able to store a subset of all the rules active on the controller. The installation of a new rule also takes a small amount of time during which packets can not be forwarded.

Because the controller manages the limited capacity of the switches so that the traffic is not, or only marginally affected, the network operators are freed from this burden.

1.3 SDN-enabled Internet Exchange Points

Internet eXchange Points (IXPs) are high bandwidth physical networks through which hundreds of network operators are interconnected. All networks connected to the IXP are automatically

connected to each other, allowing them to exchange traffic while avoiding the latency and cost of forwarding to the traditional large IP transit providers. Essentially, this is bringing the content closer to the users.

An IXP administrator operates its physical interconnectivity fabric among the connected Autonomous Systems (ASes), which are called *IXP members* in this context, allowing each member to exchange traffic with any other member. Because of the central and growing role of IXPs in Internet connectivity, IXPs are deemed an ideal and unique opportunity to introduce new functionalities for inter-domain routing.

One such functionality has for objective to bring the flexibility of Software-Designed Networking to inter-domain routing through IXPs. By enabling SDN on the IXP network, the members can gain its benefits without deploying an SDN-capable fabric of their own.

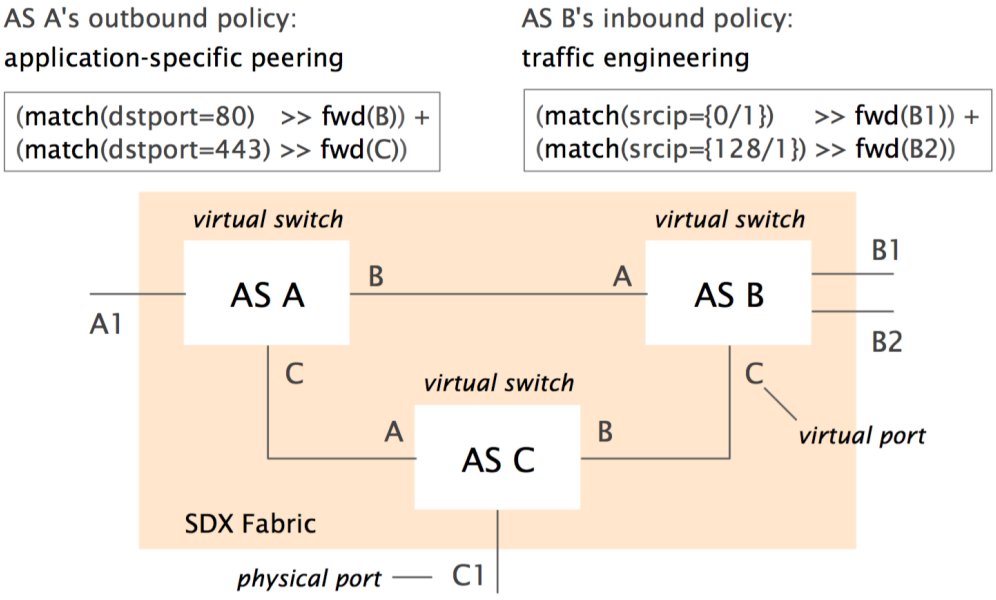


Figure 1.2: The SDX virtual switch abstraction (from [24])

Figure 1.2 illustrates the SDN virtual switch abstraction at an IXP with three members, A, B and C. In this abstraction, each member has a single virtual switch, which is connected to the virtual switch of every other IXP member. The virtual switch also has one or several connections to the member's own network.

This abstraction allows the IXP members to simply express both *outbound policies* (to decide where to send their traffic) such as those installed by A, and *inbound policies* (to select their ingress border router) such as those installed by B.

Without SDN, the traffic at an IXP would be dictated by BGP policies. However, BGP's expressiveness, which unlike SDN is limited to per-destination routing, is not sufficient to enable application-specific peering or traffic engineering.

Practical SDX solutions have been developed [24, 23, 14] and experimented in practice [19]. We built a primitive for these SDXes that enhances the privacy for network verification.

1.4 Network verification

Network verification is a broad term that encompasses many questions network operators might want to ask about the state of the network, as well as properties that should be respected at any time.

Here are some examples of relevant questions that the network operators might want to ask in order to know how their traffic is handled:

- Are the packets for a given IP prefix reaching their destination? Is there any blackhole for that prefix?
- Are my packets looping in the network?
- What network operators are able to read my packets? Does my traffic traverse a given domain?
- What happens if there is a failure or sudden change in traffic patterns?

More generally, those questions are about the state of the control plane and the forwarding behavior it implements, with the goal of knowing how packets in the data plane will be handled.

Network verification tools are often tailored for one or more specific routing protocols. However, in order to account for the versatility of network protocols, protocol-independent semantics exist to define the network and the functions operators can verify. Anderson et al. [7] proposed such a semantic with NetKAT, *"a new network programming language that is based on a solid mathematical foundation and comes equipped with a sound and complete equational theory.* [NetKAT includes] *primitives for filtering, modifying, and transmitting packets; union and sequential composition operators; and a Kleene star operator that iterates programs."*

The network verification semantics allow developers to express the properties they want to check as mathematical functions. This enables the use of static checkers [31, 32] and automatic optimization [39].

1.5 Secure Multi-Party Computation

Secure Multi-Party Computation aims to solve the following problem: given a public function f over private inputs $x_1, x_2, ...$, the parties holding the inputs want to compute $f(x_1, x_2, ...)$ without revealing the inputs (and the outputs) to each other.

The SMPC privacy guarantees holds against the so-called "honest-but-curious" attackers, which are defined as participants who follow the protocol but try to infer as much as possible from the information they see. The guarantee is that even under the assumption that some fraction of the participants colludes, but do not deviate from the protocol, they are not able to learn any information about the inputs of the non-colluding parties, except for what can be learned from the output itself.

Natural applications of SMPC are the Millionaire's Problem, electronic voting, or extracting information from the union of private data sets such as financial data [10].

SMPC has long been known as an inefficient technique, heavy in terms of communication and computation. However, focused research on specific implementations of SMPC have made it possible to compute functions securely in practical runtimes.

In particular, one key idea to achieve practical performance is *outsourcing* the computation of the function f to an external set T of untrusted parties. In this outsourced model, the privacy model is as follows: each participant sends as shared secrets of its input to T over a set of secure channels. T compute $f(x_1, x_2, ...)$ without learning anything about the private input and sends the output to all participants. T is trusted to compute the correct function f and not reveal any information except for the output itself to any of the parties.

This thesis focuses on two-parties SMPC over boolean functions. Below is an overview of two popular techniques to solve this problem efficiently.

GMW protocol

The GMW protocol [22] has been proposed by Goldreich, Micali and Wigderson in 1987. For a given input x, each party i receives a boolean *share*, i.e. an encrypted representation of the input, $\langle x \rangle_i$ such that $\langle x \rangle_0 \oplus \langle x \rangle_1 = x$.

To execute any boolean functions over the shares, GMW uses the XOR and AND primitives. XOR can easily be computed locally by each party, but the AND operation requires precomputed multiplication triples[1] and one round of information exchange. Because of this, the efficiency of the GMW protocol mainly depends on the AND-depth of the computed function and the delay between participants.

The output of the function is reconstructed by each party sending its share of the result to the other and XOR-ing them.

Yao's garbled circuits

Yao's protocol for secure computation [43] works by having one party send an encrypted and garbled table of the function to the other party. They will then use Oblivious Transfers to securely exchange the keys allowing decryption of the result.

Like GMW, Yao's garbled circuit protocol use the XOR and AND logic gates to build the circuit leading to the correct output. However, the computations do not have to be executed sequentially and as a consequence, the AND-depth of the circuit does not impact the total runtime of each computation.

[1] Those are triples of shares such that $\langle p \rangle = \langle m \rangle \wedge \langle n \rangle$. They can be pre-computed using techniques like Oblivious Transfers (OT).

Chapter 2

State of the Art

Our goal is to perform network verification of the inter-domain forwarding state. With this in mind, we will have a look at the many past works that have contributed to developing detection mechanisms for network anomalies such as loops and blackholes. This chapter presents the state of the art solutions for network verification, explaining what objective is addressed, what solutions have been proposed, and what are the limitations of these solutions.

We start by presenting SIDR (Safe Inter-domain Deflection-based Routing), which introduced the problem of SDN-induced forwarding loops in inter-domain routing. We now discuss the solution they proposed, as well as its assumptions.

Afterwards, we will discuss a paper researching transnational detours in packet routing. This problem is related to discovering which domains the network packets traverse, either to avoid countries know for mass surveillance (the purpose of this paper) or to anticipate network failures (another application of this problem).

Network verification has been well researched, and we will have a look at VeriFlow and NetPlumber, two solutions proposed to verify network invariants. We also talk about DiCE, a solution for network verification with more practical assumptions.

Finally, we conclude this chapter with a look at the state of the art for two different privacy-preserving methods used in problems similar to ours: trusted execution environments and SMPC-based routing.

2.1 SDN-induced loop detection

In SIDR [9], Birkner et al. introduced the problem of SDN-induced persistent forwarding loops in inter-domain routing and proposed a solution against it.

Because the expressiveness of BGP is much more limited than that of SDN rules, only a single route per prefix can be exported to each neighbor. On the other hand, SDN can forward traffic based on source IP, TCP ports and more, towards any neighbor that announced a route. Sending traffic on a route different from the one announced is called a *BGP deflection*. A single deflection can never create a loop with BGP, because the path towards which packets are deflected is entirely known thanks to the BGP announcement. However, multiple deflections between multiple SDXes can create cycles. Because those cycles are caused by deflections in the data plane, they are invisible to the control plane, including BGP.

Figure 2.1 illustrates one such loop. Member A from SDX_1 (resp. M from SDX_2) expects its traffic towards Z to follow the route (A N Z) (resp. (M B Z)), represented in green, and announces that route to B (resp. N). B and N store those routes but do not announce them in BGP.

B installed outbound policy at SDX_1 so that when it receives HTTP traffic, the SDX deflects it towards the route he learned from A instead of the announced BGP route, as shown with

the red arrow in SDX_1. Similarly, SDX_2 deflects traffic towards M on behalf of its member N. This creates a persistent forwarding loop along (A B M N).

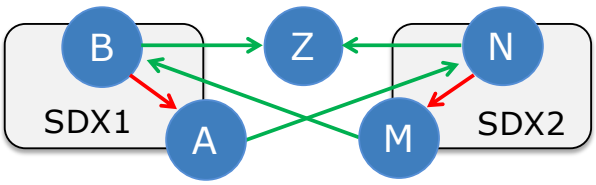


Figure 2.1: Example of SDN-induced forwarding loop along the path (B A N M)

SIDR solution

The problem of detecting forwarding loops could be solved if all ASes shared their SDN deflection rules with each other and performed verification before the installation of any new rule. However, SDN rules represent confidential business information and operators can be reluctant to share them [8, 44].

SIDR aims to mitigate that privacy concern by removing information from the exchanged rules. Concretely, for a rule of the form $match \rightarrow forward$, SIDR ignores the *match* part (considered the most private part) and only uses the per-prefix forwarding behavior. That information is shared between all ASes and tested locally before the installation of a new rule.

This achieves partial privacy, as the actions of the SDN rules are shared, at the cost of false positive results (i.e. some rules are detected as creating loops and their installation refused even though they are actually safe).

SIDR propagates knowledge of the deflections through a Deflection Information Base (DIB). The DIB stores, for each prefix, the set of ASes that deflect traffic for that prefix. When an AS M asks its SDX to install a new rule deflecting traffic for prefix p using the deflection path $C : B : A$, it must first verify that none of the ASes appearing in the AS path installed a deflection rule for the same prefix.

If the path towards which M want to send packet include an AS that already deflects some traffic for that prefix, the rule is rejected by the SDX. Otherwise, the rule is accepted and the SDX adds M to the deflection set, before propagating it to neighboring SDXes.

SDXes need to know which other SDX they need to propagate information to, and they also need a way to tell which SDXes an AS belongs to. To get this information, SIDR makes the assumption of a global SDX repository, which makes sense considering how IXP repositories already exist [3].

The problem solved by SIDR is the same as the one we are proposing a solution for, but we relax some assumptions made here. Unlike SIDR, we do not discard the *match* part of the rules and assume the number of false positives to be negligible. Furthermore, we empower the networks closer to the destination, so that allowing a rule to be installed does not reduce the capacity to install rules later.

2.2 Transnational detours

Edmundson et al. [20] observed that many countries are deploying solutions to avoid states known for employing mass surveillance of network traffic. To study the efficiency of such solutions, researchers analyzed the prominence of some countries in Internet paths, studying the possibility to avoid them fully or partially.

To detect which countries (and which networks) the packets traversed, they observed the paths from a data plane perspective rather than observing the control plane configuration. They

found that some countries are almost always present in the default paths towards the Alexa Top 100 domains.

Furthermore, they also tested the presence of the same countries using overlay network. This technique allowed them to explored path that would not be selected with a default BGP configuration. This revealed that selecting those alternative paths allowed them to reduce significantly the presence of surveillance states in the path, though it could not eliminate them completely.

In this work, we use a similar technique of discovering the path packets will follow to their destination. However, our technique differ in the information collected (the identity of ASes rather than the geographic location) and in technique (because we perform the path discovery in the control plane rather than by observing the packets in the data plane).

2.3 Network Verification

This section discusses two categories of network verification solutions. For the first category, we present *static checkers*, which verify network-wide invariants on a snapshot of the network, with VeriFlow and NetPlumber as examples. For the second category, we discuss DiCE, a solution for the verification of distributed systems which works on networks without a complete visibility.

2.3.1 Static checkers

Specialized and highly-optimized tools have been developed to verify network properties efficiently. Those tools use a mathematical model of the network and network elements (such as routers, switches and middleboxes), and verify properties expressed as mathematical functions over a snapshot of the network.

VeriFlow [32] targets specifically the case of SDN forwarding. Its goal is to check for network-wide invariants (such as the absence of loops) as the network evolves. Since the time required for an SDN controller to install a new rule is below 10 ms, real-time response requires very fast verification to ensure consistency of network state.

NetPlumber [30] is highly similar to VeriFlow, and is applicable both to SDN and conventional networks as it is designed to work in a environment with mixed protocols [31].

Both the above static checkers build a graph representing the whole network to allow for efficient, incremental verification of the network invariants. Using this technique, they are capable of achieving computation times below 1 milliseconds. However, because of the assumption of complete update visibility, which requires revealing possibly sensible business information and incurs communication delays, those techniques are likely not suited to inter-domain verification.

2.3.2 Distributed systems verification

In [12], Canini et al. propose DiCE, a solution to detect when a system deviates from its expected behavior.

DiCE explores possible system behaviors by simulation, subjecting the nodes to various inputs and observing the resulting actions. The exploration starts from the live states and runs isolated from the deployed system. This verification has been shown to be able to detect for example origin misconfiguration and BGP route leaks.

Unlike the static checkers in Section 2.3.1, this solution accounts for realistic limitations of live systems. In particular, it works without requiring the network entities to reveal the full information of their states, configuration and source code. As such, this approach is more suitable to detect misconfiguration at the inter-domain level.

However, the simulation still requires exchanging information on the behavior of nodes to the verification system. Our goal is to push the assumption of non-visibility even further, by

performing the verification over *encrypted* data, to protect the confidentiality of the rules that define the node behavior.

2.4 Privacy-preserving inter-domain routing techniques

Past studies have used privacy-preserving techniques to compute and verify inter-domain routing properties. Among the approaches yielding practical results, two main categories appear that we present next.

The first one we discuss is the use of Trusted Execution Environments, that use hardware-based proofs of trust and perform the computations in a secure environment. We will also discuss a second approach that uses Secure Multi-Party Computation to compute a function among multiple entities without any of them learning the inputs of the other entities.

2.4.1 Trusted Execution Environments

Trusted execution environments such as Intel SGX [4, 15] are a novel cryptographic solution that uses security built into the hardware as a root of trust.

Intel SGX includes a set of instructions that allow the developer to allocate encrypted regions of the memory (called enclaves) unreadable by other processes (even privileged code), and to run arbitrary code on the data in the enclave. In multi-party applications, an enclave can also verify that the remote application is also running in an enclave and authenticate the code being run on it.

Thanks to TEEs, the assumption of a trusted third party with secure communication channels becomes realistic, which enables a wide range of secure and privacy-conscious applications [27]. Thus, it should come at no surprise that this type of solution is becoming increasingly popular, even though research has found flaws related to side-channel attacks [41, 16] and the actual level of security provided by Intel SGX is debated.

Using Intel SGX, Seongmin et al. [33] looked at three different categories of network software and what effects TEE has on them. The case studies are software-defined inter-domain routing, the Tor anonymity network and secure in-network functions (i.e. middleboxes).

In the specific field of inter-domain routing, they design a system for route selection and policy verification with a TEE-enabled inter-domain controller. The controller runs its software within an enclave and receives the private policies securely, then the route selection is performed within the enclave.

Additionally, the design includes a system of policy contracts. These contracts simplify verification, making it possible to verify the correctness of the controller's output without requiring a network of trusted participants.

2.4.2 SMPC-based routing

Some papers have already preliminary experimented Secure Multi-Party Computation techniques to distribute routes and perform control plane configuration.

In [25], Gupta et al. designed an application using the Sepia platform to achieve inter-domain routing within SMPC. In [13], Chiesa et al. applied this technique to compute and distribute BGP routes within an IXP, leveraging the existence of Route Servers which compute the routes on behalf of all IXP members. By running SMPC on these servers, they provide stronger privacy guarantees, making the use of Route Servers more appealing to network operators.

On a broader, Internet-wide scale, Asharov et al. in [8] adapted this technique to work at large scale. However, this solution requires the additional assumption of a centralized BGP to function, which is very different from the current structure of the BGP system.

A common element in those techniques is leveraging the possibility to outsource SMPC computation to achieve practical runtimes. By design, it is possible with SMPC to rely on two

non-colluding third parties to perform the computation. These external parties receive encrypted inputs from all the parties involved and perform the computation, then return the encrypted output, without being able to learn any information on either the input nor the output. Once the participants have received the output from both of the computational parties, they can reconstruct the result with any information being leaked.

An important benefit of this technique is to be able to perform computations on behalf of many participants while guaranteeing privacy without requiring each participant to play an active role in the computation. However, their purpose is only the distribution of BGP routes, and they do not solve network verification when SDN is used.

Chapter 3

Design and privacy goals

Our objective is to develop a solution that would allow operators to perform verification in a network where Software-Defined Networking is enabled, at the inter-domain level, with guarantees of privacy preservation. To maximize the reach of our research, we want to focus on a development that would empower many operators at once without huge investments.

Because of their important position as traffic exchange points, we developed a solution targeted at Internet eXchange Points. Indeed, approaches that can be used at IXPs have a positive impact on each of their members as well with reduced costs. Currently, there are over 500 IXPs around the world [3], with the largest ones counting over 700 members.

In the next section, we describe the design of a Software-Defined Internet Exchange Point (SDX), that is, an IXP providing SDN capabilities to its members. This design defines some simple assumptions about the relation between members and the IXP operator, while remaining free of implementation details.

We also discuss the threat model and the behavior of actors, along with justifications for our assumptions. Finally, we discuss what kind of privacy is expected for the ASes holding SDN policies.

When discussing practical implementation and asymmetric verification, we will use the model of a *client* participant holding a single *candidate rule* and making a request to a *server* participant holding a set of *installed rules*.

3.1 Verification goals

The goal of network verification is to detect any type of incorrect routing behavior. As we discussed in section 1.4, mathematical representations of the network allow operators to verify any function that can be defined over network components.

In this thesis, we aim to solve precise, known problems hindering the deployment of SDN for inter-domain routing. Known possible network problems are the detection of forwarding loops (in which packets never reach their destination because they are indefinitely forwarded between the same routers and switches), the detection of blackholes (where packets are dropped because there is no destination configured on one of the routers or switches they traverse) and the detection of prefix hijacking (when a network operator announces a prefix for which he has no authorization).

Among the previous examples, blackholes can not be created by deploying SDN on a BGP network if the SDN rules are congruent with BGP (i.e. a SDN rule does not forward traffic for a given prefix towards a neighbor that did not announce that prefix). Similarly, prefix hijacking can be avoided by verifying the RPKI [42] for the routes a network announces.

However, as described in section 2.1, forwarding loops caused by SDN rules deflecting packets from the default BGP route are a concrete problem that needs to be solved to securely deploy SDX solutions. Thus, our work solves the problem of avoiding forwarding loops caused by SDN policies installed at IXPs.

3.2 Target environment

We expect the SDN-enabled IXPs (SDXes) to follow the general design described in [24] and illustrated in Figure 3.1. In this design, the SDX uses a (logically centralized) SDN controller running an abstraction such as the one represented in Figure 1.2. Members send their policy configuration to the controller, which then installs the appropriate rules within a fabric of SDN-capable switches.

We do not make any assumptions on how the controller compiles, translates and installs the rules. In particular, we operate on the high-level rules expressing member's policies, rather than the low-level rules installed on the routers. This allow us to remain compatible with different SDX implementations, such as iSDX [23].

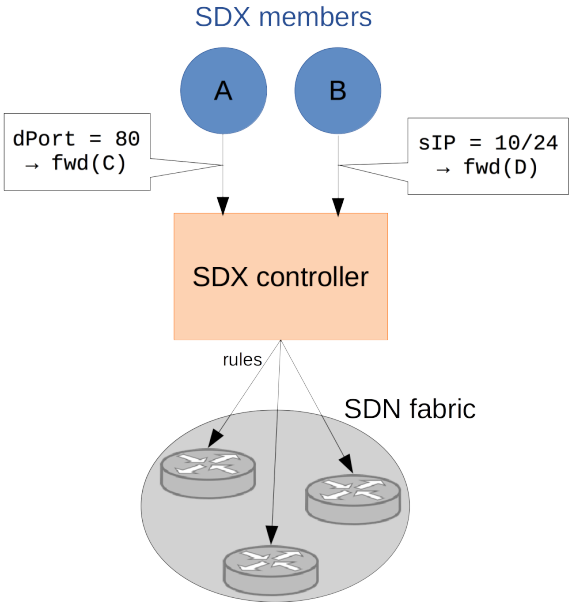


Figure 3.1: Simplified SDX architecture

With this design, the members do not need to be able to manage packets with SDN by themselves. Each member simply runs a client to communicate its policies to the controller, and the realization of those policies on the IXP switches does not need the intervention of the members. This means that, while they are not able to manage SDN by themselves, the members are still in control of the SDN policies installed on their behalf by the SDX.

The route information exchanges between members are assumed to be done using BGP, the *de facto* standard inter-domain routing protocol.

An important assumption is that all SDN rules will be congruent with BGP, i.e. if a rule is installed by member A to deflect traffic matching prefix p towards member C, then we assume that C indeed announced a route for prefix p to A. Usually, this is verified by the SDX before installing a rule.

Our goal is to verify the network consistency and the absence of forwarding loops in the traffic exchanged between SDXes. Approaches such as [32, 30], which are able verify the consistency of whole networks, do not scale to the size of an Internet-wide control plane. Our approach is different, and we will rather verify a few paths each time. The realization of this approach is more practical, because we avoid concentrating a lot of encrypted information in one point and we do not need to run a large distributed computation over thousands of entities.

Unlike [8], we do not make the assumption of having two or some non-colluding entities run the computation on behalf of everyone else. Instead, we involve all SDXes to participate in the operations that concern them, which is made possible by only verifying a few paths at once.

3.3 Threat model

In our model, we assume that each participant (both ASes and SDXes) is a possible *"honest-but-curious adversary"*, also called *semi-honest adversaries* [11]. In this model, the adversary respects the protocol but try to infer as much as possible about the policies of the other parties.

The private information to be protected is the set of rules an AS has installed or wants to install at the SDX. In regard to that information, every other entity is considered a potential attacker. This includes all other ASes and SDXes.

We believe that the honest-but-curious model makes sense for the type of application we are creating because participants identities are public and "cheating" (i.e. willingly deviating from the protocol) would cause a loss of trust from the other operators and the AS members.

We do not assume that an attacker is necessarily fully honest when choosing its inputs (both considering the candidate and installed rules), so we will discuss and address the privacy considerations in the scenario where there is no restriction on a member's inputs.

Our solution only aims to preserve privacy in the control plane. We do not try to prevent attacks relying on observations of data plane traffic to reverse engineer the rules.

Should the honest-but-curious model no longer make sense, it is possible to run SMPC using techniques that are robust even in the presence of active malicious adversaries [37], which would allow a participant to detect cheating, albeit at the cost of higher overhead and delays.

Our design also exploits outsourcing. To do so, we need to make a non-collusion assumption. Because IXP members are customers of their service provider, we design our solution around the assumption that SDXes do not collude with each other to gain information over their own members. In other words, SDXes try to learn their members policies, but not do so at the cost of colluding with another SDX.

3.4 Privacy

We want to ensure privacy of the SDN policies installed by each member. We do not aim at hiding information that is currently public in practice, such as BGP route announcements.

We preserve the privacy of SDN routing policies because they are often seen as private [25, 45] and network operators are reluctant to share them. Unlike BGP routes, those rules are fine-grained and reveal significantly more confidential information on the business strategy of operators. Additionally, policies may be used to infer the weakest points in the network and conduct denial-of-service attacks.

Unlike SIDR [9], we consider both the match part (which defines the flow) and the action part (which defines the behavior) of a rule to be private whenever possible. As such, we strive not to leak any of this information unless one of the following conditions is met:

- It is inherently required for the network operator (AS) to reveal that information to the other party.
- It can be *easily* inferred from observing the data plane as shown in [20]. We do not consider information that can be inferred with more advanced techniques such as [21] to be public.
- The operator voluntarily chose to reveal that information to facilitate the configuration of the network. This is discussed in sections 5.2.1 and 6.1.

The privacy concerns applies to each AS for its own rules. We assume that a member's rules are private in regard to everyone, including the SDX that installs the SDN rules. This might seem an excessive precaution, because the SDX needs to receive the policies from the members to configure the data plane. However, by including this guarantee in our design, we support possible future SDX designs that would allow members to obfuscate the traffic they send through

the SDX fabric. It also prevents the SDX from seeing policies that are refused, because they won't be sent to the SDX, being impossible to implement with creating network inconsistencies.

Chapter 4

Primitive

A critical property of a network with consistent control plane configuration is to be loop-free. Birkner et al. [9] introduced the problem of persistent forwarding loops caused by SDN deflections from the BGP route, and the problem is succinctly described in Section 2.1. Being unable to detect forwarding loops makes the adoption of SDN impossible.

SIDR strikes a trade-off between accuracy and privacy by ignoring the fine-grained nature of SDN rules. Precisely, it completely ignores the *match* part of the SDN rules, instead announcing the traffic this rule applies to through its BGP prefix. For each prefix on which a deflection can occur, the AS propagates its forwarding action.

However, ignoring the match part of the rule can cause false positives. We argue that it should be possible for operators to achieve privacy without sacrificing the accuracy.

To perform loop detection, we take the approach of a source-based exploration of the path a packet might take. This is, in fact, broader than simple loop detection and as discussed in Section 5.2.5, the path exploration approach can be used to perform more powerful task than a loop detection such as avoiding untrustworthy domains.

We designed and implemented the Distinct-Match primitive to fulfill that task in a privacy-preserving way. We also analyzed more advanced functions built using the primitive. Those functions are also implemented in SMPC and they provide more complex results or stronger privacy than using only the primitive itself.

4.1 Distinct-match primitive

Packet flows, i.e. the set of packets that are matched by a given rule, are to SDN what IP prefixes are to BGP. It is the unit that defines what subset of the entire network traffic is affected by a given behavior.

In BGP, when a packet destination matches the appropriate IP prefix, it is forwarded on the configured route. In SDN, when a packet matches a rule, the action linked to this rule (forward, drop, rewrite...) is applied.

It is important to note that IP prefixes can be represented as packet flows, using rules matching only on the destination IP address field.

In BGP, the path a packet follows is entirely known through the BGP route announcement. Thus, unless a deflection occurs, i.e. an SDN rule sends traffic towards a route different from the one announced, it is not necessary to perform any new verification (as information that is public in BGP is also considered public in our model and the loop-freedom guarantees from BGP applies).

Because of this, the essential question that is needed to detect a deflection is the following: *"Will any AS in the BGP path cause a deflection for packets belonging to a given subflow?"*

To answer that question, we designed the Distinct-Match primitive. Its exact formulation is as follows:

Given two *match* rules[1], is the set of packets they both match empty?

In the specific case where the primitive is used to compare a local rule to the rules installed by another participant, the primitive indirectly answers the broader question *"Will the other participant use SDN-specific operations on packets belonging to that flow?"*

If using the primitive with only the rules causing a deflection, ignoring drops and rewrites, this is equivalent to asking *"Will the other participant cause a deflection for packets in this flow?"*

We say that two rules are *distinct* if the set of packet they both match is empty.

4.1.1 Format of the inputs

The match part of an OpenFlow rule is composed of multiple *match fields*. The list of possible fields (in OpenFlow version 1.0.0) is presented in table 4.1. However, since we are applying SDN to inter-domain routing, it is assumed that the high-level policies sent to the controller do not match on local information (e.g. Ingress port, Ethernet or VLANs).

Ingress Port
Ether src
Ether dst
Ether type
VLAN id
VLAN priority CoS
IP src
IP dst
IP Proto
IP ToS bits
TCP/UDP src port
TCP/UDP dst port

Table 4.1: Match fields of a flow table entry in OpenFlow 1.0.0

The match fields can hold a concrete value (such as an IP prefix or port number) or a wildcard value to match anything. A packet matches the rule if it matches every field.

We encode this information as a block of bits of fixed size L. For each of the matched fields, the rule contains a string of bits that can have the values 0, 1 or x, where x represents a wildcard bit. For every bit b_i of a packet header, the corresponding bit in a match rule is called r_i. The rule matches a packet if for all i, $(b_i = r_i) \vee (r_i = x)$.

The input to our SMPC primitive is, for each rule r, a bit pattern p and a bit mask m. The i-th bit of the mask is 1 if r_i is different from x, else 0. The i-th bit of the pattern has the same value as r_i, unless r_i is x in which case it can be any value.

4.1.2 Circuit

Using the above input format, the primitive be expressed as a mathematical function. Given two rules r_1, r_2 composed of L bits, the primitive is:

$$\neg(\forall i \in \{0..L-1\} : (r_{1i} = r_{2i}) \vee (r_{1i} = x) \vee (r_{2i} = x)) \tag{4.1}$$

Two rules r_1 and r_2 are distinct if there is at least one bit that both are watching ($r_{1i} \neq x \wedge r_{2i} \neq x$) and for which the expected values are different ($r_{1i} \neq r_{2i}$). In other words, using our

[1] In an SDN rule of the form *match* $\rightarrow$ *action*, we refer to the *match* part as a *match rule*.

encoding, two rules are distinct if there is at least one bit $i \in \{0..L\}$ such that:

$$(p_{1i} \oplus p_{2i}) \wedge m_{1i} \wedge m_{2i} \neq 0 \tag{4.2}$$

This is illustrated in Figure 4.1. The bits in red follow the above equation and ensure that no packet matches both of them.

```
R1   11x001xxx100x10…
R2   100011xxx01110…
```

Figure 4.1: Comparison of two match rules. The bits highlighted in red are distinct and ensure that there is no overlap between the flows.

The SMPC code of our application uses the ABY framework [17] which implements both GMW and Yao protocols using a common interface of boolean circuit. ABY provides functions to easily write boolean circuits where *shares* (encrypted data) are connected through logical gates. An overview of how to write an application in ABY is given in appendix A.

The main function that needs solving for the primitive is referred to as Masked XOR (or M-XOR) and corresponds to equation 4.2. The corresponding circuit is shown in Figure 4.2.

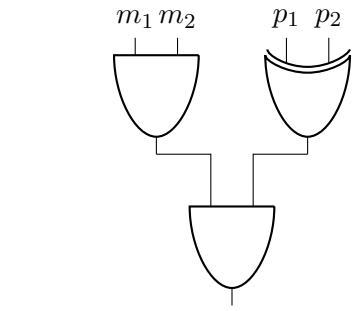


Figure 4.2: Circuit for the Masked XOR function

This circuit can be executed in parallel for each bit of the rules.

Figures 4.3a and 4.3b show the output of the circuit. In 4.3a, the bits that were highlighted in red on Figure 4.1 now return 1 as output. On the other hand, on Figure 4.3b, each rule is watching a different field and the rules are not distinct, so every bit in the output is 0.

```
R1   11x001xxx100x10…
R2   100011xxx01110…
C    0100100000010000…
```

(a) Distinct rules

```
R1   101010xxxxxxxxx…
R2   xxxxxxxx1111xxx…
C    0000000000000000…
```

(b) Non-distinct rules

Figure 4.3: Output of the M-xor function

Privacy concern using the M-XOR *function*

The truth table of the M-XOR function is shown in table 4.2. We can create this table because the Mxor function computes a bit-by-bit output. Because of this property, the circuit is not privacy-safe when the attacker can make multiple requests.

Consider an attacker that follows the protocol but chooses its inputs to try to gain knowledge of the other party's rule. The attacker makes two requests. For a given bit r_{2i} belonging to the

	1	0	x
1	0	1	0
0	1	0	0
x	0	0	0

Table 4.2: Truth table of the M-XOR function

other party, the attacker chooses first $r_{1i} = 0$ and receives the answer $a = c_i$. Then he makes a second request (r_{2i} hasn't changed), setting this time $r_{1i} = 1$ and receives the answer $b = c_i$.

Using the truth table of the M-XOR function, the value of $(a\ b)$ is enough for the attacker to make deductions and learn the value of the rule of the other party.

The possible values for $(a\ b)$ and the deduction associated to each of them are presented in table 4.3.

$(1\ 0) \quad \rightarrow \quad r_{2i}$ is distinct from 0, so it's 1.
$(0\ 1) \quad \rightarrow \quad r_{2i}$ is distinct from 1, so it's 0.
$(0\ 0) \quad \rightarrow \quad r_{2i}$ isn't distinct from either 0 nor 1, so it's x.

Table 4.3: Information gained from performing the M-XOR function multiple times

Thus it takes two requests to learn the full content of the other party's rule.

Reducing information to a minimum.

In order to leak as little information as possible, we should not tell either party *which* bits were distinct. According to the definition of our primitive, we should only return whether *any* bit is distinct of 0, independently of which one. To do this, we can collapse all the bits c_i that we computed in the previous step into a single bit with a OR function.

$$d = c_0 \vee c_1 \vee c_2 \vee ... \vee c_{L-1} \tag{4.3}$$

In ABY, $A \vee B$ is computed as $\neg(\neg A \wedge \neg B)$ (with no performance cost). However, the AND function can only take two operators at once. To perform a OR function between L bits, we use a OR-tree as shown in Figure 4.4. Using the layout of a tree rather than performing all the OR operations linearly reduces the depth of the circuit and improves performance.

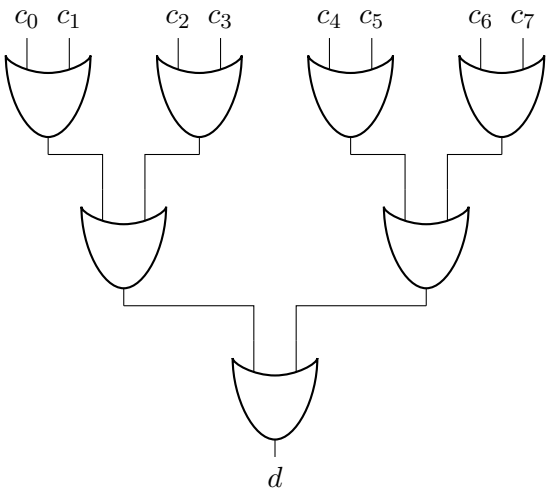


Figure 4.4: Circuit for the OR-tree

This circuit matches exactly our primitive. In the next section, we discuss the privacy guarantees it offers and how an attacker could try to exploit it by choosing its inputs.

Our implementation uses ABY optimizations, notably on SIMD operations, to perform the verification of multiple rules at once. We implemented this parallel comparison using the client-server model described in chapter 3.

4.2 Specialized functions

Using the circuit of the previous section, consider an attacker who chooses its inputs to try to learn information on the other party's rule.

The attacker submits two rules where only the first bit r_0 is different from x, i.e. the rules are of the form `1xxxxxx...` and `0xxxxxx...`. For $i \in \{1..L-1\}$, the value of c_i is 0. Thus, if the output d of the OR-tree is 1, then the value $c_0 = 1$, and if $d = 0$, then $c_0 = 0$. As we saw in the attack on the MXOR circuit, it is possible to find the other party's bit from c_i with two requests.

Using this technique, the attacker can learn any bit i of the other party's input with two requests where all bits $r_j = x$, $j \neq i$. This makes two requests per bit and a rule is L bits long, so $2L$ requests are needed to learn the full rule. If the bits are not independent, it can be lower. An example of one such dependence applies to bits in a given field (e.g. does the other party watch the *source IP* field?).

Can we improve the security further?

Since the output of the above circuit is exactly the definition of the primitive, it is impossible to improve the security without changing our definition. Remember that SMPC ensures the privacy of all information *except what can be inferred from the outputs* of a function.

This means that the primitive should not be used as-is in any application that allows multiple requests. In this section, we discuss specialized functions built upon SMPC on top of the primitive that ensures stronger privacy.

Note: Those functions are defined in the client-server model described at the beginning of chapter 3. As a reminder the model is: "a *client* participant holding a single *candidate rule* and making a request to a *server* participant holding a set of *installed rules*".

4.2.1 Collapsed

The collapsed function aims to answer a question sightly different from the primitive. The formulation is as follows:

> Is the candidate rule distinct from *all* the installed rules?

Suppose that we used the DISTINCT-MATCH circuit for a single candidate rule rc and N installed rules, rs_i, $i \in \{0..N-1\}$. For each rule, we get the output $d_i = rc \star rs_i$, where $\star$ is the DISTINCT-MATCH operation.

If the i-th rule is distinct from rc, then $d_i = 1$. The formula corresponding to the COLLAPSED mode is:

$$out = d_0 \wedge d_1 \wedge d_2 \wedge ... \wedge d_{N-1} \tag{4.4}$$

We see that *out* is only equal to 1 if every bit $d_i \neq 0, \forall i$, i.e. if all installed rules were distinct from the candidate rule.

This function is very similar to the second stage of the DISTINCT-MATCH function, except that it uses a AND-tree rather than a OR-tree as shown in Figure 4.5.

Using this circuit, the set of server rules defines a unique function $f : \{0,1\}^L \to \{0,1\}$. When a rule is not distinct from every server rule, the attacker can not know which rule returned the value $d_i = 0$, so it can no longer perform a bit-by-bit attack.

This protection is beneficial for both the server (even if the client makes multiple requests, he is not able to find any of the server rules) and the client (the server cannot choose to install $2L$ rules that would allow him to learn the client rule).

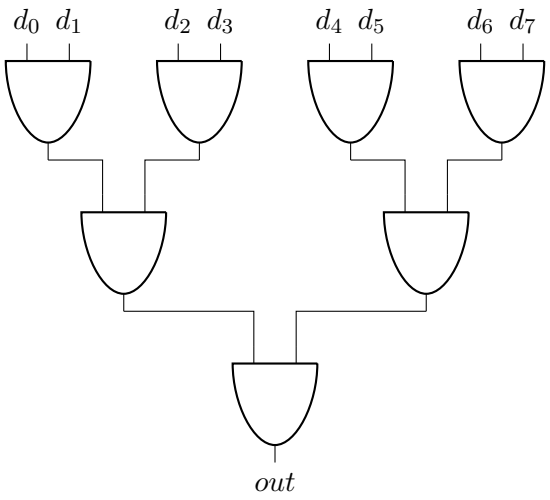


Figure 4.5: Circuit for the AND-tree

4.2.2 Forwarding path

With this function, we want to return an information along with any non-distinct match rule (such as the action part of the rule). That is, given inputs of the form (r_i, v_i) and a dummy value v_d, we want to return:

$$u_i = (v_i \text{ if } d_i = 0 \text{ else } v_d) \tag{4.5}$$

In our implementation, the information associated with each value is the identifier of the AS who receives the deflected traffic.

Mapping information to values can be done easily by adding a MUX gate after the output of the Distinct-Match circuit. This is shown in Figure 4.6

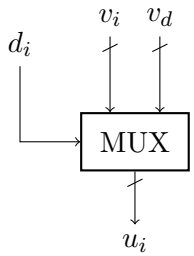


Figure 4.6: Circuit for the value mapper

Using this circuit, we can map any arbitrary information to the results of the primitive. However, we end with the same privacy problem we had when using the primitive itself: when we perform multiple requests against the same rule, an attacker can choose its inputs to discover all bits from the other party.

One solution to avoid leaking information is to outsource the evaluation of the circuit to a non-colluding third party. This approach is described in Section 6.2. However, we want to be able to evaluate the circuit securely without the assumption that such a non-colluding third party exists.

In this situation, we cannot simply aggregate all the results into a single one with a logical or arithmetic function, because we need to return all the values. What we want is to prevent an attacker, making a request against multiple rules at once, from learning which rules matched and which rules did not.

The first step is to ensure that each output is not unique to the match rule, otherwise simply checking whether the value is present allows the attacker to learn if the associated rule was

matched. Respecting this constraint is the responsibility of the application providing the circuit's inputs. In our case, we discuss it in Section 5.2.

Even if the values themselves are not sufficient to decide if a rule was matched or not, it is still easy to answer that question simply by mapping the i-th output with the i-th input. Thus, the second step of our solution is to shuffle all the output values. We perform that shuffling randomly on both the dummy and the real values, without either party knowing the permutation used.

The usual shuffling techniques would loop $\mathcal{O}(n)$ times and exchange the positions of the values in the list. However, our circuit has additional constraints due to efficiency and boolean logic. Instead, we use pairwise permutations based on a random secret key so that each value can end at any position in the output.

We use a conditional permutation gate that is depicted in Figure 4.7. That gate takes two values u_i, u_j as well as a random secret bit k as inputs. u_i and u_j are permuted if and only if $k = 1$.

By permuting each value with the values at distance $2^n, n \in \{0..log(N)\}$, we ensure that the final position of any input is uniformly distributed in the output array.

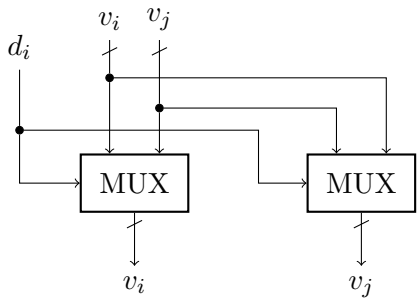


Figure 4.7: Circuit for the conditional permutation gate

We generate the secret random key by making both parties cooperate. Each party c, s generates a random key k_c, k_s and use it as a share of the input. The actual key used is $k = k_c \oplus k_s$, but the parties will not reconstruct it and simply discard their shares once they have been used.

Is the permutation truly random?

The *shuffle grid* we built has a width of N and a depth of $log(N)$. Thus, the random permutation key we generate needs to be of length $\mathcal{O}(N \cdot logN)$. A truly random shuffling would need a key of size $\mathcal{O}(N!)$ to represent all possible permutations. This means that, while each value can end anywhere in the output, the final positions are not independent from each other.

Figuring out the initial and final position of any given value would allow an attacker to find $log(N)$ bits of the permutation key. To figure out the complete permutation, the attacker would need to learn the starting and final position of N values - in others words, all of them.

4.3 Complexity analysis

In Section 1.5, we presented two approaches to two-party computation with boolean circuits, GMW and Yao. In this section, we discuss how each of them scales in regard to the function computed, the length of the rules in bits (L) and the number of rules (N).

We mainly focus on the GMW circuit. Compared to Yao, it has a better asymptotic complexity, more efficient use of pre-computation, as well as lower memory and bandwidth consumption [8]. The main advantage of using Yao's circuit, as seen in the next chapter, is that it often achieves a lower computation time than GMW when the number of rules is small enough.

Because most of the overhead comes from communication, with the actual time required for local computation being low, both protocols scale linearly with the communication delay between client and server.

Note that we are discussing the *online time* of the computation, that is, the time required for operations that are dependent on the inputs. On the other hand, the *setup time* corresponds to the time required for operations that can be done independently of the inputs and pre-computed.

Our main objective is to verify properties quickly when an update occurs, so we assume that the setup phase, also called pre-computation phase, was done in advance and its length is not critical to the application.

GMW protocol

In the GMW protocol, computing a XOR function is free, while computing a AND function requires exchanging a single message (using pre-computed multiplication triples). The NOT gate is based on XOR and is free, while OR and MUX each require a single AND gate.

Since the operations of independent logic gate are parallelized, the total complexity of the circuit is directly proportional to the number of rounds of communication required.

For all circuits from 4.4 through 4.7, the number of communication rounds is summarized in table 4.4

Input	0	Using the input itself as share for the party owning it, while the other party use 0 since $a \oplus 0 = a$.
Output	1	While the AND-depth is 0, the output gate still requires exchanging a message, so it is equivalent to an AND-depth of 1.
Masked XOR	2	
OR-tree	$log(L)$	Each OR gate requires three free NOT and one AND.
AND-tree	$log(N)$	
MUX mapper	1	
Shuffle grid	$log(N)$	We could make it deeper, using a longer key to improve security at the cost of efficiency. However, $log(N)$ is sufficient.

Table 4.4: Number of communication rounds in the GMW protocol for circuits 4.2 through 4.7

In total, the number of exchanges for DISTINCT-MATCH is $3 + log(L)$, COLLAPSED is $3 + log(L \cdot N)$, and FORWARDING-PATH is $4 + log(L \cdot N)$.

Yao's garbled circuits

Unlike with GMW, when using Yao's garbled circuit protocol, the evaluation of the circuit does not need to be executed sequentially. It means that Yao's protocol does not scale with the depth of the circuit. Instead, the number and size of garbled tables that needs to be exchanged depends on the size of the input.

Because of this behavior, the execution of Yao's protocol is proportional to $L \cdot N$. This means that the asymptotic complexity is worse than GMW. Nonetheless, this circuit has a lower baseline execution time, so it is faster for circuits small enough ($L \cdot N \leq \approx 200000$).

Chapter 5

Applications

In this chapter, we discuss concrete applications that can be designed using the circuits created previously as building blocks. These applications use the clear output of the above circuits and can be implemented locally without relying on SMPC anymore.

We start by describing the conflict detection problem for which the COLLAPSED function provides a solution, and its intended use in increasing the accuracy of the solution developed by SIDR.

We identified a few drawbacks in the SIDR approach. In the second section of this chapter, we describe the more advanced SMPC cycle detection protocol that we designed to make a more efficient use of SMPC. That protocol is the result of our design space analysis and primitive development.

5.1 Conflict detection

As described in Section 2.1, SIDR refuses to install a rule if a deflection would send traffic towards an AS that is already present in the DIB. This rejection is independent of whether both ASes actually deflect the same packets, because the *match* rules are not taken into account for SIDR.

However, we argue that rejecting those rules is excessive. To improve the efficiency of the verification, we use the COLLAPSED primitive in order to reduce the amount of false positives.

In SIDR, rules are checked independently for each prefix. By including our SMPC circuit, when an AS detects a risk of cycle over one of the prefixes for which it was trying to install the rule it can, instead of instantly failing the installation, send a request to the other AS and compare the match rules. This is a kind of *second chance* when SIDR alone would reject a rule installation.

Example.
In this example, member M from SDX_1 wants to install a new rule that would deflect traffic towards A. AS A is member of SDX_2 and previously installed the rule ($dPort = 22 \rightarrow$ use route M:N:O) for prefix 10/24.

1. Member M ask SDX_1 to install the rule ($dPort = 80 \rightarrow$ use route A:B:C) for all packets to destination prefix 10/24.

2. When SDX_1 checks the DIB, it realizes that A is already present and refuses the installation, informing M that A is in conflict.

3. M now knows who installed the conflicting rule. He sends A a request to start a SMPC COLLAPSED verification for all rules installed for prefix 10/24.

4. The output reveals that M is not actually in conflict, so it installs its rule.

5. M is added to the DIB, the exception for (M, 10/24) is registered at A as it can no longer install a new rule for this prefix without checking M.

Had SIDR identified multiple ASes in the path that were already present in the DIB, M could have made a SMPC exchange with all of them in parallel, so the actual number of deflections does not affect the overall efficiency.

While this "second chance" should significantly improve the accuracy of SIDR, it only prevents a subset of the false positives, as we discuss next.

5.2 Cycle detection protocol

While SIDR provides a working and flexible solution, there are several problems that we felt should be addressed:

- There are false positives due to not using the *match* part of the rules, as described in the previous section.
- There are false positives due to using a deflection set, which ignores the source and target of the deflections.
- The DIB reveals all deflections a member creates to a possibly large number of network operators who do not need that information, reducing the privacy.
- The system favors rules installed earlier. An AS allowing a rule to deflect traffic towards itself reduces the flexibility it has to install new rules later. This provides an incentive for an AS to install "fake" rules early, in order to ensure there is no conflict for installing a rule later.

We propose an alternative solution for cycle detection that solves the above problems. This system is based on hop-by-hop path exploration starting from the source, but with a focus on loop detection between SDXes.

The protocol works as follows: when a member wants to deflect traffic, it sends a request to its SDX. If the SDX discovers members of another SDX in the AS path, it sends a request for SMPC verification using the FORWARDING-PATH function to the other SDX. If there is a deflection, that function returns a set of *deflection next-hops*, i.e. the ASes towards which traffic is deflected.

We chose to run the protocol on the SDX and not on the AS. This reduces the number of connections required, and by creating them in advance, allow us to pre-compute the setup phase of SMPC, as discussed in Chapter 7.

The assumption of a global SDX repository is identical to the assumptions of SIDR [9]. It can either be a centralized information, similar to existing lists of IXPs [3], or be based on a neighbor-discovery protocol. Our current approach is to consider that an SDX is in the path if at least two members of that SDX appear successively in the path.

The SDX records those deflections in a local directed graph, representing all the SDN-enabled ASes that packets in the flow could go through. Then, it sends a request to the SDXes he just learned about, extending the graph by an additional hop.

The exploration stops when either a cycle is discovered, or there is no more unexplored deflection towards an SDN-enabled AS. The protocol should also stop if the number of nodes in the graph becomes large. It the graph grows too large, the packets can follow many long paths to destination. To avoid that, the verification should be aborted and the match rule should be changed to match fewer packets (i.e. reduce the aggregation level of the rule).

At the end of the exchange, the SDX sends to its member a notification of whether the installation of the rule is authorized. If required, the SDX also sends a notification to all SDXes

deflecting traffic through its member to revoke previous authorization (this power to revoke the rules installed by other ASes is discussed in section 5.2.1). We call this a *token revocation*, because SDXes hold a *token* for each rule they are allowed to revoke.

Figure 5.1 shows the messages exchanged between the members and SDX for a single member in the path, according to the scenario described in Section 5.2.4. In this diagram, the verification starts with M sending to SDX_1 a request to deflect traffic towards AS path (Z A X), then SDX_1 sending to SDX_2 the identifier of the members that need to be verified. After that, members and SDXes exchange the public cryptographic keys K_1 and K_2, and shares of the rules r_{ij}. Those shares contain the rules encrypted such that $r_X = r_{X1} \oplus r_{X2}$, which is why one of the shares need to be further encrypted with asymmetric cryptography when transferred to the remote SDX (we could also use another form of secure channel).

At the end of the protocol, SDX_1 allows or forbids the rule installation. If necessary, the SDX issues revocation tokens to other SDXes (in this case, SDX_1 was holding a revocation token for SDX_2).

This example shows an exchange between only two SDXes, the initiator and the SDX located downstream in the path. This scheme could also be performed with several SDXes at the same time by setting up connections with multiple SDXes and members in parallel.

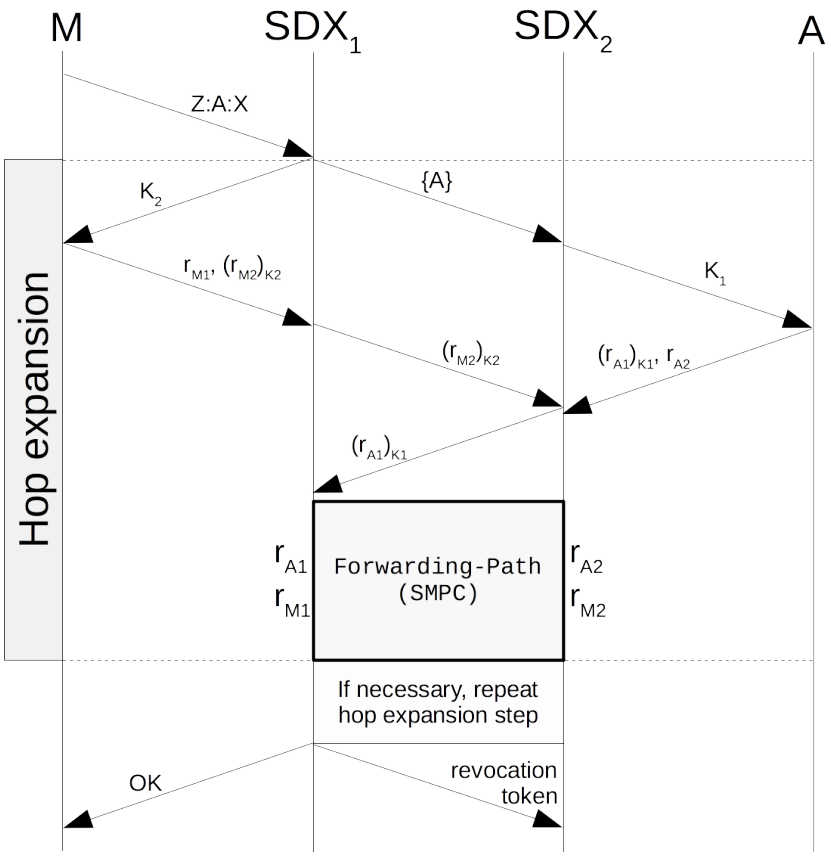


Figure 5.1: Messages exchanged between participants to perform the cycle detection protocol

In our implementation, the network configuration should take into account the chances that a rule might be revoked and the chances that a rule would be rejected. To avoid a rule being revoked, operators should avoid creating deflections on unstable routes. Another solution to prevent a rule from being revoked is to make a choice of leaking some parts of it. This is discussed in Section 6.1.

An important effect of our constraints is that operators should avoid installing deflection rules for short-lived BGP routes, as there is a non-negligible probability that the route will be

withdrawn at the operator or downstream before the rule is accepted.

To avoid a route being rejected, operators should avoid making them too broad. It is also possible to send multiple requests at once, so that if the broad rule is rejected, the more specific one can still be installed. By installing multiple rules at once, operators also need to revoke the rules installed by other ASes only once.

5.2.1 Advantage for downstream ASes

The downstream AS gives the upstream AS an authorization to deflect traffic through its network. Because of this, the downstream AS should always be capable of withdrawing its authorization, either to install its own deflections or for any other reason. We call this right to arbitrarily remove the authorization the *advantage* of the downstream AS, and we use two different mechanisms to preserve that advantage.

First, if a member indicates a deflection to itself, it immediately creates a loop - essentially sending the information that the AS refuses to receive deflected traffic for this flow. This is part of our design that the members are not required to tell the truth and reveal their private information if they do not wish to do so. Of course, preventing the installation of a rule is likely to reduce the traffic going through that AS.

Second, if the installation is accepted, every member in the graph, that was contacted by the one doing the verification, receives a revocation token. The revocation token can be used to withdraw the deflection authorization. This is an additional incentive not to use large rules and long AS paths, as that would mean more parties are allowed to revoke your rules.

5.2.2 Privacy preservation

Not all the information exchanged is transmitted publicly. When a member sends its rules to the SDX, it should not send them in clear. Instead, the member sends an encrypted share of the rules to each SDX. They are then able to perform the computation without knowing the policies of the members. As a reminder, we explained in Section 3.4 that we consider the rules to be private, even in regard to a member's own SDX.

In addition, the output of Forwarding-Path should not reveal the individual identities of the members. What is returned by the SMPC circuit is the identity of the next SDX in the deflected path, and we use public key cryptography with the key of said SDX to transmit the identities of the members that are crossed. Each SDX still knows which ASes are crossed at the next-hop (because that information appears in the BGP AS path), but this solution hides which member created a deflection and towards who the traffic is sent.

Using this technique enables a granularity of information at SDX level rather than AS level, which provides much stronger privacy guarantees albeit at the cost of some false positives in very specific cases (discussed in the next section).

5.2.3 False positives

There are two situations where false positives can occur.

In order to provide stronger privacy guarantees, we do the cycle detection at SDX level rather than at AS level. The main advantage of this coarse verification is to avoid revealing who installed the matching policies as well as their destination, exploiting the large number of IXP members to hide information.

However, this increased privacy comes at a cost. Consider the situation represented in Figure 5.2 with members M, N at SDX_1 and A at SDX_2. Member A installs a deflection rule towards N. M then wants to install a matching rule towards A. The protocol detects that SDX_2 is already sending traffic towards SDX_1, and installing the new rule would create a loop along $(SDX_1\ SDX_2)$. However, the actual path is $(M\ A\ N)$ and does not contain any loop.

This kind of path is unlikely: M and N, belonging to the same IXP, probably have a much faster connection than going through A, and sending the traffic outside makes little sense. Because of this, we strongly believe that the privacy gained by this trade-off is worth the reduced accuracy. However, should one disagree, it is possible to run the protocol at AS granularity by removing the encryption on the inputs of the FORWARDING-PATH function.

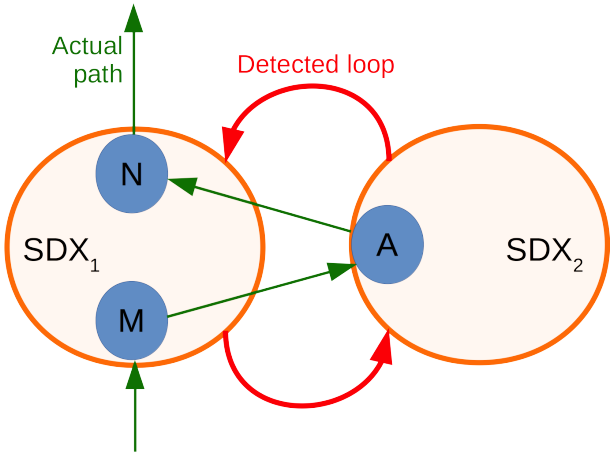


Figure 5.2: False positive due to performing the verification as SDX granularity

The other situation where a false positive can occur is when a downstream deflection only matches part of the packet flow. Consider the Figure 5.3 where ASes A, M and S are members of SDX_1, SDX_2 and SDX_3, respectively. M sends traffic with destination port 80 to S ; S sends traffic with destination port 81 to A ; and A wants to send traffic matching either port 80 or 81 towards that destination.

Because A's rule is not distinct from M's rule nor from S's rule, a non-existing cycle is detected. However, this does not mean that the rule is definitely rejected. A holds a revocation token for S's rule, since it allowed it before. A can issue that token, breaking the loop, and install its own rule. S would then make a new request for the same rule, and since it is distinct from M's rule, no loop is detected by S.

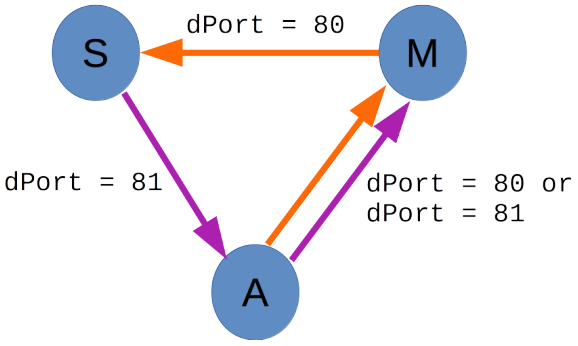


Figure 5.3: False positive due to the downstream member matching only a subflow of the new rule

5.2.4 Example

This describes a scenario where M, member of SDX_1 wants to install a new SDN rule crossing SDX_2. It must check whether A or B, both in the AS path, create a deflection.

Figure 5.4 illustrates the network topology used in this example, along with the BGP announcements that are deflected.

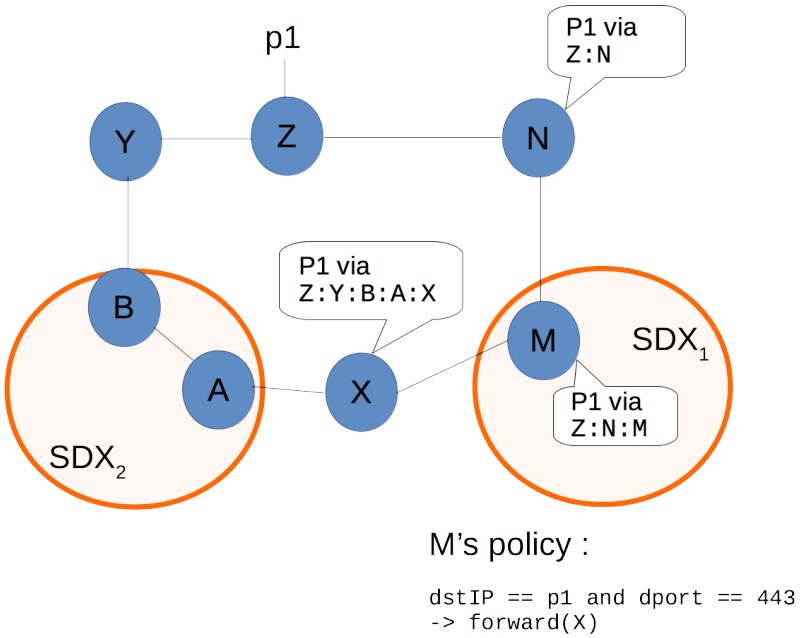


Figure 5.4: Network topology for the cycle detection protocol example

M learned the path it wants to use from AS X in BGP. That ASPath is Z:Y:B:A:X. On this path, A and B belong to the same SDX (This could be any non-zero number of SDX members, as long as there is no other SDX in-between. Two corresponds to the case where A is the AS where the packet enters SDX_2, and B is where the packet leaves the SDX).

In this example, there is no other SDX on the expected path.

1. M sends SDX_1 a request to install a rule that would redirect traffic on path Z:Y:B:A:X according to BGP
2. SDX_1 looks at the path and realize that there are members of another SDX on that path. It replies to M with the identity of SDX_2
3. M sends shares of its rule to SDX_1 : r_{M1} and $(r_{M2})_{K_2}$, the latter being encrypted with the public key of SDX_2
4. SDX_1 sends to SDX_2 the share $(r_{M2})_{K_2}$ and the names of A and B
5. SDX_2 asks A and B to send r_{A2}, $(r_{A1})_{K_1}$, r_{B2} and $(r_{B1})_{K_1}$. Those shares include the active rules and the next SDX that create a deflection in the path for each rule. SDX_2 sends the encrypted shares to SDX_1
6. SDX_1 and SDX_2 perform SMPC computation to find the set of possible SDXes the flow could be deflected to.
7. - If the set is empty (no deflection towards an SDX), continue to step 8.
   - If the set is non-empty and SDX_1 already contacted at least one SDX in the set ; or SDX_1 is in the set, reject the rule.
   - If the set is non-empty and SDX_1 did not contact any AS in the set, repeat steps 3-7 for each member in the set.
8. M installs the rule and records SDX_2 as the next SDX in the path. SDX_2 records that SDX_1 added a rule for A and B so that it can withdraw its authorization if either want to install a new rule.

Optionally, on withdrawal of a deflection authorization, SDX_2 gives SDX_1 (and indirectly M) a chance to test its rule against the new one instead of running the whole check again.

A proof-of-concept implementation of the protocol was built in Python to simulate the function of the protocol. A practical implementation would require focusing on parallelization and asynchronous request to achieve lower runtimes in potentially larger networks.

5.2.5 Path exploration in the control plane

Our protocol focuses on detecting cycles between SDXes. To do this, we only explore points in the network where deflections occur.

However, with minor changes, our protocol could explore all possible paths that a network packet might follow. This problem has several applications. By learning the path followed by packets, it is possible to anticipate network failures. In terms of privacy, discovering the paths allows network operators to avoid surveillance states, a concern expressed by multiple countries [2].

To solve this problem, the modified Forwarding-Path circuit would return the full AS path instead of the members creating a deflection. Some assumptions have to be discussed (is the semi-honest model sound?) and privacy considerations arise (how does this interferes with BGP export policies, which are confidential?).

This topic is interesting for works such as [20]. In that paper, Edmundson et al. explored the paths followed by packets by sending requests in the data plane. However, this approach only provides results for a subset of the traffic (e.g. HTTP traffic). In a network deploying SDN, packets with the same destination address can follow different paths depending on other header fields, such as their IP protocol (TCP, UDP) and application protocol (depending on the destination port).

In this type of network, a data plane exploration of traffic paths would be much more costly and require testing multiple header configurations. By using our approach of exploring the paths in the control plane instead, it is possible to learn multiple paths at once, which takes into account e.g. application-specific peering and load balancing.

Network operators also gain to benefit of being able to prove some contractual guarantees to their users (e.g. “we do not send sensitive traffic through a country for which there is an embargo”) without revealing the exact path they chose.

5.3 Comparative evaluation questions

In the next chapters, we discuss optimizations and show the execution time for our cycle detection protocol, leveraging our design of the protocol and micro-benchmark results to get an estimate. However, some questions are interesting to discuss before comparing the performance of different implementations.

Different implementations have vastly different constraints. Here are some aspects that need to be taken into account:

1. How does propagation delay affect the verification protocol?
2. Are there false positives?
3. Should operators modify their behavior?
4. Does the verification require everyone on the network to adhere to the protocol?

We will study each of these questions for three categories of solutions: first, our own protocol based on SMPC; then the formal static checkers such as [30] and [32]; then SIDR [9] as a realistic implementation.

Table 5.1 provides a summary of the points discussed in this section.

Propagation delay. Because static checkers assume knowledge of the whole control plane, they do not measure the time required to gather that information in a single point, thus ignoring the network delays. This is realistic when using a single controller that receives the policies and configures the control plane topology, but unrealistic in the inter-domain setting.

SIDR assumes the use of a global Deflection Information Base to store information that needs to be verified, which means that either all operators accept a delay to ensure consistency. Without this delay, the network could enter an inconsistent state if two ASes update the DIB at the same time.

Our implementation use path exploration from the source, thus it does not need to wait after the verification has completed. However, when an AS installs a new rule, it must revoke the authorizations for traffic going its network. The notification of these revocations (which we call *revocation token*) needs to travel upstream and packets might be forwarded by another AS before the conflicting rule is removed. This means that, when revocation tokens are issued, there is a risk of transient loop.

False positives. Static checkers do not have any; SIDR may exhibit false positives when two ASes in a path cause a deflection for the same prefix; our solution may exhibit false positives but these would be smaller subset than those in SIDR.

Additionally, correctly measuring false positives requires making assumptions about the rules that are installed. How fine-grained are those rules? How many prefixes do they apply to? Which values are more likely to be appear in the rules? In the absence of a real data set of rules, those decisions are up to the researchers who make the measurements.

Operator collaboration. Because the full network has to be known, static checkers require participation of every actor in the network . On the other hand, SIDR and our implementation only require the participation of SDN-enabled IXPs and ASes.

Because SIDR and our solution only require SDN-enabled ASes to participate, they are both more fit for incremental deployment.

Privacy. The static checkers assume complete visibility of the control plane, so they do not provide any privacy unless the verification is performed by a trusted party. SIDR provides privacy guarantees, but assumes that members are willing to reveal which prefixes they are deflecting and propagates that information. Our solution considers the entire rules to be private and only reveals forwarding information if it is necessary and the AS chooses to make it public.

	SMPC Protocol	Static Checkers	SIDR
Propagation Delay	Can cause transient loops	Ignored	Can cause inconsistencies
False Positives	Low	None	Medium
Global participation	Only if deflections	Yes	Only if deflections
Privacy	High	None	Medium

Table 5.1: Comparison of the trade-offs between different network verification tools

Chapter 6

Optimizations

Secure computation is difficult to optimize because intermediate results are hidden, and thus impossible to both prune and cache. In this chapter, we discuss some techniques that can be leveraged to reduce the SMPC input size and the number of necessary operations.

6.1 Public bits

So far, we considered a rule to be entirely private. However, this assumption is too strong and significantly reduces the flexibility of our approach.

In practice, some information does not necessarily need to be private. For example, an operator might decide that the fact that his rule selects TCP traffic is not a private information. Another example is that when applying a rule to a particular BGP route, the prefix must be included in the match rule as an additional *match field*, but because that information comes from BGP it is considered public already.

Alternatively, the operator can voluntarily leak some private bits. It could be a large IP prefix (on a rule affecting a /24 prefix, leak the first 16 bits), or randomly chosen bits that do not provide information without some correlated bits (such as the least significant bit of the destination port).

We call those bits from the match rule that operators are willing to reveal *public bits*.

The simplest use for the public bits is for the client to send them along with a SMPC request. The server will locally make a first pass on its installed rules and filter out those that are already disjoint from the candidate rule based on the public bits only.

Another, even more important use is to improve stability of the cycle detection protocol. Normally, as soon as a downstream AS installs a new rule, it issues all revocation tokens in its possession. This is problematic, as installing a single rule can potentially remove many other ones. If it can be established that the new rule is distinct from those used by the upstream members based on public bits, then it is not necessary to issue the revocation token.

It is important to note that in both cases, leaking information is directly beneficial to the member who revealed its own policies, because it speeds up the verification of the new rule and reduces the probability that said rule, if accepted, will be revoked at a later point. This provides a strong incentive to exploit the trade-off for network operators.

How many rules are rejected earlier by using the public bits optimization?

Assuming that, for a bit i, the probability it is watched by the other party is α_i (and if watched, it has equal chances to be either 0 or 1), the probability that said bit is sufficient to establish that the candidate rule is distinct from any server rule with this bit is $\frac{\alpha_i}{2}$.

Using this approximation, a network operator can compute the expected number of installed rules that it needs to compare against, as illustrated in table 6.1 under the assumption that all bits have an equal chance of being watched. (To put things in perspective, an IPv4 rule

is expected to be 104 bits long with both source and destination addresses, ports, and the IP protocol such as TCP or UDP.) In that table, α is the probability that each bit is watched, and n (the row label) indicates the number of bits that are made public.

n	α			
	0.1	0.25	0.5	1
1	5%	12.5%	25%	50%
8	33%	66%	90%	99.7%
16	56%	88%	99%	≈100%

Table 6.1: Expected fraction of the installed rules excluded early

6.2 Outsourcing

It is possible to outsource SMPC operations to external, non-colluding parties. This is useful to make computations with the inputs of more than two participants or facilitate the setup of pre-computed circuits. We propose another use, which aims to leverage the third party and encrypted inputs to perform some parts of the computation outside of SMPC.

In Section 4.1.2, we saw that revealing the result of the M-XOR gate leaks information when the attacker can make multiple requests and chooses its inputs carefully.

Because the third party is not the one choosing the inputs, it would only gain very little knowledge of the rules if the output of the MXOR gate were revealed to it. The third party can infer, for each value 1 in the output, that one of the rule matches 0 and the other rule matches 1. The third party can not learn which of the rule matched the 0 and which matched 1. It also can not know if a value 0 in the output is caused by one of the rules matching x, or by both rules matching the same value.

Using this property, it is possible to output the bits $c_0, c_1, ...c_{L-1}$ to the third party, who then performs the collapsing of the OR tree locally instead of using SMPC.

With this approach, we reduce the number of communication rounds required for the computation from $3 + log(L)$ (where L is the number of bits in a rule. For $L = 104$, our most expected case, that's 10 exchanges) to only 3.

6.3 Deflection graph caching

When an SDX follows the protocol, it builds the *deflection graph* and offers revocation tokens to all ASes in it. It is expected that when one token is received, the rule is dropped altogether and the initiating member is free to run the whole protocol again.

However, the SDX could very well keep the deflection graph in memory. By rewriting and performing a new verification only with the nodes in the graph for which revocation tokens have been issued, or new nodes that have been added to the graph by the update, it is possible to significantly reduce the overhead of update verification.

It is also possible to cache results from failed installation attempts, when the graph has been partially built. This has two uses: retrying a later installation (if a conflicting rule is removed), and retrying an installation for a rule matching only a subflow of the original one, since that subflow necessarily uses one of the paths used by the larger flow.

In particular, the latter use case can be leveraged to refine rules on failed installation. For example, if a rule that would match both HTTP and HTTPS traffic is rejected, the operator might want to apply it for HTTP only instead.

6.4 Trade-offs summary

Table 6.2 synthesizes the biggest trade-offs choices that affect the efficiency (time required), accuracy (false positives) and privacy of the verification. For each trade-off, the table contains a reference to the section which explains our design choices and a short description of the benefits and drawbacks.

Techniques	**Benefits**	**Drawbacks**
AS granularity in the deflection graph (5.2.3)	Removes false positives in the cycle detection when traffic bounces between different SDXes.	Reveals which AS installed a deflection, and towards whom.
Reveal intermediate results (6.2)	Significantly reduces the execution time of the computation.	Reduces the privacy if an SDX and a member collude.
Using Yao's circuit (4.3)	Speeds up the computation for smaller problems.	Does not scale to large inputs.
Run SMPC on members rather than SDX (5.2)	Removes the need for assumption that SDXes do not collude against their own members.	More connections necessary, making pre-computation more difficult to set up.
Limit deflection graph depth (5.2)	Avoids long paths ; reaches a decision faster ; can refine the rules.	Causes false positives when the paths are long but loop-free.

Table 6.2: Summary table of the trade-offs between different configurations

Chapter 7

Experimental results

In this section, we present a comprehensive evaluation of our implementation of the primitive. We measure the execution time for running the different SMPC circuits presented in Section 4, then compare the results and discuss the usefulness of using an outsourced approach to reduce the overhead of running the computation securely. We also initiate a preliminary discussion on the possibility of using our primitive to efficiently solve the cycle detection problem. Finally, we give an estimation of the practical runtimes for the full cycle detection protocol.

7.1 Setup

We performed our experiments on a single dedicated server with 16 hyper-threaded cores at 2.60 GHz with 128 GiB of RAM and Ubuntu Linux 14.04. This server runs both the server and client components of SMPC. To simulate network delays, we used the `tc-netem` tool to add a delay on the localhost interface.

The execution time was tested for comparing rules with three sizes of match patterns. The first one is 13 bytes long, which corresponds to an IPv4 source address (4 B), IPv4 destination address (4 B), TCP/UDP source and destination ports (2 B each), and layer 3 protocol (1 B) ; the second size is 37 bytes, which corresponds to the same fields but using IPv6 addresses instead ; the third size is 54 bytes, which is the maximal length of an IPv4 header plus the minimal length of a TCP header. The size of the header, in bytes, is written L.

Various network latencies D were tested, ranging from 1 to 250 milliseconds. This covers most even long-distance connections between ASes [6].

Finally, we vary the number of rules installed on the server side, N, from 1 to 10000.

All results are in milliseconds and averaged over 25 runs of the circuits.

The graphs answer the following comparison questions:

- Which of the GMW and Yao circuits is most efficient?
- How do the circuits scale with the number of rules compared simultaneously?
- What is the cost of running more complex functions securely, compared to only running simple functions and relying on outsourcing to ensure privacy?
- How do the circuits scale with the length of the rules?
- How do network delays affect the time required to complete the execution?
- Is it possible to run many instances in parallel?

The graphs in Section 7.2.2 show the cost of using the primitive to exchange larger blocks of information with each rule, as we described in Section 4.2.2.

7.2 Micro-benchmarks

We now discuss the efficiency of our implementation of the primitive and the effects of adding the collapsing trees to reduce the information in the output. We then compare the results for different functions, analyzing the cost of achieving better privacy.

At the end of each section, a subset of our results is shown in the form of tables. The figures are based on the full set of our results and show more data points than the tables.

SMPC separates the execution in setup phase (which can be pre-computed independently of the inputs) and online phase (which requires the inputs). We mostly focus on the time required for the online phase ("online time"), because we expect the setup phase to be performed in advance. Figures 7.3b and 7.4b show the time required for the setup phase depending on the number of rules.

All graphs varying the number of rules use a logarithmic scale for the X-axis, which provides a better fit for the tested data points. Linear scaling appears as exponential curves in the figures.

7.2.1 Primitive and collapse-tree based functions

This section describes the functions based on the OR-tree and AND-tree circuits presented in chapter 4.

We refer to the different possible circuits as "modes of operation". We distinguish three such modes.

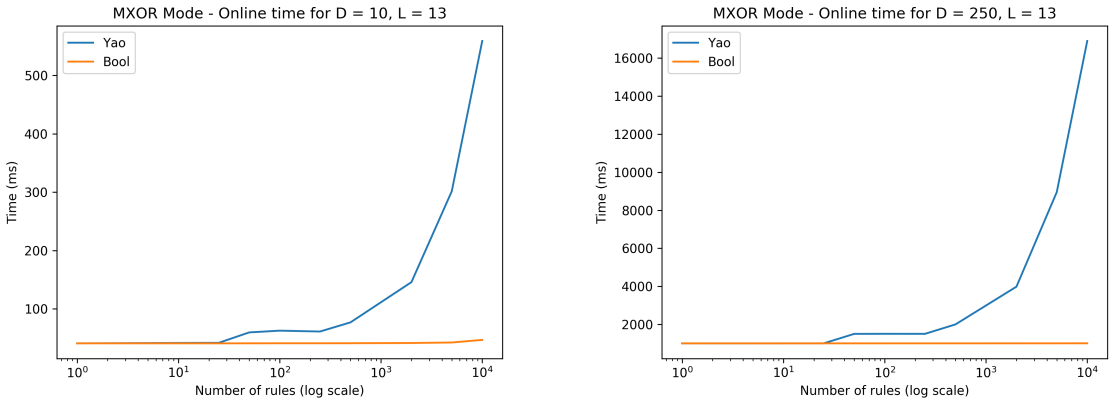
1. Mxor mode, which only performs the bitwise comparison as shown in Figure 4.2.
2. Clear mode, which returns a single bit per rule. This mode outputs exactly the result of our primitive. Its circuit is created by adding the OR-tree from Figure 4.4 after the output of the Mxor mode.
3. Collapsed mode, which returns a single bit telling if the candidate rule was different from *every* server rule. Its implementation is realized by adding the circuit from Figure 4.5 after the output of Clear mode.

The following charts illustrate the evolution of the circuits' behavior in regard to the problem size. Furthermore, they highlight the difference in behavior between using Yao's garbled circuits and the GMW protocol with boolean circuits.

Plots are provided to give a representation of the trend and order of magnitude. They will be discussed in relation to our expectations and design choices. Accurate numerical results are summarized in tables 7.2 (Mxor), 7.3 (Clear) and 7.4 (Collapsed).

Our evaluation takes five parameters:

- The mode of operation
- The circuit used (Yao for the garbled circuits protocol, boolean for GMW)
- D, the network communication delay
- L, the length of the rule in bytes
- N, the number of rules



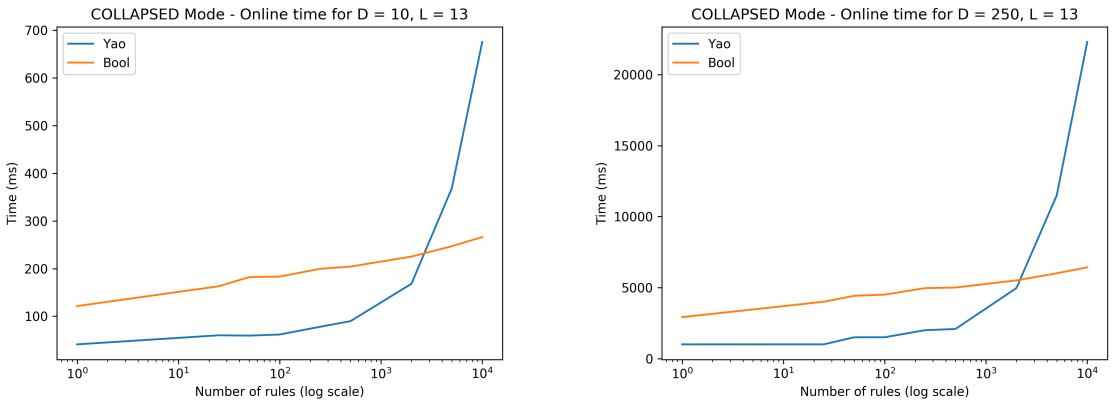
(a) Low delay (10 ms)

(b) High delay (250 ms)

Figure 7.1: Online time for the MXOR mode, varying number of rules

Figure 7.1 shows the execution time for the Yao and boolean circuits in the MXOR mode. In this mode the number of communication rounds is constant, so the execution time using the boolean circuit does not change. This is illustrated by the orange line being constant at around 40 ms for a 10 ms delay and 400 ms for a 100 ms delay.

The circuit based on Yao's protocol performs significantly worse than GMW and exhibits a trend that is linear in the number of rules.



(a) Low delay (10 ms)

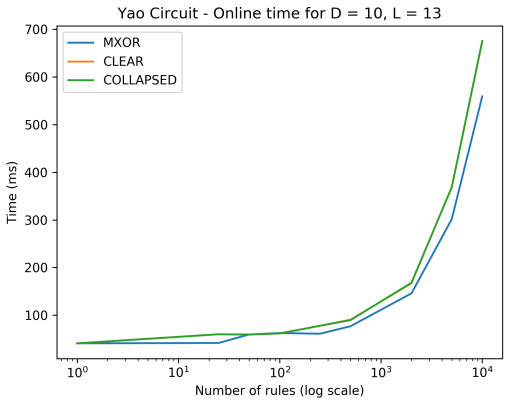
(b) High delay (250 ms)

Figure 7.2: Online time in the COLLAPSED mode, varying number of rules

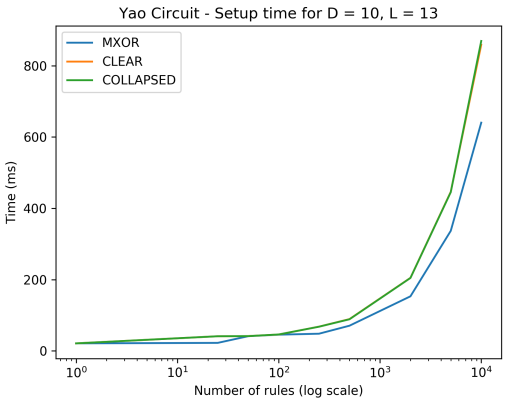
Figure 7.2 shows the same information when running in the COLLAPSED mode. The first observation is that the boolean circuit, unlike Yao, has a fixed overhead even with only a few rules. This is mainly caused by the number of communication rounds required for the OR-tree. This number grows logarithmically with the length of the rules and requires seven rounds of communication for 13 bytes rules.

The boolean circuit now scales logarithmically to the number of rules, which matches our analysis in Section 4.3. Yao's circuit still scales linearly.

Because the curves for each circuit grow at different rates, it is fairly easy to predict which of the two approaches is the most efficient depending on the parameters. Preliminary results show that the cutoff is around the point $L * N \approx 30000$, independent of the delay.



(a) Online time

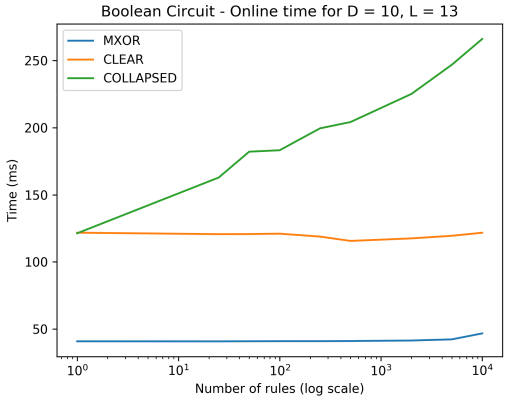


(b) Setup time

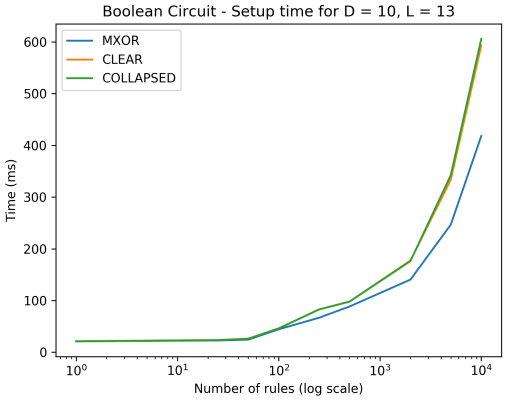
Figure 7.3: Computation time among all modes, using Yao's circuit. The orange line is merged with the green line.

The curves in Figure 7.3 show the results using only Yao's circuit in all three modes. The results in Figure 7.3a are the times for the online phase, while Figure 7.3b shows the setup phase. Note that the orange curve for CLEAR mode is fully merged with (and hidden by) the green COLLAPSED curve. It shows that both the online and setup times scale linearly to the size of the input, and that changing mode makes little difference.

Because Yao's protocol is based on the exchange of encrypted functions, it directly scales to the size of the input. Indeed, a larger number of possible inputs yields a larger number of possible outputs, and thus a larger truth table. On the other hand, the actual function does not have a significant impact, because creating the encrypted table is done locally and not affected by communication delays.



(a) Online time



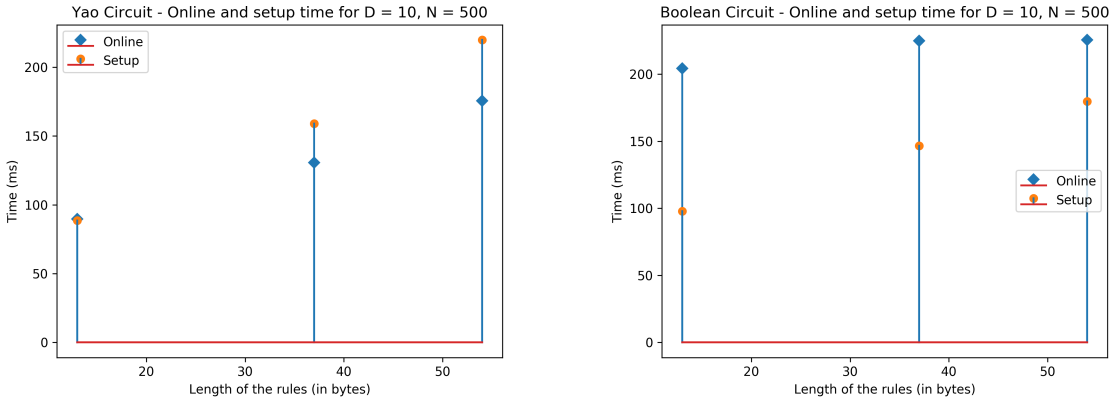
(b) Setup time

Figure 7.4: Computation time in all modes, using the boolean circuit

With the boolean circuit, unlike Yao, the choice of mode is very important, as shown in Figure 7.4. The online time for MXOR and CLEAR modes does not change depending on the number of rules, but the latter always takes longer. In the COLLAPSED mode, the execution time increases with the number of rules, but only does so logarithmically. This makes interesting to compare many rules at once using the boolean circuit rather than Yao.

Unlike the online time, the setup time increases linearly with the size of the input because more multiplication triples need to be generated for larger problems. Indeed, the setup time

grows with the number of gates, and placing the gates in the shape of a tree only reduce the depth of the circuit, not the total number of gates.



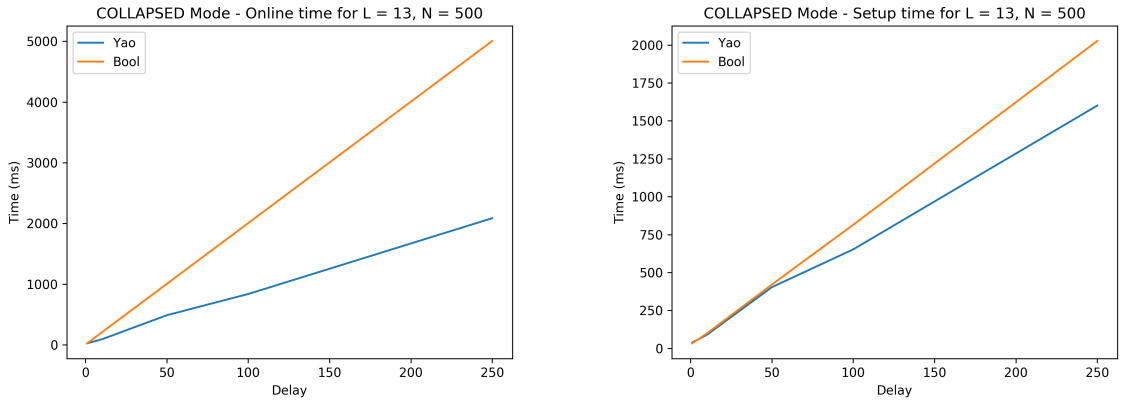
(a) Yao circuit

(b) Boolean circuit

Figure 7.5: Online and setup times for rules of varying length (13, 37 and 54 bytes) in COLLAPSED mode

Figure 7.5 confirms our intuition that the execution time increases with the length of each rule identically as it does to the number of rules. It appears by observing the blue dots in Figure 7.5b that the online time growth is sub-linear for the boolean circuit (it is in fact logarithmic because of the growth of the OR-tree depth). On the other hand, the orange dots show a linear growth. Similarly, both online and setup times for Yao's circuit grow linearly in Figure 7.5a.

For a given delay, the "size of the input", that is, the product $(L * N)$, is enough to characterize the complexity of each approach. The online time with the boolean circuit increases logarithmically to the size of the input, while the online time for Yao, setup time for Yao and setup time for boolean grow linearly with the size of the input.



(a) Online time

(b) Setup time

Figure 7.6: Evolution of execution time with communication delays

Finally, using the COLLAPSED mode, we confirm that for both Yao and boolean circuits, the online time (Figure 7.6a) and setup time (Figure 7.6b) increase (almost) linearly with the communication delay between the two parties.

For a communication delay of 0, the execution time is basically equal to the time required for local computations, which does not depend on the delay. Observe in Figure 7.6 that when the

delay is close to zero, the execution time is also close to zero. We conclude that the execution time is mostly dependent on the communication time, thus the delay among the two entities.

Because the SMPC circuit spends most of its time in idle state, waiting for information from the other party, it would be easy to run multiple instances of SMPC in parallel, provided the machine has a sufficient bandwidth.

Circuit	Num. rules	Setup	Online	Rules
Yao	2000	2.9 MiB	107 MiB	108 kiB
Boolean	2000	41 Mib	650 kiB	108 kiB
Boolean	10000	207 Mib	3.2 Mib	540 kiB

Table 7.1: Bytes exchanged for computation (54 Bytes rules, COLLAPSED mode)

Table 7.1 shows the amount of data exchanged for the computation with many rules of 54 bytes. We also show the size of the rules themselves as a baseline on the information that needs to be exchanged to perform the computation in a non privacy-preserving way.

To provide detailed information on the test results, the following tables show, for each mode, the time required to complete the setup and online phase (in milliseconds) with rules of various lengths, different numbers of rules, and different orders of magnitude for the communication delay.

		Rules [bytes : number]								
Delay		13:50	13:500	13:5000	37:50	37:500	37:5000	54:50	54:500	54:5000
		Yao								
1 ms	setup	7.705	28.24	264.8	10.73	77.76	761.3	14.19	113	1130
	online	8.131	19.93	176.1	8.371	53.53	475.6	10.1	73.07	689
10 ms	setup	41.14	70.58	336.7	46.18	124.9	887.6	47.19	163.8	1273
	online	59.56	76.76	301.5	62.01	115.9	778.7	61.18	156.4	1099
100 ms	setup	400.8	607.7	2634	405.6	1015	6713	406	1221	9653
	online	599.6	797.4	3560	602	1189	9450	601.2	1769	13580
		Boolean								
1 ms	setup	6.398	24.39	186.3	10.93	57.59	510.6	13.23	80.75	739.3
	online	4.77	4.895	6.072	4.879	5.525	8.698	5.129	5.913	12.54
10 ms	setup	24.32	88.37	246.6	45.65	124	567.6	47.93	143.7	788.8
	online	40.85	41.03	42.26	41.12	41.64	53.12	41.24	42	64.85
100 ms	setup	204.4	685.2	1674	405.8	1012	3640	408.1	1125	5205
	online	400.8	401.1	402.3	401.2	401.7	494	401.3	402.2	536.5

Table 7.2: Online and setup time (in milliseconds) for MXOR mode, with various parameters

		Rules [bytes : number]								
Delay		13:50	13:500	13:5000	37:50	37:500	37:5000	54:50	54:500	54:5000
		Yao								
1 ms	setup	8.59	38.57	364.8	13.65	106.1	1067	18.72	153.9	1565
	online	8.423	24.03	216.1	9.726	64.5	593.9	11.92	90.53	859.9
10 ms	setup	41.47	89.14	446.2	46.76	158	1217	63.46	218.9	1760
	online	59.44	90.39	369.2	60.79	130.4	929.7	74.74	176	1334
100 ms	setup	400.9	677.2	3659	405.9	1219	9767	406.4	1815	14150
	online	599	861.8	4552	600.7	1573	12320	599.2	2161	17860
		Boolean								
1 ms	setup	8.282	33.5	277	13.35	84.75	753.4	16.87	121.3	1102
	online	13.79	14.02	15.17	16.44	16.77	20.46	16.46	17.1	26.27
10 ms	setup	26.12	97.92	333.4	48.06	146	826.6	68.95	178.4	1161
	online	120.7	115.6	119.4	141.3	137	146.5	142.1	139.3	160
100 ms	setup	206.2	813.8	2238	408.2	1103	5249	608.9	1273	7427
	online	1113	1110	1119	1312	1345	1406	1315	1331	1409

Table 7.3: Online and setup time (in milliseconds) for CLEAR mode, with various parameters

		Rules [bytes : number]								
Delay		13:50	13:500	13:5000	37:50	37:500	37:5000	54:50	54:500	54:5000
		Yao								
1 ms	setup	8.673	38.66	367.4	13.76	106.4	1066	18.97	154	1561
	online	8.464	24.06	220.5	9.71	65.6	596.8	11.94	89.16	860.6
10 ms	setup	41.39	88.59	445.2	46.71	158.9	1213	63.02	219.8	1761
	online	59.28	89.61	368.2	60.8	130.8	934.2	74.75	175.5	1326
100 ms	setup	400.8	652.4	3661	405.7	1227	9767	406.4	1815	14200
	online	599	836.7	4585	600.7	1575	12330	599.2	2168	17890
		Boolean								
1 ms	setup	8.31	32.85	277.6	13.36	83.55	767.5	16.96	120.1	1097
	online	20.42	23.65	30.31	23.17	26.75	36.11	23.39	27	42.3
10 ms	setup	26.12	97.79	342.2	48.32	146.5	807.9	68.96	179.8	1160
	online	182.1	204.2	246.9	202.6	225	281.1	203.5	225.5	290.9
100 ms	setup	206.1	814.5	2237	408.2	1118	5242	609	1297	7428
	online	1714	2005	2408	1913	2206	2706	1916	2206	2718

Table 7.4: Online and setup time (in milliseconds) for COLLAPSED mode, with various parameters

7.2.2 Information-mapping based functions

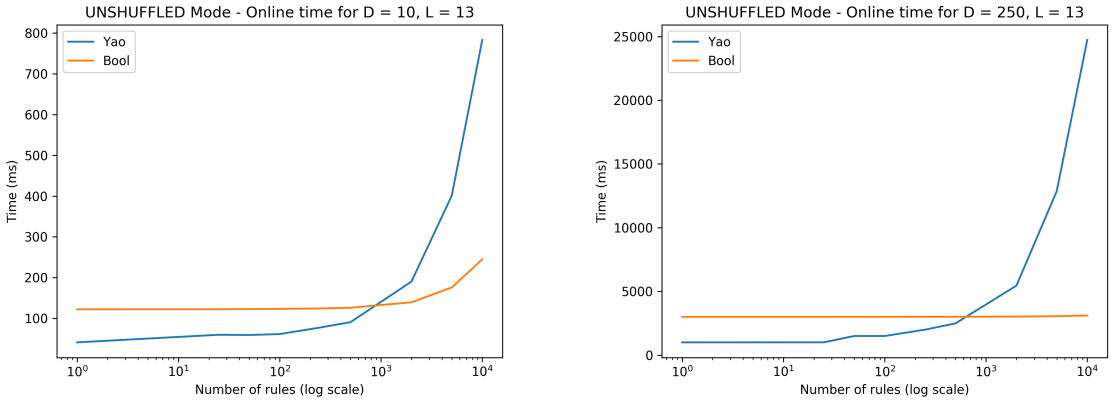
This section describes the functions that output some information depending on the results of the primitive, i.e. the FORWARDING-PATH function described in Section 4.2.2.

We describe those functions as "information-mapping" functions because they map one out of two possible values to each input rule, depending on the result of the primitive function.

The graphs are similar to those above, but for circuits returning a block of information rather than a single bit for each input. In this example, the forwarded information is a 16-bits AS number per rule.

We have two modes. UNSHUFFLED means that we simply add the circuit from Figure 4.6 to the output of the CLEAR circuit. We show that this approach of mapping larger data to the output does not significantly increase the computation time. The unshuffled mode can also be used with the outsourced approach of Section 6.2.

The second mode is called SHUFFLED and is the one described in Section 4.2.2.



(a) Low delay (10 ms)

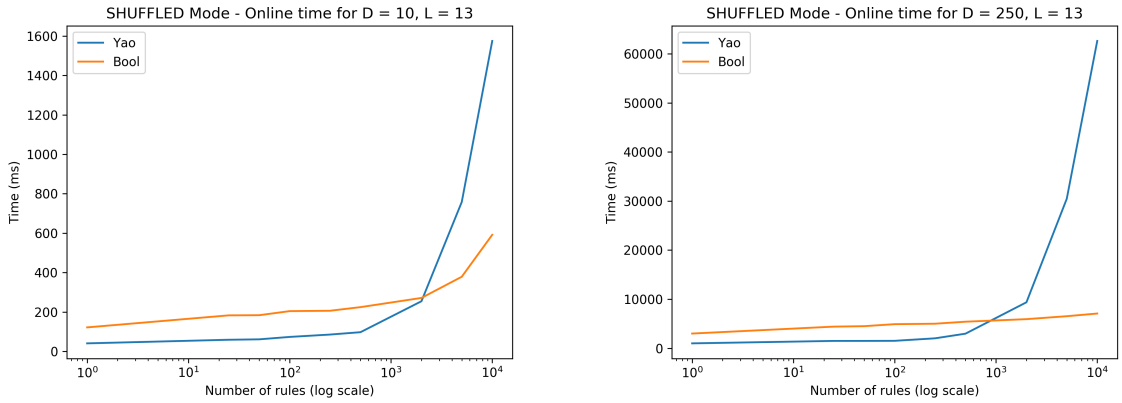
(b) High delay (250 ms)

Figure 7.7: Online time for the UNSHUFFLED mode, varying number of rules

Figure 7.7 shows the execution times for the Yao circuit (in blue) and the boolean circuit (in orange) in UNSHUFFLED mode. They exhibit a behavior similar to the CLEAR circuit, as the circuit using Yao scales linearly in the number of rules and the boolean circuit being mostly constant.

A notable difference is that the point at which Yao become slower than the boolean circuit is lower, because the size of the input is increased by the mapped data and, as we discussed previously, the efficiency of Yao's protocol scales linearly with the size of the input.

Another difference is that, for very large number of rules, the boolean circuit starts deviating from the expected behavior and is no longer constant, as the overhead caused by many parallel operations becomes non-negligible. If the communication delay increases, more rule and/or longer rules can be processed without exhibiting this behavior.



(a) Low delay (10 ms)

(b) High delay (250 ms)

Figure 7.8: Online time for the SHUFFLED mode, varying number of rules

Figure 7.8 shows the results when we add the shuffle grid. We observe that, similarly to the COLLAPSED circuit, the online time in SHUFFLED mode scales logarithmically with the number of rules, i.e. the depth of the grid.

Because the overhead caused by the large number of parallel operations in the boolean circuit is non-negligible, it affects the cutoff point at which the boolean circuit becomes more efficient than Yao. Consequently, the aforementioned cutoff point is no longer independent from the communication delay.

Circuit	Num. rules	Setup	Online	Rules
Yao	2000	5.2 MiB	131 MiB	112 kiB
Boolean	2000	54 Mib	852 kiB	112 kiB
Boolean	10000	330 Mib	5.2 Mib	560 kiB

Table 7.5: Bytes exchanged for computation (54 Bytes rules, SHUFFLED mode)

Table 7.5 shows the number of bytes exchanged during the computation. An important information that appears is that the information transferred does not scale linearly to the size of the input, which is expected as the number of gates in the shuffle grid is proportional to $N * log(N)$.

		Rules [bytes : number]								
Delay		13:50	13:500	13:5000	37:50	37:500	37:5000	54:50	54:500	54:5000
		Yao								
1 ms	setup	9.158	47.7	474.4	14.51	115.5	1163	18.95	163.7	1642
	online	8.721	30.57	281	9.671	69.43	660.2	11.72	97.48	926.1
10 ms	setup	41.42	96.29	536.7	47.27	172.6	1301	63.72	224.9	1839
	online	58.92	90.41	400.8	60.39	136.3	952.3	74.39	172	1352
100 ms	setup	400.8	810.4	4067	405.8	1407	10150	601.5	1825	14500
	online	598.7	990.1	5088	600.4	1571	12850	793.9	2157	18310
		Boolean								
1 ms	setup	8.669	36.6	314.7	13.75	90.14	823.6	17.5	125.4	1193
	online	14.3	17.6	62.57	16.73	20.53	66.83	16.88	20.87	77.24
10 ms	setup	42.82	100.9	367.5	65.63	147.3	893.9	69.01	184.5	1271
	online	122.5	125.9	175.7	143.1	146.7	211.4	143.3	147.3	231.6
100 ms	setup	310.6	813.8	2423	605.3	1101	5427	608.9	1265	7621
	online	1202	1206	1257	1403	1407	1552	1403	1408	1563

Table 7.6: Online and setup time (in milliseconds) for UNSHUFFLED mode, with various parameters

		Rules [bytes : number]								
Delay		13:50	13:500	13:5000	37:50	37:500	37:5000	54:50	54:500	54:5000
		Yao								
1 ms	setup	10.27	80.69	1283	17.17	148.3	1955	22.05	195.6	2446
	online	8.1	45.34	648.2	10.88	84.91	1026	13.22	113.2	1287
10 ms	setup	45.71	121.1	1367	62.81	197.7	2086	68.62	250.9	2612
	online	61.29	97.52	758	74.53	153	1312	77.45	196.8	1698
100 ms	setup	404.9	1015	11270	406.2	1623	17320	606.2	2029	21710
	online	601.2	1178	11910	599.3	1955	19680	797.2	2531	25100
		Boolean								
1 ms	setup	10.94	64.38	927.7	15.89	116.2	1440	19.88	152.5	1804
	online	21.4	30.79	138	23.98	33.68	147.6	24.13	33.91	157.5
10 ms	setup	45.52	126.3	974.6	67.57	175.1	1556	79.72	210.4	1913
	online	183.7	224.6	378.8	204.5	244.6	411.9	204.6	244.7	437.4
100 ms	setup	405.4	1013	6026	607.5	1243	9038	610.8	1452	11220
	online	1803	2119	2646	2004	2331	2934	2004	2331	2949

Table 7.7: Online and setup time (in milliseconds) for SHUFFLED mode, with various parameters

7.3 Estimates for cycle detection

It is possible to get an estimation of the total time for verifying, with our SMPC-based protocol, that a rule does not create forwarding cycles. We base our estimation on the messages exchanged for performing the protocol, which are represented in Figure 7.9.

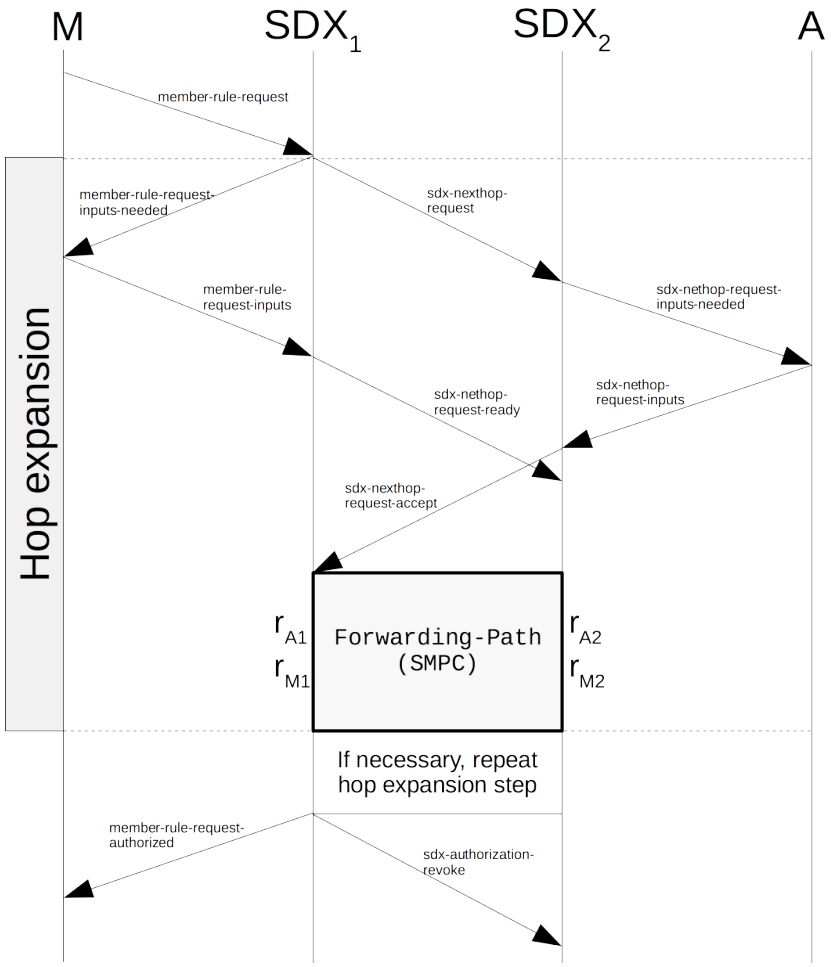


Figure 7.9: Messages exchanged for the cycle detection protocol

Let D_s be the delay between multiple SDXes, and D_m the delay between an SDX and its members. We assume $D_s > D_m$.

We also assume that requests are done in parallel and not interfering with each other, based on the observation that most of the computation time is caused by network communication delay. Similarly, we ignore the time required for local operations outside of SMPC, which is supposed to be negligible compared to network latency (for example, filtering local rules based on public bits takes around 1 ms for 5000 rules).

Since requests for rule verification between SDXes are made in parallel, the number of necessary consecutive, non-parallel exchanges (i.e. Hop Expansions in Figure 7.9) is equal to 1 + the number of consecutive deflections the packets in a flow might encounter.

When a request to install a rule is sent by a member, the SDX verifies all ASes found in the path. If there is no SDX in the path, the installation can proceed, otherwise it must make a request to all the SDXes receiving deflected traffic. We call this phase the *hop expansion*: each request discovers the next set of SDX members receiving deflected traffic.

The critical path for the whole protocol requires the exchange of the following messages:

1. The member requests the installation of a rule and sends the AS path along with the public bits.

2. The SDX finds all SDXes in the path and use them as initial nodes in the deflection graph. It performs multiple *hop expansions* until all nodes have been explored, and the last hop expansion did not return any new AS receiving deflected traffic.

3. The SDX sends the authorization to its member.

The revocation tokens need to be sent when a member installs a new rule. However, a member can install its rule during the time it takes for these tokens to travel. As such, we do not consider the delay before which tokens are received to be part of the critical path of the verification.

It is important to note that this creates a period equal to the travel time of revocation tokens during which loops are possible. Because the delay for sending the token is necessarily lower than the delay to go around the forwarding loop, packets are only at risk of looping at most once.

Alternatively, an AS might wait until it receives confirmation that the conflicting rule was removed before installing the new one. This adds a delay equal to the round-trip time between ASes to the rule installation.

The critical path for the hop expansion (which starts after the member already made the first request to its SDX) is the following:

1. SDX_1 sends SDX_2 a request to set up an SMPC exchange.

2. SDX_2 asks its members for their inputs shares.

3. A filters its rules based on the public bits and sends the rules that went through the filter to SDX_2.

4. SDX_2 sends the encrypted inputs to SDX_1.

5. If the public bits did not filter out all the rules, SDX_1 and SDX_2 proceed with the SMPC exchange.

The protocol includes another path with messages exchanged between SDX_1, M, SDX_1 and SDX_2, but assuming the delay between member and SDX is low, this should never be the critical path.

It is important to point out that, should the public bits filter out all installed rules, the SMPC exchange can be avoided.

Using the critical path described above, the total time required for computation, assuming there is at least one SDX in the AS path and the public bits were not sufficient to filter out all rules at any SDX, is equal to

$$2D_m + (1 + C) * (2D_m + 2D_s + SMPC)$$

with C being the maximal number of deflections a packet from this flow can encounter.

We assume that $D_m = 10$ ms, $D_s = 100$ ms and $N = 500$, with no optimization using public bits or outsourcing. This is a conservative estimate, as the delay between SDXes is expected to be generally lower than 100 milliseconds. If this is not the case (such as intercontinental traffic exchange), an SDX might decide to set up a single server on the other continent to handle the network verification with lower delays, especially in regards to the SMPC part of the computation.

- Best case: In the best case, there is no SDX in the AS path and no hop expansion is required. In this case, the total time is $2D_m = 20ms$.

- Expected case: The expected case is that there is one or more SDXes in the path, but none of them creates a deflection for this flow. In this case, the total time is $4D_m + 2D_s + SMPC = 240ms + 1178ms = 1418ms$.
- Worst case: We consider that in the worst case, $C = 2$ since a path with more than two deflections is unlikely to be the optimal choice (though the protocol does not enforce any hard limit). In this case, the total time is $8D_m + 6D_s + 3SMPC = 4214ms = 4.2s$. Should we use SIDR instead, this rule would simply be refused.

Chapter 8

Future work

We proposed a new type of privacy-preserving network verification primitive, with a concrete solution to the cycle detection problem in SDN-enabled networks. In this chapter, we discuss further improvements to our solution, as well as alternative approaches for solving similar problems.

8.1 Full implementation and testing

Our Python prototype of the protocol can be used for local offline simulations. The next step is to turn it into a functional software. The solution would need to be reliable and to fully exploit asynchronous and parallel exchanges.

To make it functional, we would also need to further develop the SMPC circuit and ABY. It is important for efficiency that the SDXes be able to use pre-established connections, use that connection to perform multiple requests, and pre-compute the setup phase.

Once a functional implementation is available, it will be possible to run better simulations and test the efficiency of our solution on large-scale, real world data.

This implementation would also provide us with better insight for additional improvements in terms of performance and security. It might be possible to create more efficient circuits, e.g. using Lookup Tables [18] and alternative SMPC solutions. It is also possible to modify the SMPC protocol in order to provide security against malicious adversaries as discussed in [34].

8.2 Intel SGX

The SMPC model is defined as a method of computation that ensures the complete privacy guarantees if the set of members do not collude.

Recent advancements in cryptography and hardware-based proofs have opened the possibility of using the Trusted Execution Environment model, which relies on processor guarantees to provide complete privacy guarantees [15]. The Trusted Execution Environment model is briefly explained in Section 2.4.1.

Through these techniques ,we could use the Intel SGX technology to solve the privacy-preserving network verification with lower latency constraints and possibly a better flexibility. Using this approach, the confidential rules would be securely distributed between controlled enclaves and accessible only to the trusted software.

While the Trusted Execution Environment model is very powerful and flexible, its implementation is not straightforward and comes with practical drawbacks. An Intel SGX implementation of the cycle detection protocol would first require answering some questions about the feasibility and security of that solution.

1. Is it reasonable to expect everyone to run an SGX enclave?

It requires a specific hardware and removes flexibility in software, preventing custom implementations. This constraint might be problematic when every ASes and/or SDXes using SDN is expected to participate.

2. What are the performance costs? Which symmetric and asymmetric functions are being used, and what latency or bandwidth limitation does it create?

3. Is the solution to this problem vulnerable to side-channel attacks?

   A known weakness of Intel SGX is its vulnerability to certain kinds of attacks, such as those based on memory access patterns [16]. A solution for cycle detection would need to be resilient to this category of attack, including those based on the observing information exchange between enclaves.

4. If one enclave is compromised, how much does it affect the overall network privacy?

   The protocol implies cooperation between many ASes and SDXes. Those are often large and resourceful entities, so it is unsafe to assume than none of them would be able to extract the keys from a SGX processor. In this case, how much information would be leaked from the other members in the network?

8.3 Other network verification problems

Our research is currently limited to a specific type of network verification problems with a focus on forwarding loops. Because we assume an underlying honest BGP network, we can ignore some other problems (such as blackholes and non-SDN loops) and focus on those that are caused by adding SDN rules that are congruent with BGP.

However, the primitive we designed has the potential to be used to solve a wider set of problems that just this one.

One possible direction would be to verify the absence of accidental blackholes in the network. Blackholing is a technique that can be used to counter certain kinds of attacks, but it is a problem if legitimate network packets are dropped. Coincidentally, the primitive could also be used to securely share information about DDoS attacks with other network operators, discovering the source of an attack with the cooperation of multiple domains.

Another possible application would be the detection of BGP prefix hijacking. SDN rules can be used to extend the ROAs (Route Origin Authorizations [1], the attestation that the origin AS number is authorized to announce the prefix), and forwarding configurations can be compared through SMPC to detect and correct errors in the control plane faster. This problem opens the question of the soundness of the threat model - in case of willingly harmful attack, the honest-but-curious model makes less sense and a more secure SMPC technique is required.

Part III

Conclusion

The traditional approach to inter-domain routing, BGP, and the novel approach, SDN, have fundamentally different expressiveness power.

BGP is used for its reliability and avoids revealing confidential information, qualities that led it to be the most widely deployed solution, while alternatives are rarely available.

SDN provides new capabilities for network operators. By creating a clear separation between the data plane and the control plane and exposing simple and centralized abstractions, these approaches greatly simplify network configuration. They also explicitly enable traffic engineering techniques that are difficult to achieve in classic networking, such as load balancing and application-specific peering.

However, this flexibility also comes with constraints. SDN policies inherently reveal a significant amount of information about confidential business strategies, which cause an SDN-based domain-level routing protocol to be impractical as a large amount of private information would be revealed.

Thus, an intermediate solution is to use an hybrid model mixing BGP and SDN policies. But because of the difference in expressiveness between those techniques, with SDN being much more fine-grained, that hybrid model can lead to inconsistencies in the control-plane configuration, which ultimately cause erroneous forwarding behavior and packet losses.

After analyzing the environment and problems in deploying Software-Defined Networking at inter-domain level, this thesis proposes two contributions:

- First, a privacy-preserving primitive, Distinct-Match, that allows network operators to compare their SDN rules with each other without revealing confidential information.

  We implemented this primitive using Secure Multi-Party Computation. The implementation proves the feasibility of our proposed solution, as corroborated by our evaluation and micro-benchmarks. Additionally, we also discuss optimizations and trade-offs that can be used by operators to speed up the verification.

- Second, a protocol for the verification of erroneous forwarding behavior caused by mixing BGP and SDN routing. This solution, based on the aforementioned primitive, solves a very practical problem for SDN deployment.

  Out protocol addresses the risk of creating forwarding loops when using SDN at the inter-domain level, lifting a drawback that prevents SDN from being adopted in inter-domain routing while providing better accuracy and stronger privacy guarantees than other existing solutions.

In addition to the concrete contributions, this work also aims to pave the way towards new techniques for privacy-preserving network verification and configuration. Because using SDN policies creates new privacy constraints, the classical solutions based on propagating local configuration information become impractical.

Our objective is to prove that using SMPC to compute functions over encrypted data is a feasible approach to inter-domain routing and provides greater flexibility and enhanced privacy for network configuration. We implemented a mechanism to solve an existing problem preventing the deployment of Software-Defined Networking for inter-domain routing. We also implemented a privacy-preserving primitive allowing operators to compare their SDN rules, that can help solve multiple kinds of problems, such as conflict detection and path discovery.

Part IV

References

Bibliography

[1] Apnic: Route origin authorizations (roa). URL: https://www.apnic.net/manage-ip/apnic-services/resource-certification/ROA/.

[2] Brazil builds internet cable to portugal to avoid nsa surveillance. URL: http://www.ibtimes.com/brazil-builds-internet-cable-portugal-avoid-nsa-surveillance-1717417.

[3] European internet exchange association's list of ixps. URL: https://www.euro-ix.net/ixps/list-ixps/.

[4] Intel secure guard extensions. URL: http://www.pdl.cmu.edu/SDI/2013/slides/rozas-SGX.pdf.

[5] Ryu SDN framework. URL: https://osrg.github.io/ryu/.

[6] Verizon ip latency statistics. URL: http://www.verizonenterprise.com/about/network/latency/.

[7] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *ACM SIGPLAN Notices*, volume 49, pages 113–126. ACM, 2014.

[8] Gilad Asharov, Daniel Demmler, Michael Schapira, Thomas Schneider, Gil Segev, Scott Shenker, and Michael Zohner. Privacy-preserving interdomain routing at internet scale. *Proceedings on Privacy Enhancing Technologies*, 3:1–21, 2017.

[9] Rüdiger Birkner, Arpit Gupta, Nick Feamster, and Laurent Vanbever. Sdx-based flexibility or internet correctness?: Pick two! In *Proceedings of the Symposium on SDN Research*, pages 1–7. ACM, 2017.

[10] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis. In *International Conference on Financial Cryptography and Data Security*, pages 57–64. Springer, 2012.

[11] Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 236–252. Springer, 2005.

[12] Marco Canini, Vojin Jovanovic, Daniele Venzano, Boris Spasojevic, Olivier Crameri, Dejan Kostic, et al. Toward online testing of federated and heterogeneous distributed systems. In *USENIX Annual Technical Conference*, 2011.

[13] Marco Chiesa, Daniel Demmler, Marco Canini, Michael Schapira, and Thomas Schneider. Towards securing internet exchange points against curious onlookers. In *Applied Networking Research Workshop (ANRW'16)*, 2016.

[14] Marco Chiesa, Christoph Dietzel, Gianni Antichi, Marc Bruyere, Ignacio Castro, Mitch Gusat, Thomas King, Andrew W Moore, Thanh Dang Nguyen, Philippe Owezarski, et al. Inter-domain networking innovation on steroids: empowering ixps with sdn capabilities. *IEEE Communications Magazine*, 54(10):102–108, 2016.

[15] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[16] Shaun Davenport and Richard Ford. Sgx: the good, the bad and the downright ugly. *Virus Bulletin*, 2014.

[17] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[18] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the communication barrier in secure computation using lookup tables. 2017.

[19] Christoph Dietzel, Mario Abt, Marco Chiesa, and Philippe Owezarski. *ENDEAVOUR: D4. 5: Implementation of the Selected Use Cases for the IXP Members*. PhD thesis, DE-CIX, 2017.

[20] Anne Edmundson, Roya Ensafi, Nick Feamster, and Jennifer Rexford. A first look into transnational routing detours. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 567–568. ACM, 2016.

[21] Lixin Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Transactions on Networking (ToN)*, 9(6):733–745, 2001.

[22] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

[23] Arpit Gupta, Robert MacDavid, Rüdiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Laurent Vanbever. An industrial-scale software defined internet exchange point. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 1–14. USENIX Association, 2016.

[24] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. Sdx: A software defined internet exchange. *ACM SIGCOMM Computer Communication Review*, 44(4):551–562, 2015.

[25] Debayan Gupta, Aaron Segal, Aurojit Panda, Gil Segev, Michael Schapira, Joan Feigenbaum, Jenifer Rexford, and Scott Shenker. A new approach to interdomain routing based on secure multi-party computation. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 37–42. ACM, 2012.

[26] Yoonseon Han, Jonghwan Hyun, and James Won-Ki Hong. Graph abstraction based virtual network management framework for sdn. In *Computer Communications Workshops (INFOCOM WKSHPS), 2016 IEEE Conference on*, pages 884–885. IEEE, 2016.

[27] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP@ ISCA*, page 11, 2013.

[28] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the one big switch abstraction in software-defined networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 13–24. ACM, 2013.

[29] Sukhveer Kaur, Japinder Singh, and Navtej Singh Ghumman. Network programmability using pox controller. In *ICCCS International Conference on Communication, Computing & Systems, IEEE*, number s 134, page 138, 2014.

[30] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *NSDI*, pages 99–111, 2013.

[31] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *NSDI*, volume 12, pages 113–126, 2012.

[32] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. Veriflow: Verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review*, 42(4):467–472, 2012.

[33] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. A first step towards leveraging commodity trusted execution environments for network applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 7. ACM, 2015.

[34] Yehuda Lindell, Benny Pinkas, and Nigel P Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *International Conference on Security and Cryptography for Networks*, pages 2–20. Springer, 2008.

[35] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[36] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on*, pages 1–6. IEEE, 2014.

[37] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology–CRYPTO 2012*, pages 681–700. Springer, 2012.

[38] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending networking into the virtualization layer. In *Hotnets*, 2009.

[39] Gordon D Plotkin, Nikolaj Bjørner, Nuno P Lopes, Andrey Rybalchenko, and George Varghese. Scaling network verification using symmetry and surgery. In *ACM SIGPLAN Notices*, volume 51, pages 69–83. ACM, 2016.

[40] Yakov Rekhter, Tony Li, and Susan Hares. A border gateway protocol 4 (bgp-4). Technical report, 2005.

[41] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. *arXiv preprint arXiv:1702.08719*, 2017.

[42] Matthias Wählisch, Olaf Maennel, and Thomas C Schmidt. Towards detecting bgp route hijacking using the rpki. *ACM SIGCOMM Computer Communication Review*, 42(4):103–104, 2012.

[43] Andrew C Yao. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.

[44] Mingchen Zhao, Wenchao Zhou, Alexander JT Gurney, Andreas Haeberlen, Micah Sherr, and Boon Thau Loo. Private and verifiable interdomain routing decisions. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 383–394. ACM, 2012.

[45] Mingchen Zhao, Wenchao Zhou, Alexander JT Gurney, Andreas Haeberlen, Micah Sherr, and Boon Thau Loo. Private and verifiable interdomain routing decisions. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 383–394. ACM, 2012.

Part V

Appendix

Appendix A

Using ABY to write SMPC applications

ABY [17] is a publicly available framework providing convenient interfaces for writing two-party SMPC applications. ABY also includes many recent optimizations to bring computations times down to practical levels. It also makes it easy to switch between two different SMPC approaches, using either the GMW protocol[22] or Yao's garbled circuits [43].

This appendix describes the creation of a SMPC application using ABY, with examples from the source code of our Distinct-Match implementation.

1. Create and setup the SMPC parties, and connect the client to a server.

   The argument `role` can be either `CLIENT` or `SERVER`. The address and port parameters are used to connect the client and server. The other parameters allow tuning the configuration for this exchange.

   Returns a `ABYParty` object that will be used to build and execute the SMPC circuit.

```
ABYParty* party = new ABYParty(role, address, port,
                seclvl, bitlen, nthreads,
                mt_alg, maxgates);
```

2. Create the circuit.

   The next step is to get a `Circuit` object appropriate for the type of sharing we use. `sharing` can be either `S_BOOL` for a GMW circuit, `S_YAO` for using Yao's garbled circuit, or `S_ARITH` for arithmetic circuits. We only use boolean and Yao circuits, because both their interfaces allow us to build boolean circuits.

```
vector<Sharing*>& sharings = party->GetSharings();
BooleanCircuit *circ = (BooleanCircuit*)
                sharings[sharing]->GetCircuitBuildRoutine();
```

3. Add input gates to create shares.

   The input gates take a value as parameter and return a share representing this value, i.e. the encrypted version. One party initializes the gates with the values stored in the `ptrn` and `mask` buffers, while the other party use dummy gates to indicate that they will receive a shared version of that value.

   The SIMD `in` gates (used by the server in this example) are used for optimization by storing multiple values in a single share. The `bitlen` parameter defines the number of bits in a share.

```
share *s_p1, *s_m1, *s_p2_u, *s_m2_u;

if (role == SERVER) {
        s_p1 = circ->PutSIMDINGate(nvals, ptrn, bitlen, SERVER);
        s_m1 = circ->PutSIMDINGate(nvals, mask, bitlen, SERVER);
        s_p2_u = circ->PutDummyINGate(bitlen);
        s_m2_u = circ->PutDummyINGate(bitlen);
} else if (role == CLIENT) {
        s_p1 = circ->PutDummySIMDINGate(nvals, bitlen);
        s_m1 = circ->PutDummySIMDINGate(nvals, bitlen);
        s_p2_u = circ->PutINGate(ptrn, bitlen, CLIENT);
        s_m2_u = circ->PutINGate(mask, bitlen, CLIENT);
}
```

4. Add the logic gates.

   Those gates define the function computed by the circuit. They take or more shares as input, and return a share containing the output. This means that the shares are used similarly to variables, except that their value can not be accessed directly because they are encrypted.

   For the computation to complete, both parties need to build the same circuit.

   In this example, we compute the function $(p_1 \oplus p_2) \wedge (m_1 \wedge m_2)$.

```
share *s_bothwatch = circ->PutANDGate(s_m1, s_m2);
share *s_different = circ->PutXORGate(s_p1, s_p2);
share *s_distinct = circ->PutANDGate(s_bothwatch, s_different);
```

5. Add output gates.

   The `out` gate takes an encrypted share as input and returns a readable share. To perform this decryption in GMW, each party sends its share to the other.

   By changing `ALL` in this example to `SERVER` or `CLIENT`, it is possible to only enable the decryption for one party and not the other.

```
share *s_out = circ->PutOUTGate(s_result, ALL);
```

6. Execute the circuit.

   All steps until now were performed locally, to build an internal representation of the circuit on the client and server. The `ExecCircuit` function tells ABY to start running SMPC and actually propagate the values through the circuit by exchanging encrypted information between the client and server.

```
party->ExecCircuit();
```

7. Read the reconstructed output.

   In the previous step we reconstructed `s_out`, whose value can now be read.

```
uint8_t *out_vals = s_out->get_clear_value();
```

Appendix B

Structure of messages for the SMPC Cycle Detection protocol

This appendix describes all the messages exchanged between ASes and SDXes during the cycle detection protocol, barring the messages related to the SMPC computation itself.

Figure 7.9 gives an illustration of how those messages are exchanged during the protocol, while this section describes the messages themselves.

```
<rule> =
{
    pattern: "00112233445566778899AABBCC",
    mask: "FFFFFFFFFFFFFFFFFFFFFFFFFF",
    length: 13
}
```

1. Member M requesting installation of a new rule to SDX1. It sends the AS path and the public bits of that rule.

```
{
    type: "member-rule-request",
    data: {
        aspath: [Z, B, A, R, Q],
        public_bits: <rule>
    }
}
```

2. SDX1 finds the others SDXes in the path (SDX2 for A, B and SDX3 for Q, R). It sends the keys of the SDXes in the path (it is assumed that M authenticates that those keys are correct), as well as the list of tokens that M is allowed to revoke.

```
{
    type: "member-rule-request-inputs-needed",
    data: {
        keys: {
            "sdx2": pk_sdx2,
            "sdx3": pk_sdx3
        },
        revocable: {
            "token1": <public_bits>,
            "token2": <public_bits>,
            "token3": <public_bits>
        }
    }
}
```

3. Member M sends to SDX1 the shares of its inputs. For each rule, one of the shares is encrypted with the public key of the other SDX. M also sends all the tokens that need to be revoked based on their public bits.

```
{
    type: "member-rule-request-inputs",
    data: {
        public_bits: <rule>,
        local_shares: {
            "sdx2": <ruleM_share02>,
            "sdx3": <ruleM_share03>
        },
        remote_shares: {
            "sdx2": Enc(pk_sdx2, <ruleM_share12>),
            "sdx3": Enc(pk_sdx3, <ruleM_share13>)
        },
        revoke: [token1, token2]
    }
}
```

4. SDX1 sends a rule comparison request to SDX2, along with the encrypted identifier of the members that need to be checked and a revocation token. By sending the token early, we ensure that SDX2 can revoke it at any time after it has allowed the rule, even if SDX1 did not complete the verification yet.

```
{
    type: "sdx-nexthop-request",
    data: {
        public_bits: <rule>,
        revoke_token: 123456789,
        members: [Enc(pk_sdx2, A), Enc(pk_sdx2, B)]
    }
}
```

5. SDX2 requests the input rules to its members (filtered to remove those that match the public bits).

```
{
    type: "sdx-nexthop-request-inputs-needed",
    data: {
        public_bits: <rule>
        key: pk_sdx1
    }
}
```

6. Members A and B send their encrypted inputs to SDX2.

```
{
    type: "sdx-nexthop-request-inputs",
    data: {
        local_share: <rulesA_share0>,
        remote_share: Enc(pk_sdx1, <rulesA_share1>)
    }
}
```

7. SDX2 confirms that it is ready to start the SMPC exchange and sends to SDX1 its share of the inputs.

```
{
    type: "sdx-nexthop-request-accept",
```

```
    data: {
        num_rules: 999,
        inputs: [
            Enc(pk_sdx1, <ruleA_share1>),
            Enc(pk_sdx1, <ruleB_share1>)
        ]
    }
}
```

8. SDX1 confirms that it is ready to start the SMPC exchange and sends to SDX2 its share of the inputs.

```
{
    type: "sdx-nexthop-request-ready",
    data: {
        inputs: [
            Enc(pk_sdx2, <ruleM_share12>)
        ]
    }
}
```

9. After the previous steps have been repeated with all relevant SDXes, SDX1 sends the confirmation of authorization to M.

```
{
    type: "member-rule-request-authorized",
    data: {
        revoke_token: 123456789,
        nexthops: {
            sdx2: [Enc(pk_sdx2, A), Enc(pk_sdx2, B)],
            sdx3: [Enc(pk_sdx3, Q), Enc(pk_sdx3, R)],
        }
    }
}
```

10. If at a later point A or B (resp. Q or R) installs a new rule that matches the public bits, SDX2 (resp. SDX3) revokes the authorization given previously.

```
{
    type: "sdx-authorization-revoke",
    data: {
        revoke_token: 123456789
    }
}
```

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**