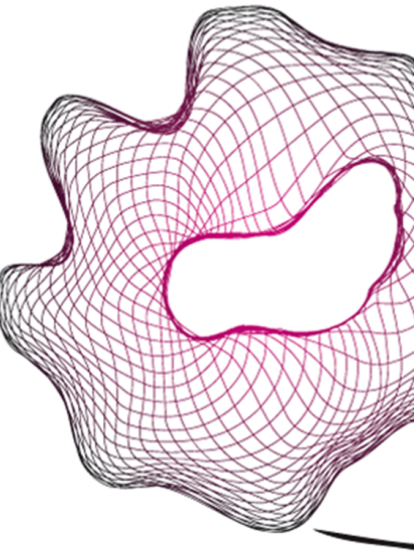
UNIVERSITY OF TWENTE.

**Faculty of Electrical Engineering,
Mathematics & Computer Science**



Accelerating the SCION IP Gateway using programmable data planes

Max C. Kellaway

M.Sc. Thesis
September 2022

Examiners:
prof. dr. ir. C. E. W. Hesselman (UT)
prof. dr. ir. R. M. van Rijswijk-Deij (UT)
dr. ing. F. W. Hahn (UT)

Advisors:
dr. A. Abhishta (UT)
dr. ing. R. Koning (SIDN Labs)
ir. C. Schutijser (SIDN Labs)

DACS Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

SIDN LABS

Abstract

This work focuses on performance accelerating the SCION IP Gateway (SIG) using Open Programmable Data Plane (OPDP) technologies. SCION is a new clean slate Internet protocol with the goal of increasing scalability, control, transparency and security. SCION achieves these goals primarily by encoding the packets' path inside the packet header and cryptographically signing and verifying this path. The SIG is used to enable communication between IP-based hosts and SCION-based hosts. The SIG needs to be performance optimised since many IP-based end-hosts will send their traffic to the SIG, resulting in traffic aggregation at the SIG due to the many-to-one connection.

An OPDP is an open-source data plane written in code resulting in transparent data plane behaviour. We define three possible spaces for OPDPs, the hardware space, the kernel space and the user space. Each space brings along a variety of possible programming languages. For the hardware space, we consider P4 with the Intel Tofino architecture. We consider eBPF an in-kernel virtual machine for custom in-kernel code execution with the eXpress Data Path (XDP) and Traffic Control (TC) networking hooks for the kernel space. The current open source SIG implementation is a user space OPDP written in Go.

We have performed a literature study on the performance of OPDPs and SCION. To the best of our knowledge, at the time of writing, we are the first to perform measurements on the SIG performance and to accelerate the SIG since we did not find any literature on the performance or acceleration attempts of the SIG.

To determine which OPDP is best suited for the acceleration, we tried to develop prototypes for a P4-based (hardware space), XDP-based (kernel space) and TC-based (kernel space) SIG. The TC-based SIG implementation was the only functioning prototype. The P4-based SIG could not perform the required reassembly operations. An XDP-based OPDP was incompatible since XDP programs could not be connected to the ingress side of interfaces, a requirement for correctly redirecting the packets.

We have created an automated virtualised test setup for performance measurements on SIGs. We perform the measurements on the user space SIG and our TC-based SIG. We analyse RTT to determine the processing delay caused by the

SIG and analyse throughput to assess the maximum forwarding speed.

Our analysis shows a 64% throughput increase whilst reducing the average latency by 45% and the CPU utilisation by 99% for the TC-based SIG compared to the user space SIG.

Therefore, we show that accelerating the SIG is possible using a TC-based OPDP resulting in higher throughput and lower RTT whilst reducing the CPU load. However, there still is room for improvement of the TC-based SIG supported by our measurement results and future work ideas.

Acknowledgements

This thesis and research would not have been possible without the help of some organisations and people.

Firstly, I would like to thank SIDN Labs for supporting me and making the research possible. I want to thank Ralph and Caspar specifically for the daily supervision from the perspective of SIDN Labs. Cristian Hesselman is the chair of my committee, who helped guide the research direction during our meetings which I am grateful for. Thank you to Roland van Rijswijk-Deij, Florian Hahn and Abhishta Abhista for participating in my graduation committee.

I would also like to thank Ronald van der Pol from SURF as the subject matter expert which whom I had the email contact concerning the OPDP implementations and SIG requirements. Besides Ronald, I also want to thank Intel for reviewing my thesis concerning their Tofino switch.

Contents

Appendices

Chapter 1

Introduction

In this chapter, we give an introduction to Internet Architectures (sec. 1.1) and introduce the concept of an Open Programmable Data Plane (OPDP) (sec. 1.2). Next, we discuss our problem statement (sec. 1.3), the accompanying research questions (sec. 1.4) and our contributions to science (sec. 1.5). Then, we explain where to find our open-source work (sec. 1.7). We conclude this chapter by discussing the approach we used for our research (sec. 1.6) and the different chapters our thesis consists of (sec. 1.8).

1.1 Internet Architectures

The current generation Internet consists of inter-connected Autonomous Systems (ASes), with Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6) used as addressing schemes. Border Gateway Protocol (BGP) is the routing protocol used in the Internet and distributes the available routes between the AS. BGP is an inter-domain routing protocol for routing between ASes whereas an intra-domain routing protocol would be used for routing within an AS.

The current Internet is not perfect, resulting in researchers looking into possible replacements for the Internet. Multiple clean slate Internet architectures are developed like NDN [1], RINA [2] and Scalability, Control, and Isolation On Next-Generation Networks (SCION) [3], to name a few. Clean slate Internet architectures have the advantage that they can be designed with the experience gained from the current Internet, without the limitations of the existing Internet infrastructure. The downside of clean slate architectures is the need for transition technologies like the SCION IP Gateway (SIG) to achieve the deployment of the clean slate architecture. Our research focuses on SCION as the replacement for the current Internet architecture. SCION is designed from the ground up with the goal of increasing scalability, control, transparency and security.

New clean slate Internet Architectures cannot be trivially deployed to the Internet due to protocol incompatibilities. A transition technology is required to translate between these two protocols. For SCION, this translation technology is called the SIG [4]. The SIG lets legacy hosts communicate with each other over a secure SCION backbone, enabling coexistence. The SIG devices will be deployed at the edges between the current Internet Protocol (IP)-based and SCION network.

1.2 Open Programmable Data Planes (OPDP)

The data plane is responsible for packet processing and forwarding operations. The data plane is sometimes referred to as the forwarding plane. We expand the data plane definition with the addition of an Open Programmable Data Plane (OPDP). An OPDP is a reprogrammable data plane defined using open-source code.

OPDPs are important since they increase the transparency of the network. The open-source OPDP code defines the functionality and behaviour of the data plane resulting in total transparency with regard to the forwarding operations. Furthermore, OPDPs increase the flexibility of the network since the data planes' behaviour is reprogrammable.

An OPDP can be implemented using hardware or software. There is a distinction between kernel space and user space for software implementations. The different implementation options result in hardware-based, kernel-based or user space-based OPDP types.

OPDPs need to be programmed through a programming language such as P4 or eBPF. P4 is a domain-specific language made for hardware space implementations. eBPF is a technology used for kernel space implementations. eBPF supports two networking hooks to attach programs to eXpress Data Path (XDP) and Traffic Control (TC), where XDP programs execute before TC programs. XDP, however, is more restricted in functionality than TC.

1.3 Problem Statement

Suppose the SIG is deployed in a large-scale production environment. In that case, the packet processing throughput should be highly optimised since multiple IP-based end-hosts will send their packets to the SIG to be forwarded using SCION as secure backbone [5]. The aggregation of packets at the SIG makes the SIG a potential bottleneck in the packet forwarding plane.

The current open-source implementation of the SIG utilises a user space-based OPDP written in Go. The implementation does not utilise state-of-the-art OPDP

technologies, such as eBPF or P4, to accelerate network processing, probably resulting in sub-optimal performance and efficiency. There is a closed-source performance optimised implementation by Anapaya [6], but it is not an OPDP; therefore, we will not consider it in our work.

Furthermore, optimising the SIG is not a straightforward task. There are at present multiple promising state-of-the-art technologies for OPDPs (e.g., p4 and XDP), but it is unclear which technology best suits the job of accelerating the SIG. The uncertainty is due to the fact that limited functionality of a new technology could result in the SIG not being implementable with the specific technology, e.g. a P4 hardware constraint like an insufficient parsing depth. Even if the SIG can be implemented through an accelerated OPDP, the achieved performance gain with a specific technology is also uncertain.

To conclude, the problem that this thesis addresses is that we currently do not sufficiently understand which state-of-the-art OPDP technologies are most suitable and efficient to speed up the performance of the SIG.

1.4 Goal and Research Questions

The high-level goal of our research is to aid the implementation and deployment of a SCION-based Internet. This contribution is achieved by researching possible performance optimisations of the SIG using state-of-the-art OPDPs. However, the lessons learned should not be limited to the SIG framing protocol or specific OPDPs. An implementation of the SIG is already available for commodity hardware using a CPU [7] written in Go. However, we expect that there are performance and efficiency gains to be made when using state-of-the-art OPDPs like P4 switches or eBPF. We expect these gains since the OPDPs are designed to perform packet forwarding compared to the CPU's general execution of tasks.

We formulate the main research question and divide it into sub-questions to achieve the abovementioned goal.

RQ1 To what extent can state-of-the-art OPDPs be used to accelerate the performance of a SIG?

RQ2 What test setup do we require to perform the performance measurements on the SIG and OPDP SIG?

RQ3 Which OPDP technology is best suited to improve the SIG performance?

RQ4 What performance gains can be achieved using state-of-the-art OPDPs?

1.5 Contributions

Our research contributes to society and science in various ways. Our innovations are:

- An open-source performance accelerated SIG prototype, based on eBPF TC.
- A detailed performance analysis of the SIG and an TC OPDP-based SIG.
- An automated, reproducible measurement setup for performance measurements on SIG implementations.
- Recommendations for protocol creators and OPDP developers on points to consider for future Internet Architectures and the implementations thereof.

1.6 Approach

We start our research with a literature study to understand the working of SCION and the SIG and to map the current state-of-the-art OPDPs. Using the literature found, we also get an indication of the baseline performance of SCION.

Next, we specify the measurement methodology. Then we create a test setup to perform a baseline measurement of the performance specific to our testbed and to be able to evaluate our design. The baseline performance measurement helps us determine current bottlenecks in the SIG and we use it to evaluate our design once implemented.

After we perform the baseline measurements, we implement the OPDP-based versions of the SIG. We analyse multiple OPDPs to determine which is best suited for accelerating the SIG performance. A Go (user space) implementation of the SIG already exists [7]. Therefore we create a P4 (hardware space) and an eBPF (kernel space) implementation.

Once we have implemented the designs, we use the same measurement methodology as the baseline measurement to evaluate the OPDP performance. Finally, we compare the OPDP SIG implementations against the existing user space SIG and analyse the effectiveness of our performance acceleration.

1.7 Open-source Code and Data

We mention in sec. 1.5 that we have created an open-source TC-based SIG. The link to the repository containing the implementation is [1]. The repository contains our

[1] `https://github.com/SIDN/opdp-sig`

TC-based and XDP-based implementations, the scripts for the automated virtualised measurement testbed and the raw data of our experiments. The readme in the repository explains the repository structure in more detail.

1.8 Thesis Outline

In chapter 2, we give a more in-depth explanation of the workings of SCION and, more importantly, the working of the SIG. We explain the working of different OPDP types and their programming languages in chapter 3. In chapter 4, as a result of our literature study, we discuss the research performed by others on the SIG and OPDP performance.

Chapter 5 describes the data plane requirements of the SIG as well as our OPDP-based SIG implementations. In chapter 6, we explain the measurement methodology and testbed we use for performance measurements on OPDP-based SIGs. Next, we discuss the results of the experiments of a TC-based SIG in chapter 7. Chapter 8 discusses the possible improvements for our measurements, prototype and the SIG framing protocol. Finally, we conclude our thesis and provide recommendations and future research directions in chapter 9.

Chapter 2

The SCION Internet Architecture

The first major technology in our work is SCION. In this chapter, we briefly introduce how a SCION-based Internet works (sec. 2.1 to 2.3) as well as the component that we will focus on in the rest of this thesis: the SCION IP Gateway (SIG) (sec. 2.4). However, we cannot cover all SCION details within this master thesis. For this, we would like to recommend the official SCION book [5].

2.1 SCION Goals

Scalability, Control, and Isolation On Next-Generation Networks (SCION) is a clean slate Internet architecture developed by researchers at ETH Zurich. SCION aims to solve the problems found in the current Internet (e.g., BGP hijacking [8]) by providing a new design that is not limited by the current situation. SCION has the following (new) high-level objectives: availability in the presence of distributed adversaries, transparency and control (over trust roots), efficiency, scalability, extensibility, support for global but heterogeneous trust, deployability and foundation for other architectures [3]. These objectives are met by providing secure inter-domain routing and path-aware networking.

In contrast to most other clean slate Internet architectures such as RINA and NDN, SCION is at quite an advanced stage in development. SCIONLab, the global SCION research network proves the advanced stage (more on SCIONLab in sec. 2.3).

2.2 SCION Key Concepts

For SCION to achieve the above-mentioned goals, they introduced the concept of Isolation Domains (ISDs). An ISD is a logical grouping of Autonomous Systems (ASes) enabling scoped trust. This grouping can be performed based on different criteria, e.g. political, geographical, or business criteria. A smaller subset of ASes

from the ISD form the ISD core. Within this core, the ASes agree on a so-called Trust Root Configuration (TRC). The TRC defines the roots of trust and validates names, public keys and addresses. An AS can be in multiple ISDs. Fig. 2.1 gives an overview of how ASes can be grouped in ISDs. Each coloured circle illustrates an ISD with the light grey circle forming the ISD core. The white circles are the ASes, and links between the ASes are grey.

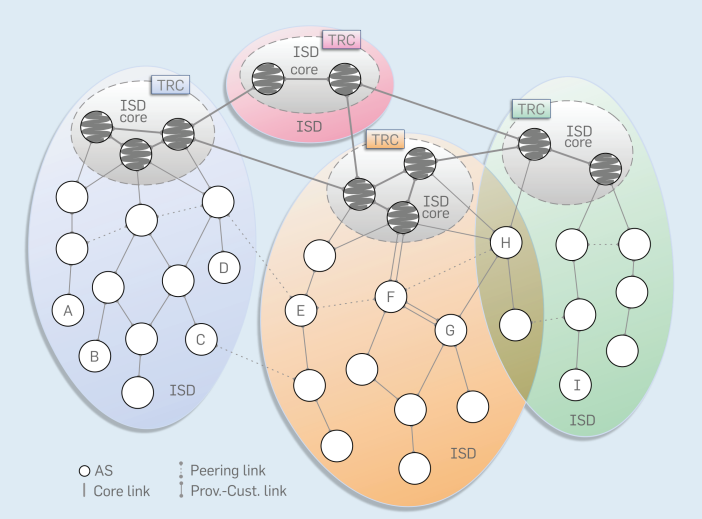


Figure 2.1: Overview of possible AS groupings within SCION ISDs. [3]

Path discovery is the process in SCION where the AS determines the available paths by listening to beaconing messages advertising paths. SCION defines three types of connections between ASes: core links, customer-provider links and peering links. Core links are between ASes in an ISD core. Customer-provider links are between down and upstream ASes. ASes can also share a link without path discovery, called a peering link. This link is only available for the traffic of the two connected ASes. All three link types are allowed to cross ISD boundaries.

SCION achieves complete separation between control and data plane operations using Packet Carried Forwarding State (PCFS). PCFS enables the separation since the data plane has access to the required routing data since it is stored in the packet header. The header of every packet stores the required path information. Hop Fields (HFs) in the packet's header store every AS hop. The separation of the two planes is desirable for multiple reasons. It increases resiliency against adversaries since the data plane can keep forwarding messages if the control plane is attacked. Furthermore, the separation of planes results in increased transparency due to the separation of concerns narrowing down the responsibility of a component.

"SCIONs PCFS" is beneficial for more reasons than separating the control and data plane. It enables path control for the user to choose the path that the packet takes on an AS level. This is a crucial difference compared to the working of the

current Internet and a big step forward for the security, transparency and controllability properties. Currently, the inter-domain routing tables inside routers are filled using the BGP protocol. The router looks up the egress port using the destination IP address in the routing table to determine how to route the packet. The PCFS eliminates the need for inter-domain routing table lookup because the forwarding path is in the packets themselves, speeding up the routing performance and decreasing the hardware requirements since the size of the current routing tables for the Internet is increasingly problematic [9]. SCION ensures authenticity by using a cryptographically signed field the Message Authentication Code (MAC), for every HF.

2.3 SCION Deployment

To deploy SCION within an AS, multiple server components are necessary [5].

- A beacon server to handle the path discovery messages (Path-segment Construction Beacons (PCBs)).
- A path server that stores the available paths and responds to client path queries.
- A name server that provides translation between human-readable names and (ISD, AS, address) triples that clients need to query paths.
- A certificate that server manages the AS certificates, keys, and caches the TRC.
- A border router for packet forwarding, be it to the next border router or an end-host inside the AS.

At least one of each server type is necessary to operate a SCION AS, but an operator can deploy multiple servers of each type for performance/scaling considerations. An example of the structure inside an AS can be seen in fig. 2.2.

Once an AS is SCION-enabled, the next step is to make hosts SCION aware. Fig. 2.3 gives an overview of how hosts interact with SCION. The SCION daemon (purple box) and dispatcher (red box) are the components that enable a host to communicate SCION natively. The daemon handles the control plane messages from the SCION control service (dark grey) and provides an Application Programming Interface (API) for applications (orange box) and libraries to interact with the control plane, e.g. for path lookup requests. The dispatcher is responsible for handling the incoming SCION messages from the SCION border router (green box) and distributing them to SCION native applications via a UNIX socket. The SCION book [5]

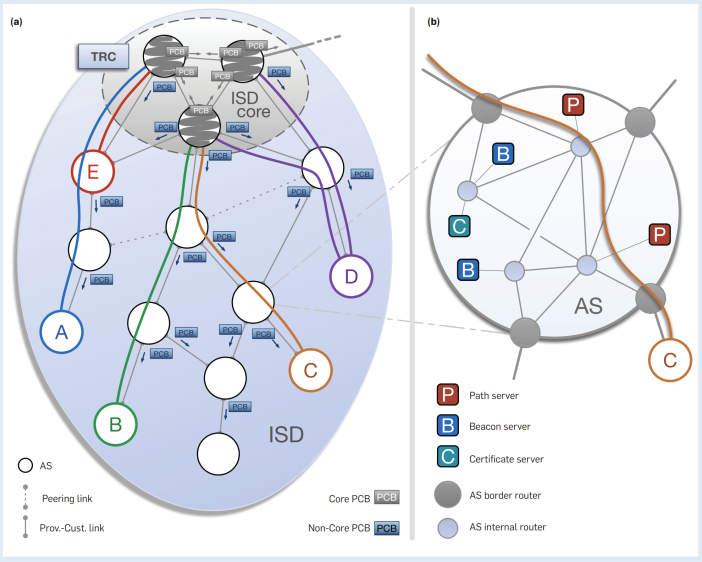


Figure 2.2: In-depth SCION architecture example. On the left an ISD is shown with the PCB propagation between ASes. On the right the inside of an AS with the required server components is shown. [3]

mentions that the dispatcher functionality in the future will probably be transferred to inside the kernel due to performance considerations.

Where possible, SCION gives freedom to the operators of an AS. Examples of this are the option to choose the address format used for hosts within an AS, the option to choose the used encryption algorithm for the MAC and the option to choose the intra-domain routing protocol. The protocol developers think this freedom could help in the adoption of the protocol since the operators do not have to agree on all the implementation details [5].

SCION offers built-in support for extensions. Every Packet Is Checked (EPIC) and Collaborative lightweight inter-domain bandwidth-reservation infrastructure (COLIBRI) are the two currently available extensions enabling increased packet authentication (EPIC) and path bandwidth/Quality of Service (QoS) reservations (COLIBRI). These extensions are out of scope for this study. Therefore, we will not explain them in more depth.

SCIONLab [11] is a global research network for developing and testing the protocol. Stichting Internet Domeinregistratie Nederland (SIDN) Labs operates one AS inside this network, the topology of SCIONLab can be seen in fig. 2.4. We could use the SCIONLab network to perform tests for this research. Anapaya [6], an ETH Zurich spinoff company, created and runs the first production SCION network. This network includes Internet Service Providers (ISPs), banks and research institutes in Switzerland.

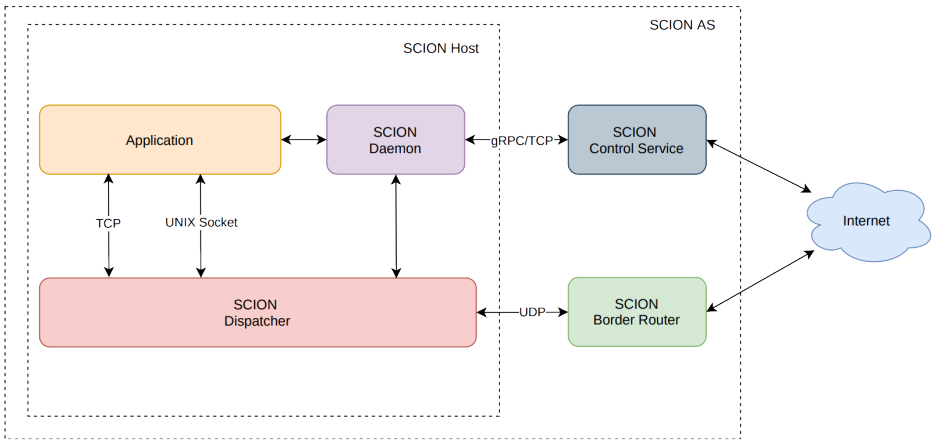


Figure 2.3: Overview of the SCION components enabling native host communication. The SCION daemon and dispatcher run inside the host whereas the border router and control service are deployed within the AS. [10]

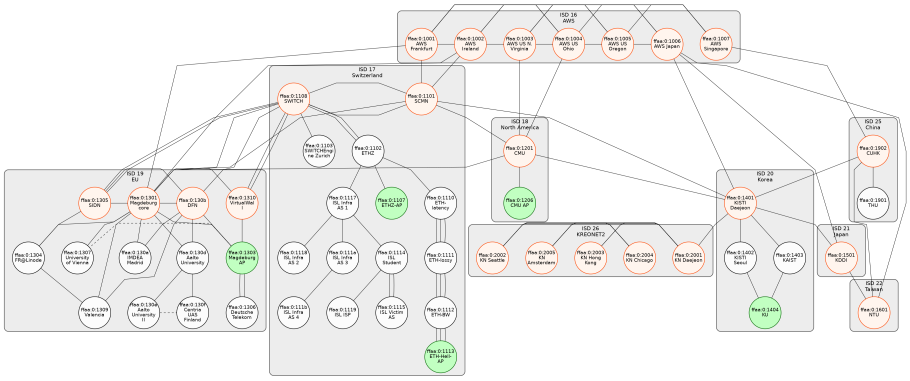


Figure 2.4: The SCIONLab topology showing the different ISDs and the connected AS. [11]

2.4 SCION IP Gateway (SIG)

As explained in the previous section, SCION aims for an incremental and easy deployment process, stimulating faster protocol adoption. Legacy hosts and equipment are not capable of talking native SCION since they are unaware of the new protocol. The SIG [4] aids in the gradual transition to SCION by solving that problem. The legacy hosts are unaware of the SCION functionality of the SIG. The hosts can communicate with each other as usual using IP but with the advantages of being connected via a secure SCION-based backbone. Fig. 2.5 illustrates how two SIG instances tunnel the traffic of IP-based hosts in different IPs networks over SCION.

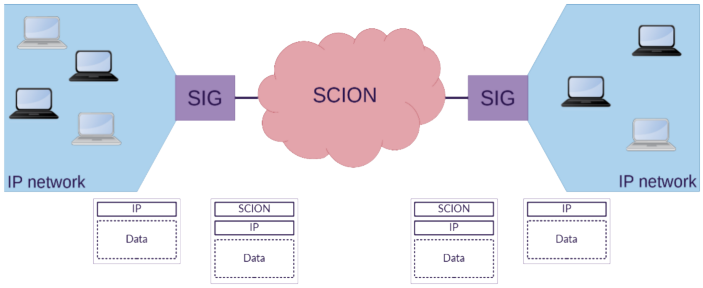


Figure 2.5: High-level overview of the SIG tunnel between the sending and receiving SIG connecting two IP networks over SCION. [12]

The SCION community created an implementation [7] of the SIG in Go [13]. Since it is open-source and written in Go, anyone can run the implementation on commodity server hardware which is widely available. However, the efficiency and throughput are likely lower than those achieved with an OPDP (e.g., based on P4 and a Tofino Application-Specific Integrated Circuit (ASIC)), as we will discuss in the next chapter.

The SIG encapsulates IP traffic between two SIG instances using the SIG Framing Protocol [14]. SCION/User Datagram Protocol (UDP) carries the SIG framing protocol, resulting in that the SCION dispatcher can handle the SCION packet receiving and distribution [5]. A SIG frame can encapsulate a partial, a single or multiple IP packet(s). We give a high-level overview of the SIG's tunnelling functionality in fig. 2.5. The sending SIG in the client's AS will encapsulate the IP traffic inside a SCION packet. The receiving SIG in the receiver's AS will unwrap the packet, resulting in a regular IP packet.

Fig. 2.6 gives an example of the different types of packet encapsulation. The blue dotted boxes are the SIG frames and the orange boxes are the encapsulated IP packets. For example, packet P3 is to big to fit in a single SIG frame so it is fragmented over frames with sequence numbers 1 through 3.

The data plane operations of the SIG are different for the sending and receiving

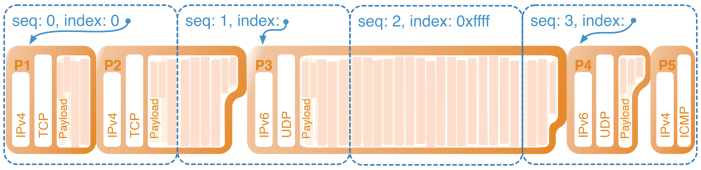


Figure 2.6: SIG framing protocol encapsulation examples showing the scenarios of IP packets being grouped into a single SIG frame or IP packets being fragmented over multiple SIG frames. [5]

sides. The sending side buffers the IP packets and encapsulates them with the SCION header and path information gathered from the control plane. The sending side might have to fragment incoming IP packets if these do not fit inside a single SIG frame. An example of this is packet `P3` in fig. 2.6. Packet `P3` is too big to fit inside a single SIG frame. Therefore it is fragmented over SIG frames 1 to 3. The SIG is also responsible for setting correct checksums and other fields of the lower level headers shown in fig. 2.8. The receiving side decapsulates the incoming IP packets carried by the SIG frames. It might need to perform reassembly of IP packets if the encapsulated IP packet is fragmented over multiple SIG frames. The SIG is not responsible for performing MAC verification since the border router performs this, which is within the same trusted AS.

SIG traffic can be sent over multiple SIG sessions. The SIG uses different sessions to transport different classes of traffic (e.g. priority vs. normal.) Within each session there may be multiple streams. Streams are useful to distinguish between traffic sent by different SIG instances. For example, if the SIG is restarted, it will create a new stream ID for each session. That way, the receiving SIG will know that the new frame with a new stream ID does not carry trailing part of the unfinished IP packet from a different stream [14].

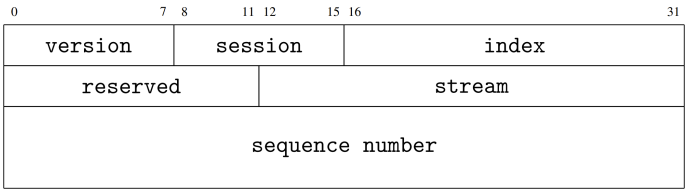


Figure 2.7: The SIG framing protocol header defenition. [5]

Fig. 2.7 shows the fields of the SIG frame header, and fig. 2.8 gives an overview of the total header stack. The header fields are defined as follows [5] [14]:

- The `version` field indicates the SIG framing version. Currently, only version 0 is supported.
- The `session` field indicates the used SIG session. The sessions are used to distinguish priority and regular traffic.

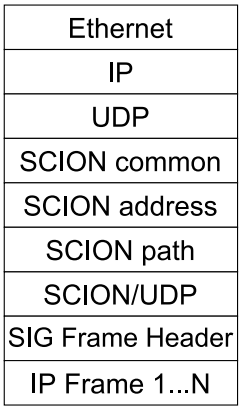


Figure 2.8: The entire header stack for the SIG framing protocol.

- The `index` is the byte offset of the first beginning of an IP packet within the payload. If no IP packet starts in the payload, for example, if the frame contains only a trailing part of an IP packet, the field must be set to `0xFFFF`.
- The `reserved` field is reserved and must set to zero.
- The `stream` and the `session` field identify a unique sequence of SIG frames.
- The `sequence number` field indicates the position of the frame within a stream. It starts from zero for a given direction of traffic and pair of SIGs (sending and receiving) and increases monotonically by one with every SIG frame. It resets whenever the sending SIG service restarts or the value reaches $2^{64} - 1$. Consecutive frames enable the reassembly of split IP packets split among multiple frames.

Chapter 3

Open Programmable Data Planes (OPDPs)

The second major concept in our work is that of an Open Programmable Data Plane (OPDP), which is the set of functions responsible for processing and forwarding packets implemented through open-source code. In this chapter, we introduce the concept and discuss the properties and types of OPDPs (sec. 3.1). Next, we discuss the advantages of an open data plane compared to a closed data plane (sec. 3.2). Finally, we explain the different programming languages we use for the OPDP implementations (sec. 3.3).

3.1 OPDP Types

Networking equipment consists of a control plane and a data plane. The data plane is responsible for processing and forwarding network packets, whereas the control plane's primary responsibility is deciding where and how to route the packet. The control plane fills the routing tables necessary for the data plane to forward packets. There are multiple implementation levels for the data plane, hardware-based, kernel-based and software-based, each with their pros and cons. We discuss these pros and cons for each space in the sections below. Fig. 3.1 shows a high-level abstraction of different data plane possibilities using the SIG as an example. The arrows indicate that the lower levels have increased performance at the cost of flexibility. The following subsections explain the different spaces for the data plane implementations.

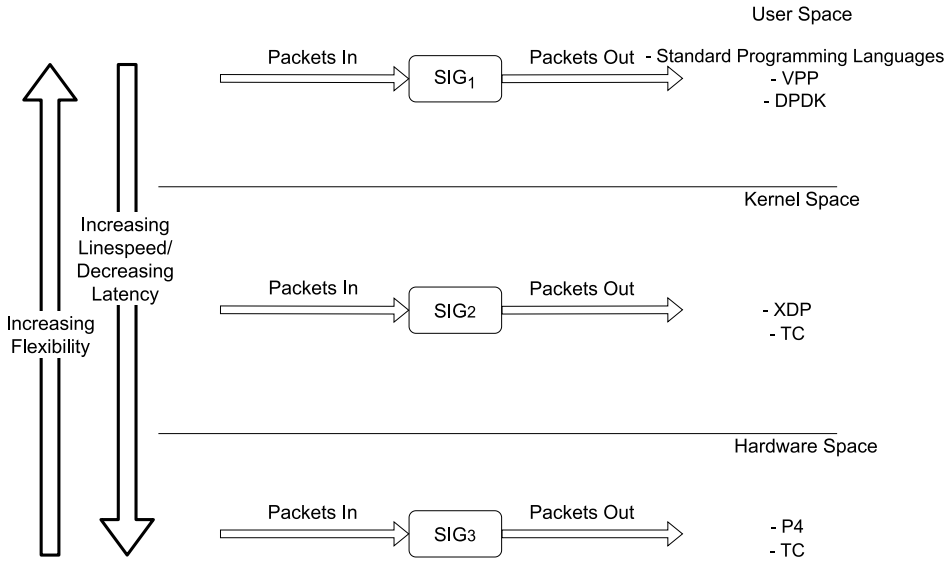


Figure 3.1: An overview of the different spaces at which a data plane can be implemented. The arrows indicate that the hardware space has the highest throughput, lowest latency and lowest flexibility whereas user space has it the other way around. We also list some of the available programming languages per space.

3.1.1 Hardware-based

Most high throughput current generation routers or switches perform packet forwarding using ASICs [15]. These ASICs are fixed-function hardware chips used for their efficiency and relatively low cost. The packets are forwarded using hardware only; hence there is no operating system involved for the packet forwarding. However, most devices still incorporate an operating system for control plane operations.

The designs of the ASICs are highly optimised for packet processing, which is possible because they only have to perform packet forwarding. There are no wasted clock cycles on non-packet forwarding tasks reducing the overhead. In current generation networking equipment, the work is highly parallelised [16]. A pipelined hardware design enables parallel operation inside the system (e.g., the Intel Tofino see sec. 3.3.1).

Fixed function ASICs lack re-programmability and transparency. P4 is an open-source domain-specific language that aims to solve this problem by describing the behaviour of networking hardware. P4 [17], in combination with P4-enabled switches, offer high-performance re-configurable packet forwarding. The performance is in the order of terabits per second. We describe the working and principle behind P4 in more detail in sec. 3.3.1.

3.1.2 Kernel-based

The kernel is the lowest level program in an operating system and has complete control over the system. It is responsible for tasks such as managing the interactions between hardware and software, process scheduling, memory management, peripheral management, I/O operations, interrupt handling, networking [18] [19]. Thus, the kernel performs the low-level tasks and management to offer generic interfaces in the form of syscalls, file descriptors and sockets for higher-level user space applications.

For security reasons, there is a distinction between user and kernel mode. A program loaded into kernel mode is privileged and can directly access hardware and all other resources, e.g. user space programs. Within a computer's memory, there is a distinction between these two modes. Programs in kernel mode exist in a separate virtual address space shared between all kernel programs, whereas user mode programs each have their own virtual memory space. Kernel programs can reference user programs but not the other way around.

Located inside the Linux kernel is the Linux network stack [20]. The network stack handles the link, network and transport layer operations. Handling these layers is not the only responsibility of the kernel; it also performs routing, forwarding and filtering operations. Device drivers interface with the Network Interface Card (NIC) and the kernel to send or receive packets.

Due to all the networking functionality inside the Linux kernel, it is not as performant as it could be for given use cases. eBPF is a technology that allows developers to attach custom programs to low-level kernel hooks [21], enabling the user to safely deploy code within the Linux kernel space. eBPF is widely used to increase packet processing speed or to add more customisation flexibility [22]. We explain eBPF and its possibilities in more detail in sec. 3.3.2.

3.1.3 User Space-based

The kernel and user space programs communicate packets via a socket. The socket acts as the boundary to cross kernel and user space. A user space program listens on this socket for new packets or sends packets over it for the kernel to transmit.

The advantage of user space packet processing is that it offers the highest flexibility out of all spaces. There are fewer restrictions and constraints from the hardware or kernel; there are, e.g. no real-time deadlines. The developer is less restrained in programming language choice within the user space, and the code is portable to different operating systems. An additional advantage of user space programming is that there is less required knowledge of the lower-level networking operations for developers since the kernel performs these operations.

Since a packet is always first processed by the kernel and only then by the user space program, the user space program is less performant in packet processing operations than the kernel. The performance decrease is logical since the packet has already traversed the entire networking kernel stack. A context switch between kernel and user mode additionally slows down the process [19]. A context switch is costly for performance because of the required reload of registers, stacks and address spaces.

A solution for this problem is Vector Packet Processing (VPP) [23] [24] and Data Plane Development Kit (DPDK) [25] [26]. These solutions bring the NIC and its operation into user space, removing the need for context switches. VPP/DPDK has the disadvantage of needing to perform all low-level operations within the program and making the NIC unavailable for the rest of the system. We do not consider VPP/DPDK because of time limitations.

3.1.4 Hybrid-based

Besides deploying an OPDP in a dedicated space, combining the different spaces is possible, resulting in a hybrid-based OPDP. Such a hybrid-based OPDP is achieved by deploying multiple OPDP implementations in different spaces since a single OPDP can not span over multiple spaces. An example would be to deploy a kernel-based eBPG OPDP in parallel to a user space OPDP.

For example, a hybrid-based OPDP can combine the positive characteristics of the used spaces. The lower level hardware or kernel-based OPDP would form a fast path data plane where the user space OPDP could handle packets requiring more advanced operations. The lower level OPDP receives the packets and forward the unsupported packets (e.g., packets requiring reassembly operations) to a higher level OPDP for processing the packet.

3.2 Open versus Closed Programmable Data Planes

We define open networking as a network infrastructure based on open-source programmable data planes. We believe that programmable state-of-the-art networking technologies are essential for a production-ready open network and for deploying new protocols such as SCION. The use of OPDPs can provide new network properties, such as the transparency, flexibility, energy efficiency and line rate required for operating networks at scale, a requirement for, e.g. ISPs.

Open networking is unlike current fixed function networking equipment [27] which are black-boxes. Vendors often supply networking equipment with closed-source pre-installed software, which the user can only reconfigure to a certain degree.

The user is very limited in reprogramming or understanding the data plane of the equipment itself, with the result that changes in functionality or the support for new protocols are limited and in some cases impossible. The equipment is not only a black-box concerning the functionality, but it is also a black-box with regard to the metrics. Most current-generation networking equipment lacks real-time telemetry, which is an issue concerning the openness and monitoring of the equipment [27].

With open networking, the network uses re-configurable code resulting in agile programmable networks [28]. We expect the agile nature to positively influence the experimentation and adoption of new protocols used within the Internet. Besides an OPDP, an open control plane is also required to achieve the goals of open networking.

3.3 Programming Languages for OPDPs

In this section, we explain the programming languages we use for implementing our OPDPs. Other OPDP programming languages like, e.g. Rust are outside the scope of our research. We start with P4 (sec. 3.3.1), a domain-specific language used to describe hardware-based OPDPs. Next, we explain eBPF (sec. 3.3.2), a technology that enables the creation of kernel-based OPDPs. Finally, we explain XDP (sec. 3.3.3) and TC (sec. 3.3.4), which are two specific types of eBPF networking programs.

3.3.1 P4

Programming Protocol-independent Packet Processors (P4) [17] is a domain-specific language for describing the behaviour of networking devices. We show P4 in fig. 3.1 as language to write hardware-based OPDPs. P4 is suited to describe the data plane algorithms which can be compiled for multiple targets like Field Programmable Gate Arrays (FPGAs), NICs, switches and software models. It achieves this by using target-specific architectures and compilers whilst keeping the same language syntax. Fig. 3.2 shows the general workflow for a P4 program. The developer writes a P4 program (top purple block), and a compiler is used to generate an info file for the control plane and a binary file for deployment on the P4 target (data plane).

P4 defines standard building blocks like headers, tables, actions, parsers, deparsers, match action units and externs. The availability and the exact functionality of these building blocks depends on the P4 target architecture used. In fig. 3.3, we show an example architecture and how the building blocks are combined to form the pipeline. The architecture shown is of the Public Switch Architecture (PSA) [29], an

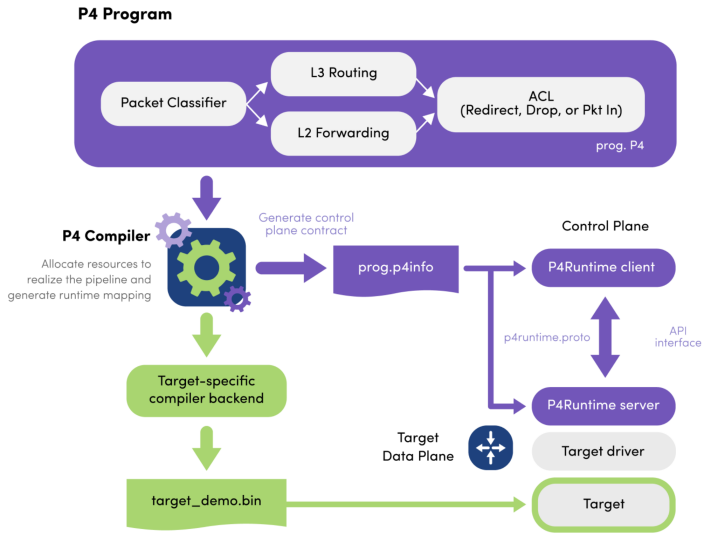


Figure 3.2: A high-level overview of the general P4 workflow. [17]

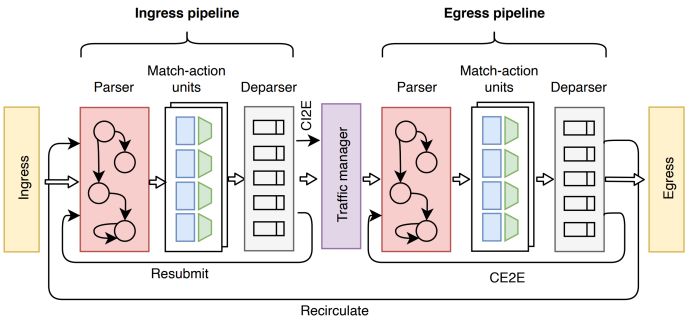


Figure 3.3: Overview of the components making up the PSA architecture. [31]

open-source architecture used in the Behavioural Model version 2 (BMv2) [30] software switch. We describe the functionality of the different components in the bullet list below.

We will now go over the different building blocks and describe their purpose:

- Headers define the packet's header fields, possibly metadata, on a bit level.
- Tables store information, enabling state between packets in the data plane and control plane. The control plane program fills the tables with data, e.g. routing data. The data plane can match against these tables whilst operating at line rate.
- Actions define the different possible packet operations to apply to the packet. The operations range from dropping the packet to altering the bits in the parsed headers.
- Parsers parse the incoming packets and make the information in the headers available to the program. They can also skip and drop header bits.

- Deparsers function the same as parsers but perform after the parsing and logic operations of the pipeline. They usually handle and remove specific metadata still attached to the packet.

- Match action units are control blocks which use data in the parsed headers as entries for table matching to determine what actions to perform on the packets.

- Externs are custom functions implemented in a target architecture to aid specific tasks. Examples of this are checksums and Cyclic Redundancy Check (CRC) calculations.

P4 has a syntax similar to the C programming language and uses the building blocks mentioned above to describe the program's behaviour. These high-level constructs allow for easy experimentation and reconfiguring of the chip's behaviour. This flexibility is opposite to the current generation ASIC switches with vendor-locked functionality.

The Intel Tofino [32] is a hardware ASIC-based switch with P4 compatibility with throughput in the terabits per second range. Intel defined the Tofino Native Architecture (TNA) [33], a P4 architecture that provides a P4 programming interface for Tofino switches. This specification is open-source, but for accessing in-depth documentation and Software Development Environment (SDE) releases signing an Non-Disclosure Agreement (NDA) and Software License Agreement (SLA) is required. Intel supports academic researchers through the Intel Connectivity Research Program (ICRP). The bf-p4c compiler is the custom compiler required for compiling P4 code for Tofino.

Tofino can achieve the terabits per second line rate by utilising hardware-based parallelisation. Fig. 3.4 shows the block diagram of the Tofino, where four different pipes can be distinguished, labelled zero through three. Each of these four pipes is attached to several physical ports and can be programmed independently of each other. The Traffic Manager is a block that allows advanced operations on packets between the pipes like recirculation, mirroring and replication. The Tofino also has the capability of sending packets to the CPU. This feature can be helpful for further packet inspection or handling edge cases on the CPU.

Fig. 3.5 shows the core software components and APIs used in the Tofino switch. The control plane program uses these APIs to fill the match-action tables and configure the Tofino settings and externs. In this figure, the SDE is not present. With the SDE, developers can run their programs to simulate the behaviour and verify correct functionality. The bf-p4c compiler is shown in fig. 3.5, which generates the binary file for deployment on the Tofino hardware (under the Linux kernel line) and the files necessary for configuring the APIs.

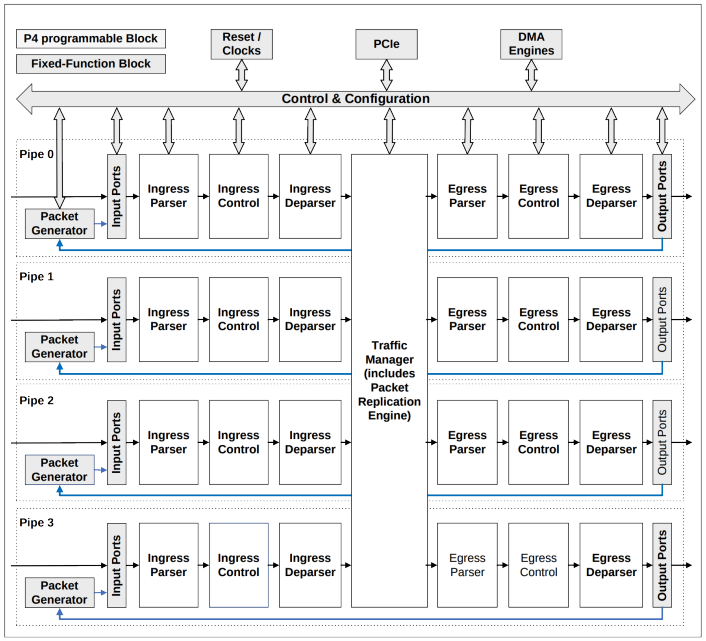


Figure 3.4: A block diagram of the Intel Tofino. It clearly shows the different pipes enabling the parallelisation. [33]

TNA supports the following externs (custom functions) that could be of interest: `Checksum`, `Counter`, `CRCPolynomial`, `Meter`, `Mirror`, `Random` and `Resubmit`. The TNA public document [33] describes their workings in-depth.

When developing a P4 program for the Tofino, there is a big difference compared to a software model like the BMv2 [30]. Being mapped to physical hardware adds extra constraints on the design, but this also enables the orders of magnitude higher throughput. Limited resources are one of these constraints. For example, TCAM or SRAM memory stores the data of the P4 tables, but this resource is limited by the amount present in the hardware. Another example of limited resources is the finite amount of states available for the parser and deparsers and a finite parsing depth. Besides limited resources, the hardware also has additional timing constraints it has to meet. The SDE takes care of and checks the supplied program for these constraints.

3.3.2 eBPF

eBPF is used to implement kernel-based OPDPs as shown in fig. 3.1. Traditionally the kernel is hard to change because of its critical role and high requirements for security and stability. eBPF (not an acronym [21]) is a technology that allows programs to run in a sandboxed environment inside the kernel. These programs can be attached to the kernel at runtime without reloading kernel modules or recompiling the kernel, resulting in the ability to augment the kernel's functionality without

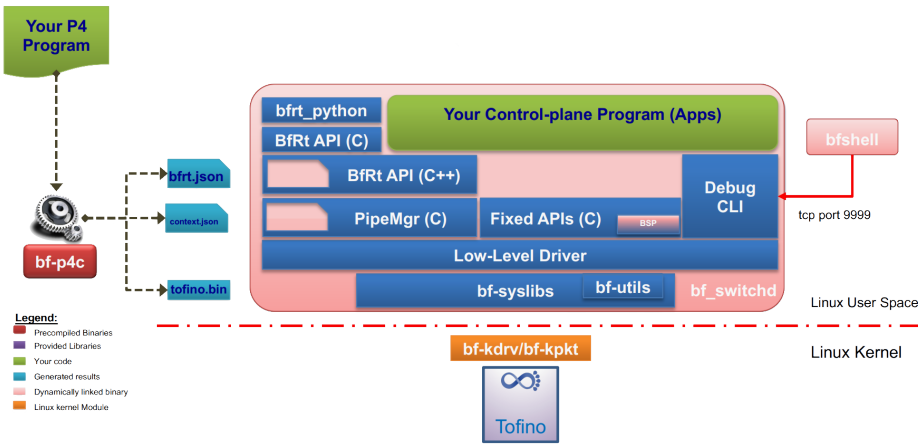


Figure 3.5: An overview of the different software components (APIs and compiler) of the Tofino SDE. [34]

downtime. Executing the program sandboxed inside kernel space enables the elevated permissions required for certain operations, like direct hardware access, whilst guaranteeing kernel stability and security [21].

Linux kernel version 3.15 (2014) introduced eBPF into the kernel. eBPF is the newer and extended version of the Berkeley Packet Filter (BPF) originating from BSD in 1992 [35]. eBPF's most significant changes were a 64-bit word size, an increase in the number of registers, function calls, a 512-byte stack, the addition of map storage and the capability to attach eBPF programs to different kernel hooks. Adding different kernel hooks gives eBPF more functionality than BPF, which could only perform network packet operations.

eBPF is an Instruction Set Architecture (ISA) combined with an in-kernel Virtual Machine (VM). The VM allows dynamic reloading of the programs. The kernel translates the instructions dynamically using Just in Time (JIT). The code can be offloaded to specialised hardware, e.g. NICs or executed on the CPU. Registers pass arguments to functions, allowing function calls to be one instruction. Table 3.1 shows the different registers and their function.

Register	Description
r0	Stores function return value
r1-r5	Store function arguments
r6-r9	Preserve values between functions
r10	Stack pointer (Read-Only)

Table 3.1: eBPF registers and their functionality.

Fig. 3.6 shows a typical workflow for creating and running eBPF programs. The compiler compiles the source code, typically written in a subset of the C program-

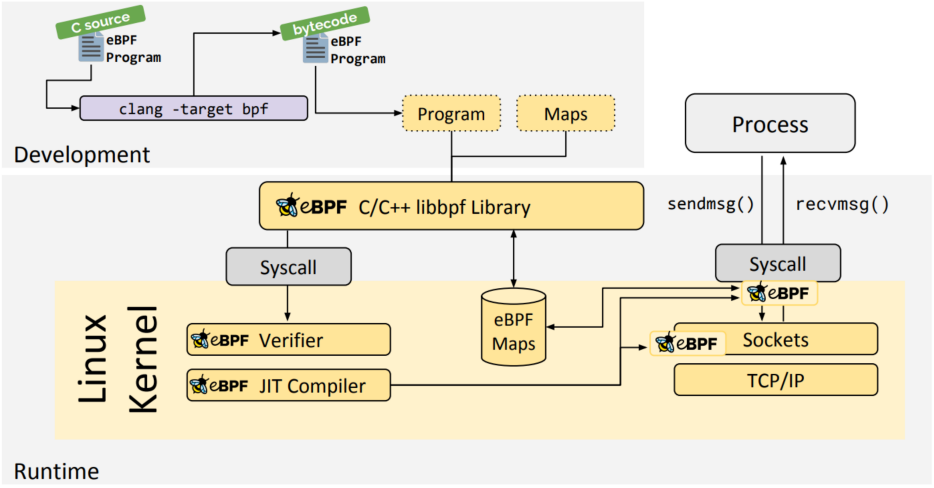


Figure 3.6: An overview of the eBPF workflow and the different components. [21]

ming language. The compiler checks if there are no unbounded loops, no use of the standard library nor non-static global variables. The compilation results in an Executable and Linkable Format (ELF) binary file ready to load in the kernel.

The verifier is responsible for checking the ELF object file for safety and integrity reasons before allowing the code to execute. To start, the verifier determines if the code is a Directed Acyclic Graph (DAG). Finding a DAG would guarantee the program's termination and enable the computation of the Worst-Case Execution Time (WCET). Furthermore, the verifier checks if all function calls are to functions with a General Public License (GPL) compatible licenses and checks all memory accesses. To ensure the kernel's integrity and security, checking the memory boundaries is essential to remediate buffer overflow attacks or reading non-allowed memory [36]. The program is rejected and not executed if the verifier fails to check one of these criteria.

An important concept in eBPF are so-called eBPF maps. eBPF maps are key-value stores to persist data between program executions (packets in our context) and to share data with other eBPF programs. These maps can be of different types, from array maps and hash maps to Longest Prefix Matching (LPM) maps specific to IP address matching. Besides eBPF programs, user space programs can also access the maps enabling interaction between the user and kernel space. The user space programs can retrieve, add, update and delete these entries.

eBPF programs can be attached to different hooks inside the kernel to augment standard behaviour. This augmentation behaviour of eBPF has the advantage that the kernel programs can run beside the user space programs allowing us to create a hybrid SIG implementation. We are only interested in the eBPF network hooks since our thesis focuses on accelerating the SIG using OPDPs. The available networking hooks are XDP and TC. Fig. 3.7 gives a visualisation of the Linux network stack

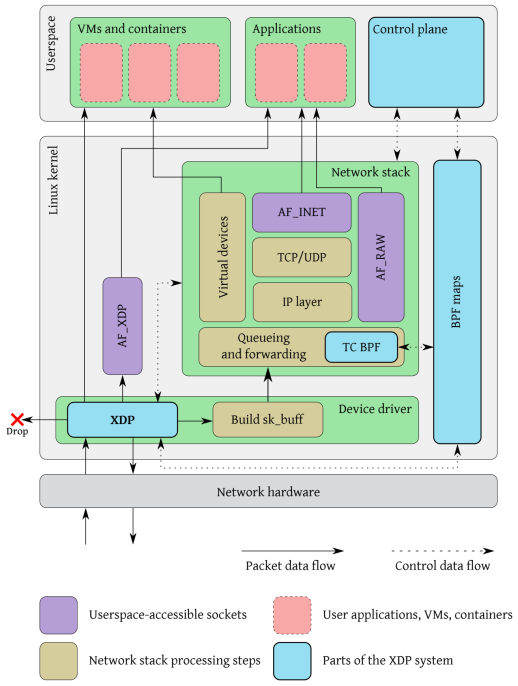


Figure 3.7: Schematic overview of the Linux Network stack with eBPF networking hooks TC and XDP shown. [37]

and eBPF hooks. Packets enter the Linux kernel via the network hardware and are passed on to XDP in the device driver. XDP processes the packets if an XDP program is attached to the interface. If XDP passes on the packet, the packet will enter the network stack (biggest green box). There the packet arrives at TC the second and last eBPF network hook.

3.3.3 XDP

eXpress Data Path (XDP) [37] is the lowest level networking hook, which means the program executes directly after the NIC receives a packet. XDP defines a structure called `xdp_md` containing pointers to the packet's beginning and end memory addresses. This data type is more barebones than the usual `skb_buff` type used inside the kernel because `xdp_md` removes the dependency on the kernel network packet parsing functionality. It also means that the XDP program has less information at its disposal.

Return values of eBPF programs signal to the kernel what action to take. Table 3.2 shows the possible return values an XDP program can have. For the SIG, `XDP_PASS` and `XDP_REDIRECT` are of interest. `XDP_PASS` signals that the packet should traverse the networking stack as usual, whereas `XDP_REDIRECT` signals that the packet should redirect to another interface.

eXpress Data Path (XDP) introduces `AF_XDP` in Linux kernel 4.18 [38], a new

XDP return values	Description
XDP_PASS	Pass the packet onto the network stack
XDP_DROP	Drop the packet
XDP_ABORT	Drop the packet creating a kernel
XDP_TX	Retransmit the packet out of the interface
XDP_REDIRECT	Redirect the packet to the egress side of another interface

Table 3.2: The possible XDP return values and their meaning.

socket type. This socket allows XDP to create an in-kernel fast path to the user space bypassing the Linux kernel network stack. XDP programs use XDP_REDIRECT to forward packets to the AF_XDP. Applications must be adjusted to take advantage of the AF_XDP functionality.

There are three modes to deploy XDP programs. These modes are offloaded, native and generic. In offloaded mode, the program executes on the NIC. Currently, only Netronome NICs with the nfp driver support this mode [39]. In native mode, the program executes using the CPU but does not require allocation of the skb_buff. The generic mode supports all NICs and interfaces since the required XDP code is in the kernel. Generic mode is the least performant since a skb_buff is allocated for every packet. Offloaded and native modes are similar in performance, but the offloaded mode has lower CPU utilisation.

3.3.4 TC

The Linux Traffic Control (TC) part of the kernel executes right after the XDP hook. In TC, packets enter a queueing discipline (qdisc) to which rules, filters and policies can be attached. There is a special qdisc for eBPF TC programs called clsact. The clsact qdisc type allows the programs to hook into the ingress or egress side of the network interface, whereas XDP programs can only be attached to ingress.

TC eBPF programs also have defined return values. Table 3.3 shows the possible return values and their description. A difference between XDP and TC is that TC_ACT_REDIRECT can redirect the packet to either the ingress or egress side of the other interface [36].

TC return values	Description
TC_ACT_OK	Pass the packet onto the networking stack
TC_ACT_SHOT	Drop the packet
TC_ACT_PIPE	Goto next TC action
TC_ACT_RECLASSIFY	Start over the TC pipeline
TC_ACT_REDIRECT	Redirect the packet to ingress or egress of another interface

Table 3.3: The possible TC return values and their meaning.

Chapter 4

Related Work on SCION and OPDP Performance

To the best of our knowledge, at the time of writing, there has been no other research into the performance of the SIG nor the acceleration of the SIG using OPDPs. There have been performance studies on SCION border routers, OPDPs, and their combination, but not on the use of the SIG and OPDPs. Well-performing SIGs are particularly important to couple the large existing IP networks with emerging SCION-based networks.

In the following sections, we will elaborate on work done on benchmarking the SCION performance (sec. 4.1) as well as performance optimizations using kernel-based OPDPs (sec. 4.2) or hardware-based OPDPs (sec. 4.3).

4.1 SCION

Chandrashekar and Riaz [40] researched the performance of SCION on VMs and compared the results to IP routed traffic. The VMs were part of the SCIONLab network and were connected using direct gigabit links. Chandrashekar and Riaz measured a reliable SCION throughput of 150 megabits per second.

Gartner mentions a throughput of 600 megabytes per second in his master thesis ”Improving SCION Bittorrent with efficient Multipath Usage” [41]. Gartner achieved this using multiple connections over a 10 gigabit per second direct link using a deprecated high-speed border router implementation of Anapaya.

Wagner measured a baseline throughput of 1.2 gigabits per second in his thesis ”Improving Packet Processing Speed on SCION Endhosts” [10]. Wagner performed the measurements in a similar environment compared to Gartner. The servers were equipped with an Intel Xeon Silver 4114 CPU with 10 cores running at 2.20 GHz, and 48 GB of DDR4 RAM at a clock rate of 2666 MHz. The measurement was also

over a 10 gigabit per second direct link using the old Anapaya high-speed border router. Wagner argued that the performance increase compared to Gartner's result is due to not having the BitTorrent overhead.

4.2 XDP and TC

Wagner [10] used XDP to accelerate the SCION dispatcher and was able to quadruple the measured throughput. Wagner increased the throughput from 1.2 gigabits per second to 4.6 gigabits per second.

Schulz and Hausheer [42] created an XDP implementation to accelerate the performance of the SCION border router. Schulz and Hausheer perform the MAC verification that involves Advanced Encryption Standard (AES) inside eBPF.

Neukom achieves SCION file transfer speeds at nearly 100 gigabits per second in his master thesis [43]. Neukom uses the Hercules protocol and `AF_XDP` sockets on a high-performance testbed using 100 gigabits per second direct links. `AF_XDP` sockets are used to bypass the Linux network kernel and the SCION dispatcher. Neukom mentions neither the exact topology nor the use of a SCION border router.

NLnet Labs performed several XDP and TC experiments to accelerate and augment Domain Name System (DNS) and wrote a collection of blogs about their experiments [22]. The code of the experiments is available in their GitHub repository [44]. They do not quantify their performance gains.

CloudFlare uses XDP to drop packets at wire speed to further increase Distributed Denial of Service (DDoS) protection [45]. XDP allows them to drop the packets earlier in the packet processing pipeline, saving CPU resources.

Lee's presentation "Faster Packet Processing in Linux: XDP" [46] at the Samsung Open Source Conference (SOSCON) gives a quantitative comparison between the packet processing option in Linux comparing user space, Netfilter, TC and XDP. They could drop 783.063, 1.226.730, 4.083.820 and 9.941.337 packets per second, respectively, clearly illustrating the performance gains.

4.3 P4

Součková implemented the SCION border router for a NetFPGA [47] using P4. Součková describes this in her master thesis, "FPGA-based line-rate packet forwarding for the SCION future Internet architecture" [48] and achieves a line rate of 10 gigabits per second. The code is available on GitHub [49]. However, it is not actively maintained.

De Ruiter and Schutijser created an open-source implementation of the border router for the Intel Tofino [50]. In their paper ”Next-generation internet at terabit speed: SCION in P4” [51] they describe that the switch can run at terabits per second line rate. They made this possible by adapting the SCION protocol header definition to better suit hardware switching. These changes were adopted by the official protocol specification.

Van Hove presents a hybrid routing option based on software and hardware routed in ”P4 Hybrid Routing for Next-Generation Networks Experimentation based on Open Source Software Defined Routing” [52]. Van Hoven uses existing routing software to fill the P4 tables with routing information and tests them in two use cases.

Chapter 5

OPDP-based Implementations of the SIG

In this chapter, we explain the prototypes we created to accelerate the performance of the SIG using OPDPs. The prototype must interoperate with the existing SIG implementation and framing protocol. This compatibility is essential for the accelerated SIG to function outside a controlled test environment.

We start this chapter with the required operations for the data plane of the SIG (sec. 5.1). Then, we explain the P4 design created for the Intel Tofino (sec. 5.2), after which we will discuss the XDP OPDP (sec. 5.3). Finally, we explain the TC OPDP prototype (sec. 5.4), which we used for our measurements.

5.1 Requirements

From a functional perspective, our OPDP-based implementation needs to provide the SIG functions we outlined in sec. 2.4. The six key functions are:

- Receive IP packets.
- Put the IP packets in SIG frames.
- transmit SIG frames.
- Receive SIG frames.
- Retrieve IP packets (possible from multiple SIG frame).
- Send out IP packets.

The OPDP-based SIG also has non-functional requirements. As stated in the SCION book [5], all legacy traffic into (or out of) a SCION AS passes through the

SIG service through IP routing rules. Since all the legacy traffic is routed through the SIG, it must be fast to keep up with the packet flow whilst being robust and able to handle packet loss. Furthermore, the SIG functionality should be flexible, e.g. it should detect path changes.

5.2 Hardware-based Implementation using P4

We started with a design for the Intel Tofino since this would likely result in the most significant performance increase (see sec. 3.1). A downside of implementing the SIG in P4 is that we must create all the functionality from scratch since there is no user space implementation running in parallel as there is with the eBPF designs. In retrospect, a hybrid design might even be possible; we discuss this in sec. 8.1.

In order to create the prototype, we started with a design experiment for the first requirement. The experiment was to isolate a specific part of the packet payload. With isolation, we mean removing parts of the front and back of the packet's payload resulting in a specific piece of the payload. This behaviour is required for the SIG to decapsulate framed IP packets. An example of this requirement would be isolating packet `P1` in fig. 2.6 since that frame contains the beginning of packet `P2`. Therefore, removing the beginning part of packet `P2` from the frame is necessary to successfully decapsulate packet `P1`.

We successfully isolated specific parts of the packet payload. However, the range (number of bytes we could remove) of the operation was limited, resulting in that we could not isolate small packets in the middle of the payload of a SIG frame. We concluded that the hardware-based P4 platform does not provide the featureset we need to implement the SIG framing protocol since we were not able to perform the fragmentation and reassembly operations. This mismatch resulted in us stopping with the OPDP SIG prototype implementation in P4, and we further focused on an eBPF-based implementation. In sec. 8.1, we discuss the mismatch more thoroughly.

5.3 Kernel-based Implementation using XDP

Since XDP is the lowest level eBPF networking kernel hook, we start by making an XDP-based implementation. We give an overview of the architecture in fig. 5.1, which clearly illustrates the fast path. Another thing to notice in the figure is that the XDP program runs parallel to the user space SIG. The parallel deployment has the advantage of not having to cover all edge cases since we can pass packets from the unsupported cases to the user space SIG. The parallel operation enables us to

create a hybrid design between user and kernel space, implementing a fast path for most packets.

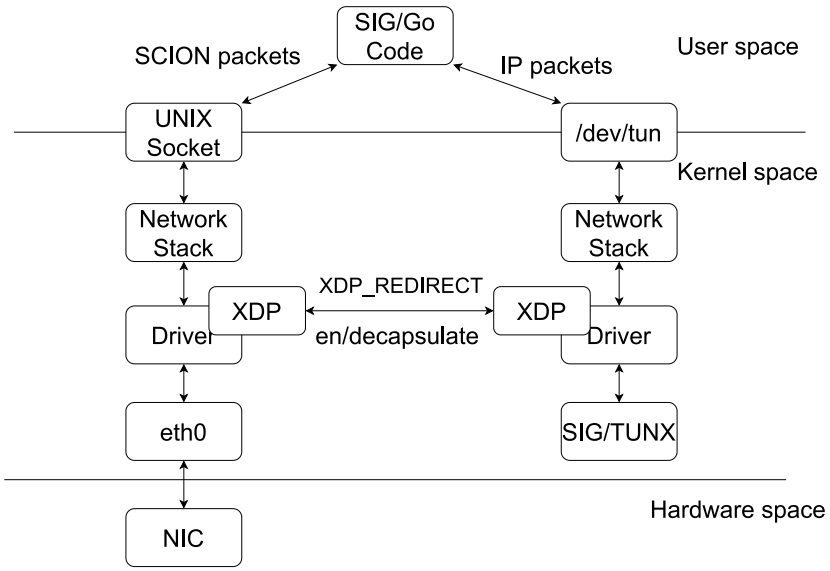


Figure 5.1: XDP prototype architecture clearly illustrating the fast path using `XDP_REDIRECT`.

To clearly illustrate the logic of our OPDP prototype, we created a flow diagram shown in fig. 5.2. The left and right flows represent the logic applied to the Ethernet and SIG interfaces, respectively. We will use the number and letter markings of the flow graph in our explanation. From a high-level, the Ethernet interface receives and decapsulates the SCION packets. Conversely, the SIG interface receives and encapsulates the IP packets. The error handling consists of passing on the packets to the user space SIG, 'PASS' in the flow diagram. The current design can only fast-track SIG frames containing a single entire IP packet. We will now describe the steps in more detail.

The SCION side (on the left) starts by parsing the Ethernet `(1)`, IP `(2)` and UDP `(3)` headers and checks if the UDP port number signals a SCION payload. Next, we parse the SCION common header `(4)` and skip over the variable length address and path headers using the header length information from the common header. We then check for a SCION/UDP header `(5)` and parse if present to see if the port number signals a SIG frame header.

If we find a SIG frame, we parse it and check the index number to see if it is set to zero `(6)`, indicating that at the start of the payload is an IP packet. If that is the case, we can parse the IP header and check if the destination IP address of the packet indicates an in or outgoing SIG frame `(7)`. For an outgoing SIG frame, we check if the key (destination IP address) is new for the map `(8)`. If this is the case, the packet contains new header information, which can be used at the SIG side to encapsulate IP packets. Therefore, we add the raw bytes of the Ethernet, IP, UDP and SCION

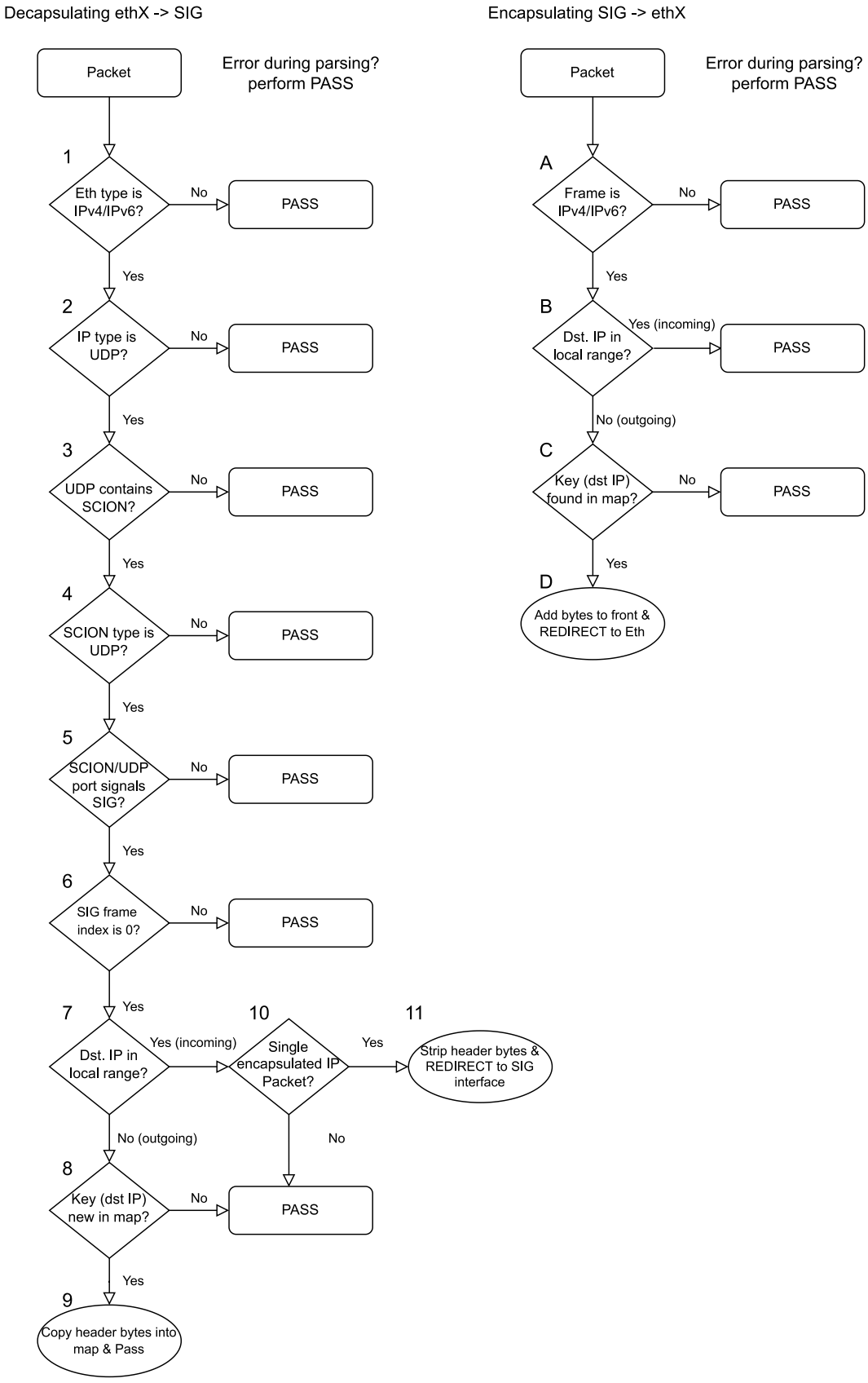


Figure 5.2: Flow diagram for the XDP prototype illustrating the logic and performed actions for both interfaces. The left flow is for the Ethernet interface and the right flow for the SIG interface.

headers as value to the map (9) using the destination IP address as the key and pass the packet out of the interface. For an incoming SIG frame, we check if the entire IP packet and only one IP packet is contained in the frame (10). We check this by comparing the packet's total length to the total parsed header size combined with the payload size indicated in the framed IP header. If this is the case, we strip the encapsulated IP packet from the lower level header and redirect the packet to the SIG interface (11).

Now turning our attention to the right-hand side of the diagram, the SIG interface handling IP packets parses the packet's IP header (A) and checks if the packet is incoming or outgoing using the destination address (B). If the packet is outgoing, we check if the key (destination address) is found in the map (C). If a match exists, the bytes stored in the map are used to encapsulate the packet (D), and then we redirect the packet to the Ethernet interface.

Our design is user space agnostic since the eBPF implementation fills the map required for encapsulation by parsing outgoing packets inside the kernel space. Therefore, there is no dependency on the user space implementation of the SIG. The self-learning design enables compatibility with running in parallel to other future user space implementations. Accordingly, the design not only fulfils the requirement of being compatible with the current implementation, it even is compatible with future implementations.

We have created a user space helper program for our design. The program periodically loops over the map entries assigning a Time To Live (TTL) to new entries and deleting entries with an expired TTL. This feature enables updating the map entries since otherwise, the learned MAC inside the SCION header would become invalid. The program also dynamically loads the interface index into a map determined by the interface name. The dynamic reloading is useful because the interface index may change. The eBPF program uses this index for the redirect operation.

We did the initial development of the design on our laptop and used `tcpreplay` [53] to mimic the incoming packets on both interfaces. In that setup, we made a working OPDP prototype. However, we encountered issues once we deployed the prototype to the testbed. Packets would not appear in XDP; hence they could not be redirected. The problem was that XDP is only attached to the ingress side of the interface. Therefore it did not see packets leaving the interface. `tcpreplay` sent all the captured packets (incoming or outgoing) to the ingress explaining why we overlooked the issue during development.

Fig. 5.3 shows the ingress and egress flow directions for our two interfaces. XDP programs not capable of attaching to the egress side of interfaces was not our only problem. Forwarding packets from the Ethernet ingress to SIG ingress was also problematic since ingress to ingress redirection is currently not supported in XDP.

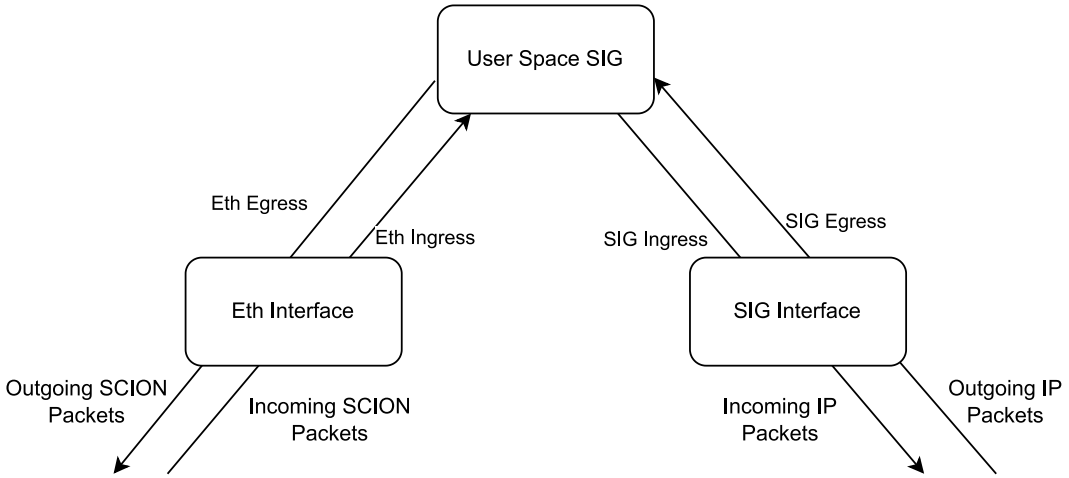


Figure 5.3: Overview of the ingress and egress direction for both interfaces.

XDP developers are working on adding ingress to ingress redirection and egress attach points [54]. However, the functionality was not available at the time of writing. As a result, we changed our approach slightly, which we will explain next.

5.4 Kernel-based Implementation using TC

Since ingress to ingress redirection and the program's attachment to egress hooks are required and because XDP is not capable of doing that, we switched to using TC. TC allows us to attach programs to the ingress and egress side of the interface and redirect packets to either from another interface.

Luckily porting our XDP OPDP to use the TC hook was relatively straightforward since both designs are eBPF programs. We changed the `xdp_md` struct to `skb_buff` and some `bpf_helper` functions. The main change was splitting the logic per interface into ingress and egress programs. A benefit was that the ingress and egress split removed the necessity to check whether the packets were incoming or outgoing on the interface, further simplifying the programs.

We find the direction for ingress and egress of the two interfaces a bit confusing since they seem mirrored for the two interfaces. We assume that the SIG interface has the ingress and egress side based on the perspective of the socket. We illustrate in fig. 5.3 how the ingress and egress are defined together with the traffic flow per interface.

Fig. 5.4 contains our updated flow diagram. The main difference is that we split the program's functionality to ingress and egress for each interface. The split removes the need to check if the IP destination address is in the local range (step `(7)` in fig. 5.2). We can now directly check if the SIG frame contains only an entire single IP packet `(7)`, after which we strip the headers and perform a redirection to the SIG

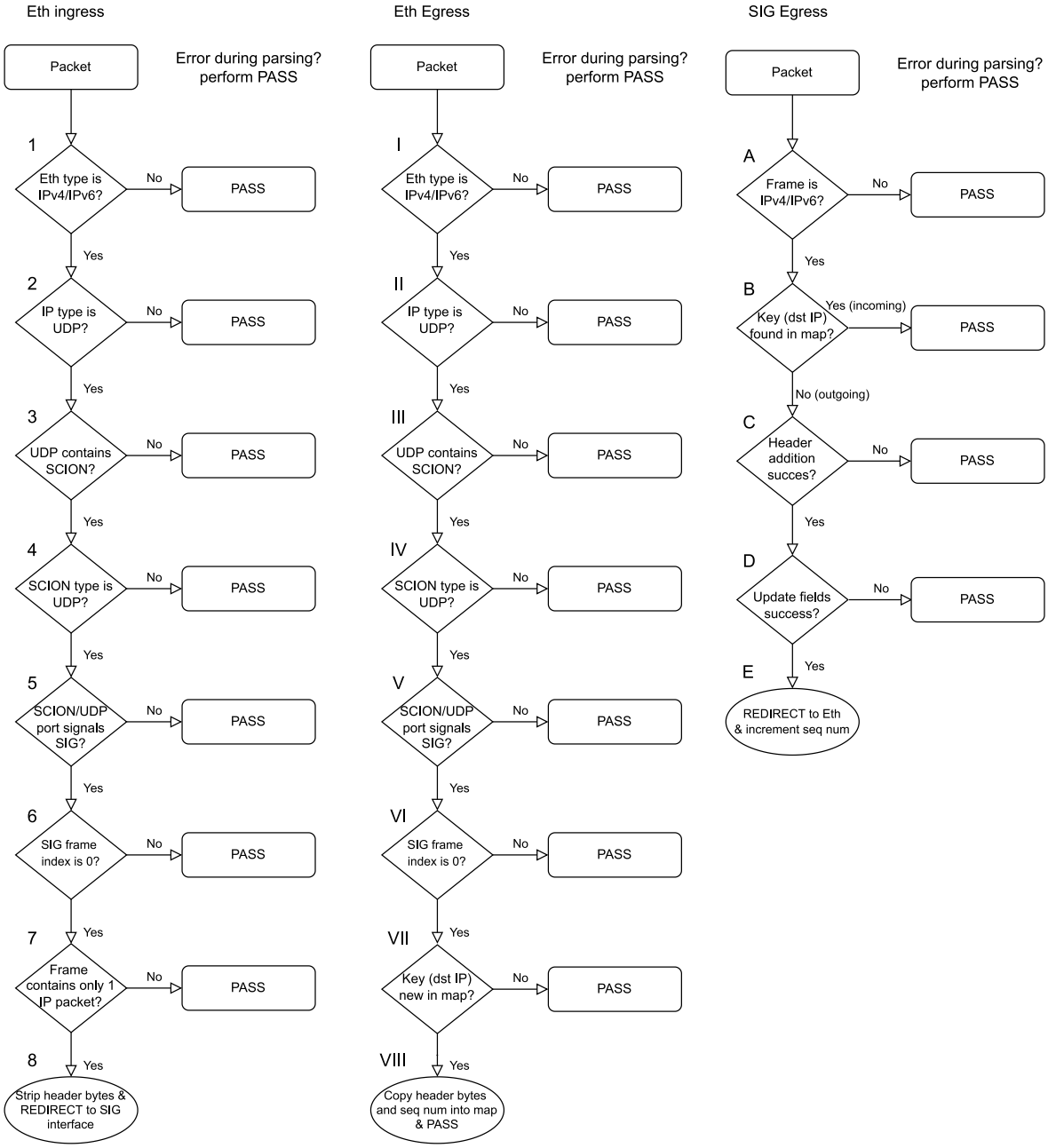


Figure 5.4: Flow diagram for the TC prototype illustrating the logic and performed actions for the attached TC programs. The left flow is for the Ethernet ingress, the middle flow is for the Ethernet egress and the right flow for the SIG egress.

interface. Alternatively, for the egress side, we check if the IP destination address key is new for the map (VII). If the key is new, we add the raw header bytes to the map and add the sequence number of the packet to the sequence map (VIII).

We created a secondary map for storing the SIG sequence numbers since we need to increment the SIG frame sequence field for every packet we send. Additionally, we update the IP length, IP checksum, UDP length and UDP checksum fields (D) in order to match the length of the new packet. These changes are necessary since the map entry could be filled with header data from a packet of a different size, causing errors due to header fields indicating wrong lengths. We set the value of the UDP checksum to 0 to bypass the kernel checks since this checksum includes the payload. In sec. 8.3, we discuss the consequences of the bypass.

To summarise, the four (ingress and egress for two interfaces) programs have the following functionality:

- The Ethernet ingress decapsulates the incoming SIG frames and redirects the decapsulated packets to the SIG ingress.
- The Ethernet egress checks the outgoing SIG frames and stores the header bytes for new map keys (destination IP address).
- The SIG ingress does nothing.
- The SIG egress checks for the presence of the key (IP destination address) in the map. If found, it encapsulates the packet with the bytes from the map and redirects the packet to the Ethernet egress.

Chapter 6

Measurement Methodology

The goal of our measurements is to evaluate the performance of the TC-based OPDP we implemented (see sec. 5.4). To start, we explain the performance parameters we use for the measurements (sec. 6.1). After that, we explain our measurement setup (sec. 6.2) and measurement traffic and scenarios (sec. 6.3). Then we discuss the tooling we use for our measurements (sec. 6.4). Finally, we explain how we make the measurements reproducible (sec. 6.5). We use our methodology for our measurements of the TC-based SIG in chapter 7.

6.1 Performance Parameters

The main criteria we use to measure the performance are the Round Trip Time (RTT) and the throughput between two SIG instances. We give a high-level overview of two SIGs in fig. 2.5.

The RTT consists of the network latency and the processing delay. The network latency will be constant and significantly smaller than the processing delay as we use a virtualised testbed (see sec. 6.2). The distance between the nodes typically contributes to the network latency; however, in our case, the virtualisation driver determines the network latency. The virtualisation driver acts as the switch forwarding the packets between the VMs. The processing delay is the most interesting to us since the SIG implementation influences the processing delay. The goal of our acceleration is to reduce the processing delay, which would result in a reduction in the RTT. End-hosts would, e.g. notice the delay reduction when establishing a connection.

We define throughput as the rate of successfully received bits at the measurement point. We use the throughput to determine the maximum packet forwarding speed of the SIG. The throughput is of interest since the traffic of many end-hosts will be routed through the SIG. Therefore, if the SIG is not able to handle high

throughput, it would cause congestion in the network.

Additionally, we measure CPU utilisation. We use CPU utilisation to determine any bottlenecks in the system. Besides bottlenecks, utilisation also determines the efficiency of the system. If two implementations achieve the same RTT and throughput, a lower CPU utilisation is preferred to save on computing resources.

We monitor the packet loss since a high packet loss is not desirable. Packet loss might result in required retransmissions, thus wasting networking resources. Packet loss could e.g. occur if the user space SIG drops packets due to sequence number errors.

6.2 Measurement Setup

Our goal for the testbed setup was to create the minimal topology required for measuring the performance of two SIGs whilst avoiding possible bottlenecks and ruling out the effects of the network. Fig. 6.1 gives an overview of our virtualised testbed. Our testbed consists of two ASes, the blue boxes in the figure, and each AS consists of two VM. We use two ASes since a SIG is not required for communication within a single AS. The SIG is not required because hosts can directly communicate inside a single AS without needing SCION as the inter-domain routing protocol. For simplicity, we used one ISD with one of the two ASes forming the ISD core.

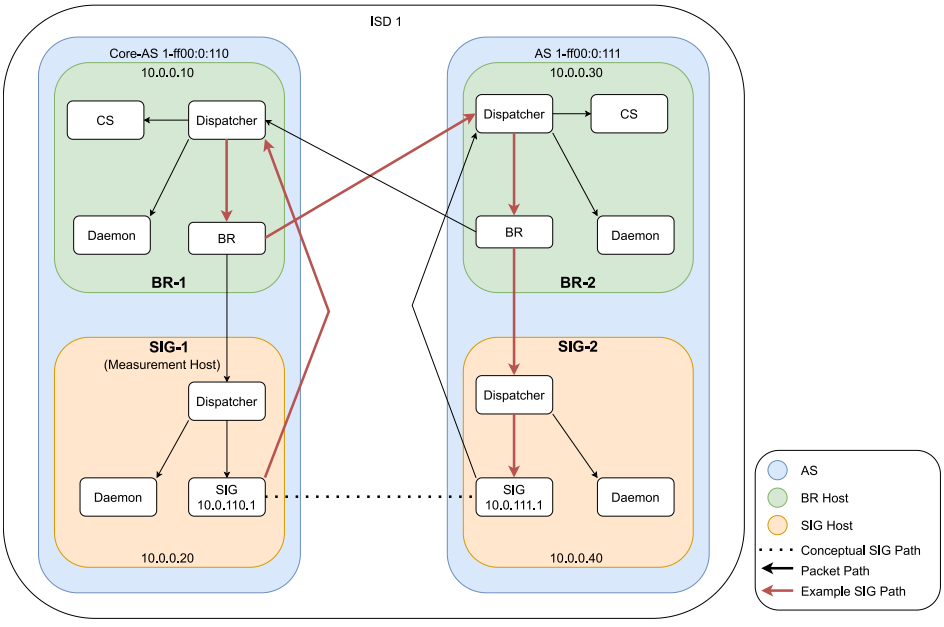


Figure 6.1: Schematic overview of the testbed design with the four VMs and the different components within each VM. The arrows indicate the communction between components whereas the dotted line symbolyses the communication between the SIGs.

Each AS contains a Border Router (BR) and a SIG host, the green and orange

boxes in the figure, respectively. The BR hosts contain a Control Service (CS), a SCION daemon, a dispatcher and a BR. The CS performs the control plane operations for the AS, whereas the daemon performs these operations for the host. The control plane operations include path, beaconing and certificate operations. The dispatcher is responsible for receiving incoming SCION packets and delegating the packets to the different SCION components. The BR forwards the SCION packets between the ASes. We explain the SCION components in more depth in sec. 2.3.

The SIG hosts (orange) contain the dispatcher and daemon for the host connectivity. These components have the same functionality as in the BR hosts. Additionally, a SIG or OPDP SIG is deployed inside each SIG host. The SIG encapsulates the IP traffic and forwards it to the other SIG indicated with the dotted line. However, the actual packet path from SIG-1 to SIG-2 would be via BR-1 and BR-2. We show this path with the red arrows in fig. 6.1. From SIG-2 to SIG-1, the path would be the arrows pointing in the other direction.

We deploy the SIGs inside their own VMs to reduce the possible bottlenecks caused by the BR or CS using CPU resources. The daemon and dispatcher are required for SCION connectivity in the SIG hosts. Whereby we cannot isolate these two components out of the SIG hosts to reduce possible bottlenecks.

We have opted for a virtualised environment to minimise network latency and increased reproducibility. Our setup enables us to measure the performance of the SIG and OPDP SIG in a controlled and isolated environment. Additionally, it allows for easy deployment and reconfiguration for faster development cycles.

The functionality and deployed SCION components in the testbed ASes are the same as for a SCION-based Internet. However, the small topology with only two ASes with one path option is not representative of a SCION-based Internet. In sec. 8.2 we further discuss the representativeness of our testbed.

6.3 Scenarios for Measurement Traffic

We define five scenarios for our measurements and discuss why we chose them. Furthermore, we explain the packet flow path per scenario. We perform all the measurements in chapter 7 on the SIG-1 host from fig. 6.1. After defining the scenarios, we explain the traffic distribution used per experiment and why we chose it.

The five scenarios are:

1. **IP**: A native IP connection between the two SIG hosts (SIG-1 and SIG-2 fig. 6.1). The packet path is directly from SIG-1 to SIG-2 using the 10.0.0.20 and 10.0.0.40 IP addresses. The goal of this scenario is to determine the performance of the underlying IP network. However, there is a difference since this scenario routes

the traffic between the two SIG nodes directly, whereas the other scenarios are routed through the BRs.

2. **SCION**: A native SCION connection between the two SIG hosts. The packet path is from the SIG host via the two BR hosts to the other SIG host (e.g. SIG-1 $\rightarrow$ BR-1 $\rightarrow$ BR-2 $\rightarrow$ SIG-2 in fig. 6.1). The SIG functionality to encapsulate IP packets is not used since the connection utilises native SCION. This scenario aims to determine the performance of the SCION BRs because the BR performance determines the lower limit (RTT) and upper limit (throughput) of the SIG measurements since the SCION packets are routed through the BRs.

3. **US-SIG**: The SIG scenario is between the two SIG hosts with the user space SIG implementation deployed in hosts SIG-1 and SIG-2. This scenario allows us to determine the baseline performance of the existing SIG implementation. We use this scenario to determine the performance increase of our accelerated OPDP SIG. The packet path from SIG-1 to SIG-2 for the SIG scenarios is indicated by the red arrow in fig. 6.1.

4. **TC-SIG1**: The interoperable SIG scenario is identical to scenario three with the difference that we deploy our TC-based SIG in the SIG-1 host. The SIG-1 host is the host where we perform our measurements. That is why we deployed our TC-based SIG in that host since it allows us to measure CPU utilisation of our TC-based SIG whilst interoperating with the user space SIG. This scenario allows us to verify interoperability between our TC-based SIG and the existing user space SIG implementation, and it allows us to determine the performance gain. The interoperability is essential since, in a global SCION deployment, there are no guarantees that the TC-based SIG would be deployed in the other ASes.

5. **TC-SIG1+2**: The accelerated SIG scenario is identical to scenario three and four except that we deploy our TC-based SIG in both SIG hosts (SIG-1 and SIG-2). Deploying the TC-based SIG in both hosts allows us to determine the maximum performance gains made by our implementation.

Now that we have defined the different scenarios we use for our measurements, we will explain the traffic characteristics per experiment type. First, we vary the packet size from 100 bytes to 1300 bytes for the RTT and throughput experiments. We do this to increase our experiments' representativeness since packets on the Internet are not of a fixed size. We perform measurements separately on specific sizes. After the measurement for a specific size is collected, only then do we increase the size. This step-wise increase allows us to determine the performance for

specific packet sizes; however, the downside is that the traffic is not mixed, which is the case for the Internet.

We aim to measure the RTT of the system under load for the RTT experiments. We perform the measurements under load since a deployed SIG by, e.g. an ISP would also be under load, increasing the representativeness of our experiment. The load on the system is achieved by sending Internet Control Message Protocol (ICMP) packets as fast as possible. Therefore the send and receive rates vary depending on the scenario's processing delay. The ICMP ping requests are sent by SIG-1 to SIG-2 where SIG-2 will reply to SIG-1 with ICMP ping responses. Additionally, SIG-1 is able to determine the packet loss by the amount of not received ping responses versus sent ping requests.

We send UDP packets with a specific size from 100 bytes to 1300 bytes as fast as possible for the throughput experiment to determine the maximum forwarding speed. We used UDP for the tests since we ran into performance issues whilst using the Transmission Control Protocol (TCP); we discuss these TCP limitations in sec. 8.3. SIG-1 initiates the throughput tests where SIG-2 has the responsibility of being the throughput test server. The speed tests are performed in duplex, meaning that both SIG hosts transmit packets.

We repeat the RTT and throughput for the specific sizes 100 times to increase accuracy and determine the measurement error.

6.4 Measurement Tools

We used frequently used tools for our measurements. Ping [55] is the industry standard for RTT measurements, whereas, for throughput measurements, it is iPerf [56]. These tools' advantages are that ping is preinstalled on most systems, and iPerf is widely available, increasing the reproducibility of our measurements. Additionally, both tools have built up credibility since they have been widely available and used for RTT and throughput measurements.

For the RTT measurements, we specify the packet size, packet count, flood flag, and preload amount using ping command line parameters. The flood flag enables ping to send a request as soon as it receives a response. The preload flag determines the number of allowed outstanding requests allowing us to increase the load on the system. We empirically determined the preload size to be 65 by increasing the value until incoming requests would be dropped for the IP scenario due to an overflowing receive buffer at SIG-2. We set the packet count to 50,000 for each experiment per packet size. Three factors influence the choice for this value. Firstly the higher the count, the more accurate the measurement. Secondly, the count needs to be high enough for the experiment to have a sufficient duration of multiple sec-

onds to determine the CPU utilisation accurately. Finally, the value must not be too high since all experiments are performed 100 times for 13 packet sizes and would otherwise take too long.

We use iPerf3 [56], the newest version of the iPerf tool, for the throughput measurements. We specify the UDP flag, size flag and an arbitrarily high target bandwidth to perform our experiment. The target bandwidth needs to be set; otherwise, iPerf3 will default to a one megabit per second test. The tool reports the successfully received packet throughput. We use this value as the result of our measurement.

We use pidstat [57] to measure the CPU utilisation during the experiments. Pidstat polls the CPU usage once per second. Therefore each experiment should run for several seconds to achieve an accurate measurement.

The tools used to perform the native SCION experiments are the SCION ping and SCION bwtester [58] tools. The SCION ping command does not have all the functionality the native ping command has. Out of the box, it did not return the minimum, maximum and standard deviation of the RTT nor did it have a flood or preload option. The minimum interval between packets had a minimum of one millisecond. Therefore, we patched the SCION ping command and were able to remove the interval limit and add the missing RTT statistics. This patch allowed us to approach behaviour similar to the native ping, only missing the feature of the preload option. Missing the preload option could result in less load on the system during the measurements.

6.5 Reproducibility

We scripted the VM testbed topology using Vagrant for reproducibility and used VirtualBox [59] as the backend to run the VMs. Vagrant [60] is a tool for building and maintaining portable virtual software environments. The VMs are assigned four CPU cores and 6GB of ram each. We automated the deployment and building process of SCION using Bash scripts for easy deployment and a further increase in reproducibility.

We deployed the testbed on an HP ProLiant DL360 Gen 10 equipped with a Xeon Gold 5122 running at 3.7 GHz and 32 GB of RAM. However, the testbed can be deployed with minimum effort on different hardware. The results' absolute value would differ due to the different computational resources. However, we expect that the relative results of the experiments should still be valid on other testbed hosts. Our experiments found that the SCION components are limited by the CPU, whilst the hosts used less than 1 GB of RAM. Therefore, a testbed host with less available RAM would still be suited for deploying the testbed. We discuss the CPU dependency in sec. 7.3.

Chapter 7

Experiments with a TC-based SIG

In this chapter, we evaluate the performance of our TC-based SIG using the methodology discussed in chapter 6. We start with the RTT experiments (sec. 7.1) for the five scenarios of sec. 6.3 to determine the processing delay gains. Next, we analyse the throughput for the different scenarios of sec. 6.3 to determine the maximum packet forwarding speed (sec. 7.2). Then we analyse the CPU utilisation (sec. 7.3) and finally the packet loss (sec. 7.4) for the RTT experiment. We only discuss the CPU utilisation for the RTT experiment since the utilisation results are comparable for the throughput experiment. We do not discuss the packet loss of the throughput experiments since iPerf3 does not report packet loss.

7.1 RTT

As discussed in chapter 6, we perform the RTT experiment by sending ping packets as fast as possible between the SIG-1 and SIG-2 hosts. We perform the experiment for five different scenarios and different packet sizes. Furthermore, we repeat the measurement 100 times to increase accuracy and to facilitate the calculation of the standard deviation.

The names in the legends of the figures correspond to the bold names of the scenario bullet list in sec. 6.3.

The first RTT results we discuss are the average measured values for the different packet sizes. The value for each size is calculated by taking the average RTT of the 50,000 packets over the 100 measurement repetitions.

We expect the TC-SIG1+2 measurements average RTT to be lower than the US-SIG and TC-SIG1 scenario. The TC-SIG1 scenario measurements should be between the TC-SIG1+2 and US-SIG measurements since the scenario only utilises half the acceleration. IP and SCION RTT values should be lower than the SIG and TC-SIG1+2 scenarios since the SIG protocol is built on top of those network layers.

We show the results in fig. 7.1. TC-SIG1+2 (purple line) has a higher average RTT than the US-SIG (red line) for packet sizes smaller than 500 bytes. Yet, TC-SIG1+2 shows a constant relation to an increase in packet size, whereas US-SIG RTT increases linearly. TC-SIG1 (grey line) shows a linear increase just as the US-SIG. However, the linear increase of TC-SIG1 has a smaller slope than US-SIG resulting in smaller RTTs than US-SIG for larger packet sizes. Yet, at a packet size of 1300 bytes, TC-SIG1 shows similar behaviour to the US-SIG. After inspecting logs and packet captures, we conclude that our TC-based SIG implementation is bypassed. Our implementation is bypassed at a packet size of 1300 bytes due to the fact that the user space SIGs at both hosts (SIG-1 and SIG-2) fragment the encapsulated IP packets over multiple SIG frames. The baseline measurements, IP (dotted blue line) and SCION (dotted brown line) achieve lower average RTTs as expected.

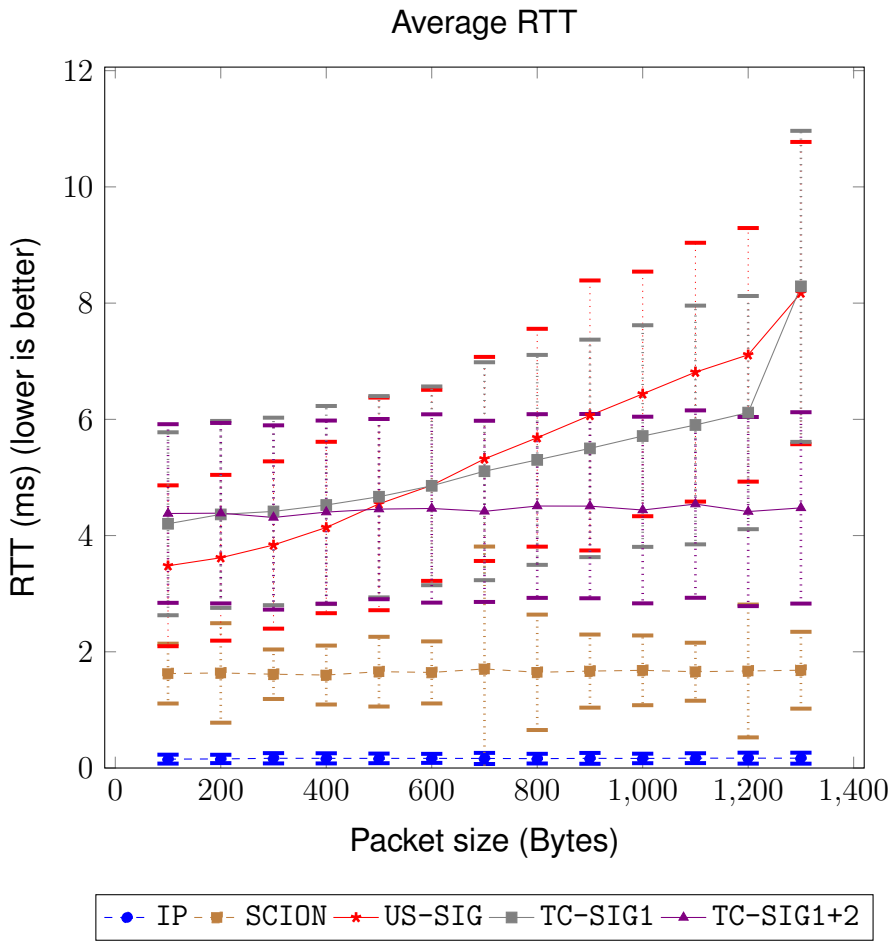


Figure 7.1: Average RTT for flooded ping traffic measured for different packet sizes. Each size is tested with 50,000 packets and repeated 100 times. The error bars indicate the standard deviation. TC-SIG1+2 (purple line) shows a constant relation whereas the US-SIG (red line) shows a linear relation to increase in packet size.

The US-SIG packet size dependence could be caused either by an enforced copy operation of the packet data between user space and kernel space or an operation in the user space program. The constant behaviour of TC-SIG1+2 can be explained by the fact that no operation in our TC-based SIG implementation relies on packet size. The less steep slope of TC-SIG1 results from the constant OPDP SIG and the steeper sloped SIG deployed at the hosts.

We determine the minimum RTT by taking the minimum value per measurement (1 out of 50,000) and averaging that over the 100 experiment repetitions. We expect the same relative behaviour as for the average RTT. The expected behaviour is US-SIG having the longest RTT, then TC-SIG1 followed by TC-SIG1+2, after which SCION and finally IP with the shortest RTT.

We show the calculated minimum RTTs in fig. 7.2. Our expectations were confirmed, except that TC-SIG1+2 (purple line) achieved an even lower minimum RTT than SCION (brown dashed line). A lower RTT is possible for TC-SIG1+2 since the packets do not need to go through the dispatcher. TC-SIG1 shows the same result as the SIG for a packet size of 1300 bytes for the same reason (TC-based SIG bypass) as with the average RTT results.

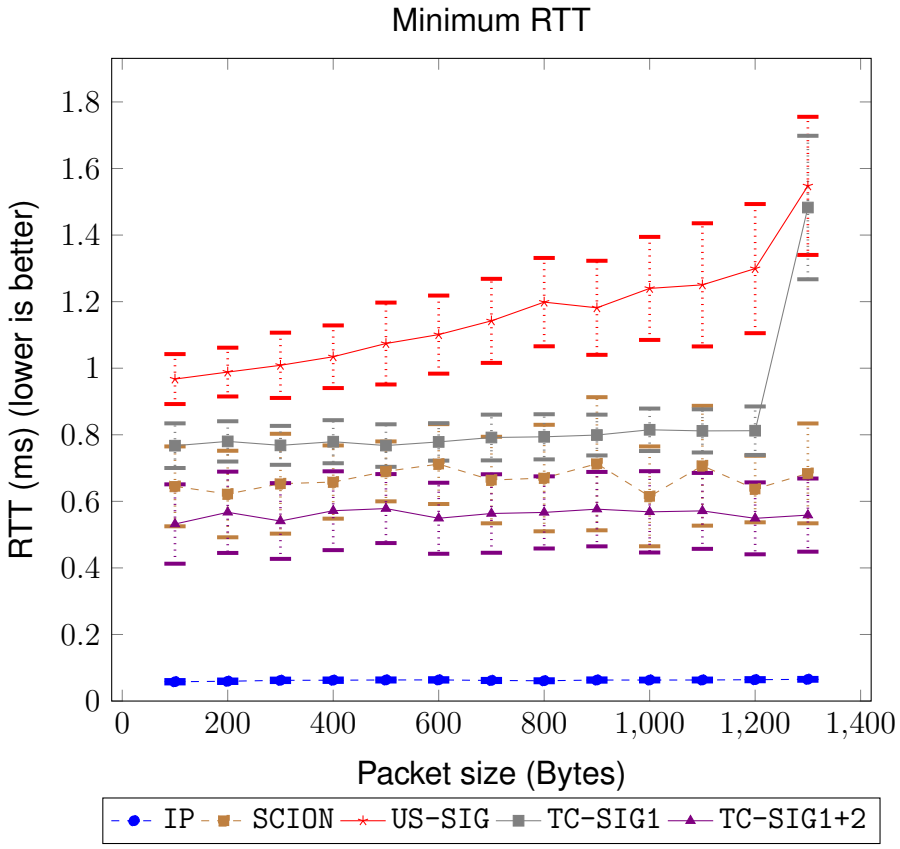


Figure 7.2: Minimum RTT measured for different packet sizes. The value is calculated using the minimum value out of 50,000 packets and averaging it over the 100 measurement repetitions per packet size. The error bars show the standard deviation. Low RTT values are the desired outcome.

The maximum RTT values are determined using the maximum RTT per measurement (1 out of 50,000) and averaged over 100 measurement repetitions. We perform the calculation for each packet size. We expect the same relative distributions for the maximum RTT measurements as we did for the minimum and average RTTs.

In fig. 7.3 we show the maximum RTT results. The measured SCION values are significantly higher than the others. We suspect this is caused by the modified SCION ping tool being less optimised than the IP ping tool. This statement is backed by the fact that none of the SIG scenarios has measured such a high maximum RTT whilst relying on the same SCION network. Besides the SCION results, the other results are in line with expectations, with TC-SIG1+2 achieving lower maximum RTTs than US-SIG and TC-SIG1. TC-SIG1 once again shows behaviour similar to US-SIG for a packet size of 1300 bytes.

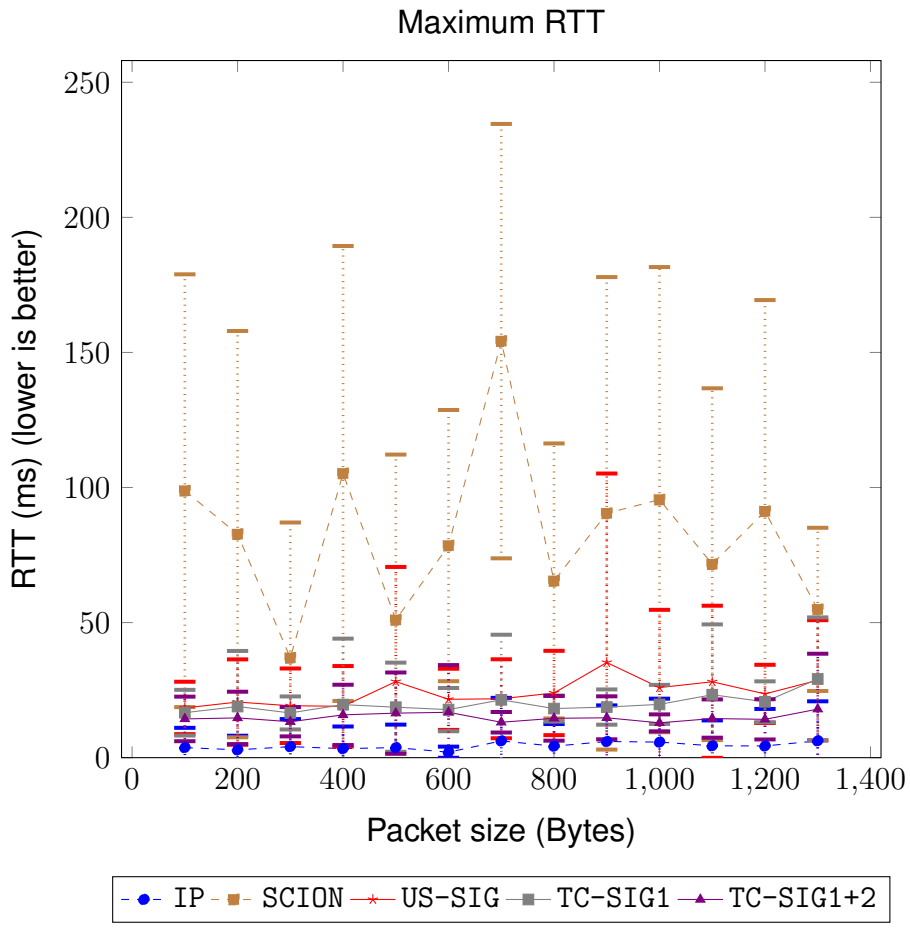


Figure 7.3: Maximum RTT measured for different packet sizes. Each maximum is 1 out of 50,000 packets averaged over 100 measurement repetitions. The error bars show the standard deviation. Low RTT values are the desired outcome. The maximum SCION RTT measurements should not be this high and are probably caused by inefficiencies in the patched SCION ping command.

7.2 Throughput

In this section, we determine the maximum throughput for the scenarios defined in sec. 6.3. We define throughput as the rate of successfully received data at the measurement host (SIG-1). The names in the figure legends correspond to the bold names in the scenario bullet list (sec. 6.3). We determine the throughput in three different ways. Firstly, we analyse and discuss the results achieved from the `scion-bwtester`. Then we discuss the calculated throughput using the RTT experiments data. Finally, we discuss the results of the iPerf3 measurements.

The forwarding speed of the SCION network will be dictated by the SCION BRs and the receive rate of the SCION dispatchers. We perform the `scion-bwtester` measurement to achieve the maximum throughput of the SCION forwarding plane for the different packet sizes. The `scion-bwtester` has packet size and attempted throughput as input parameters and returns the achieved throughput.

In fig. 7.4 we plot the maximum throughput for the different packet sizes. SCION is able to achieve a throughput of roughly 200 megabits per second for a packet size of 1300 bytes. However, SCION is only able to achieve a throughput of roughly 20 megabits per second for a packet size of 100 bytes caused by a limitation in the SCION dispatcher or BR.

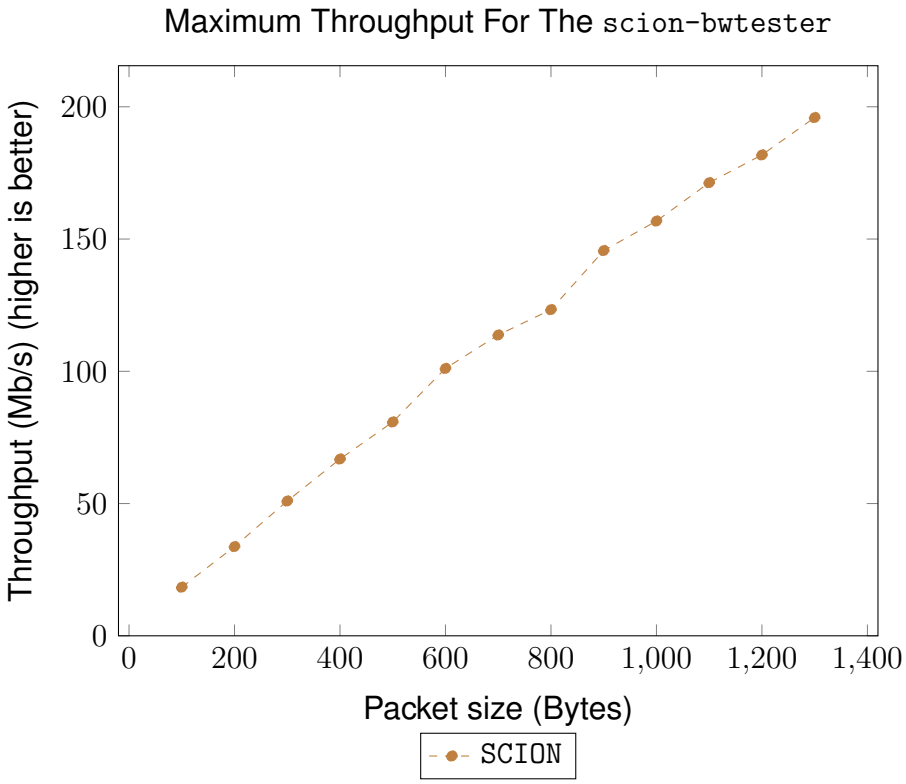


Figure 7.4: Maximum throughput results for the `scion-bwtester` per packet size. This figure indicates the maximum forwarding speed of the SCION network dictated by the SCION BRs

Since we sent the ping packets as fast as possible for the RTT experiment, we can calculate the maximum throughput for that test. We calculate the throughput in megabits per second using the following formula:

$$throughput = \frac{packetcount * packetsize(bits)}{duration(seconds)} * \frac{1}{1024*1024}$$

We expect TC-SIG1+2 to achieve higher maximum throughput than US-SIG. TC-SIG1+2 throughput should be close to 200 megabits per second. That throughput would indicate that the SCION BRs forms the bottleneck, not our TC-based SIG implementation.

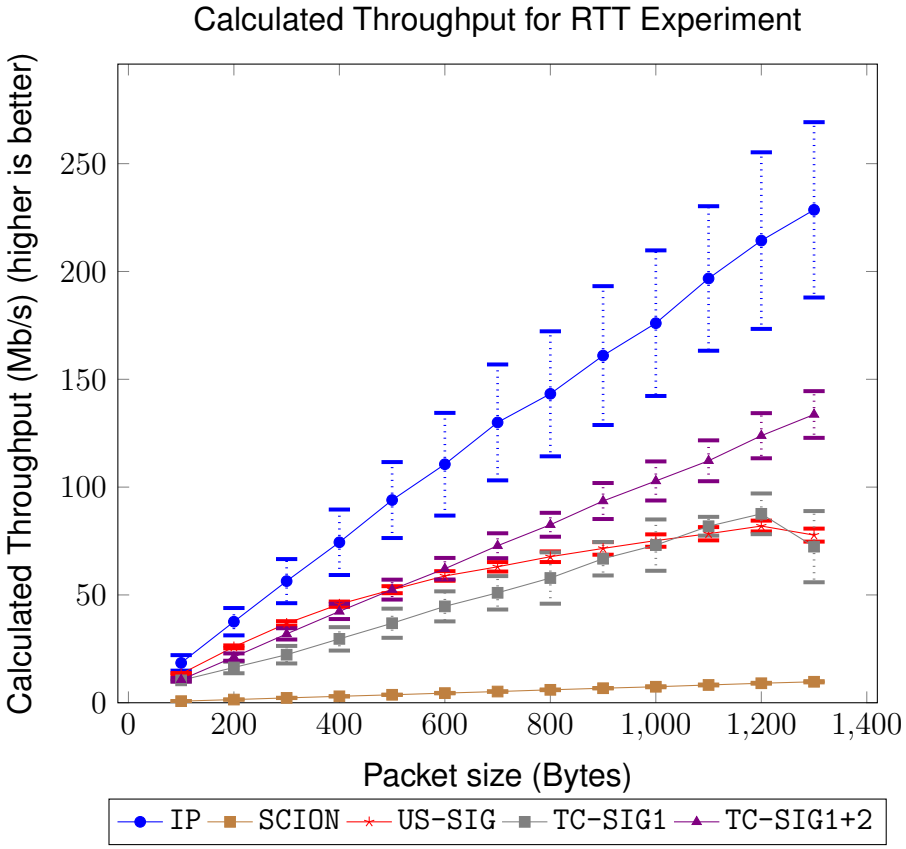


Figure 7.5: Calculated throughput for the RTT experiment. The error bars show the standard deviation calculated using the 100 measurement repetitions. The desired outcome is a high caculated throughput.

In fig. 7.5 we show the calculated throughput. The results show that TC-SIG1+2 (purple line) achieves higher throughputs than US-SIG for packet sizes greater than 500 bytes. However, TC-SIG1+2 does leave room for improvement since it achieves only 130 megabits per second instead of 200 megabits per second. The TC-SIG1

scenario (grey line) lacks behind since it performs worse than US-SIG (red line). The higher packet loss of TC-SIG1 (sec. 7.4) probably causes the lower throughput. The unexpectedly low throughput achieved by the SCION ping test may be caused by the inability to set the flood and preload option flags for the SCION ping command. US-SIG (red line) calculated throughput flattens off for larger packet sizes. We expect the increased processing delay for larger packet sizes (sec. 7.1) to cause the flattening.

Besides the calculated RTT throughput, we also conduct throughput measurements using iPerf3. We expect the results to be comparable to the throughput we got from the RTT experiments. If the results differ, the throughput should be higher since iPerf3 is designed for maximum throughput measurements, whereas the ping tool is not.

Fig. 7.6 shows the measured throughput values. We repeated the measurements 100 times, and the figure shows the average and standard deviation (error bar). TC-SIG1 is left out of the figure since we were not able to perform that test successfully. The UDP data packets would correctly flow between SIG-1 and SIG-2. However, SIG-2 did not correctly transmit the measurement results to SIG-1 due to reordered TCP packets. We discuss the TCP reordering limitations in sec. 8.3. We did not encounter this issue for the TC-SIG1+2 scenario.

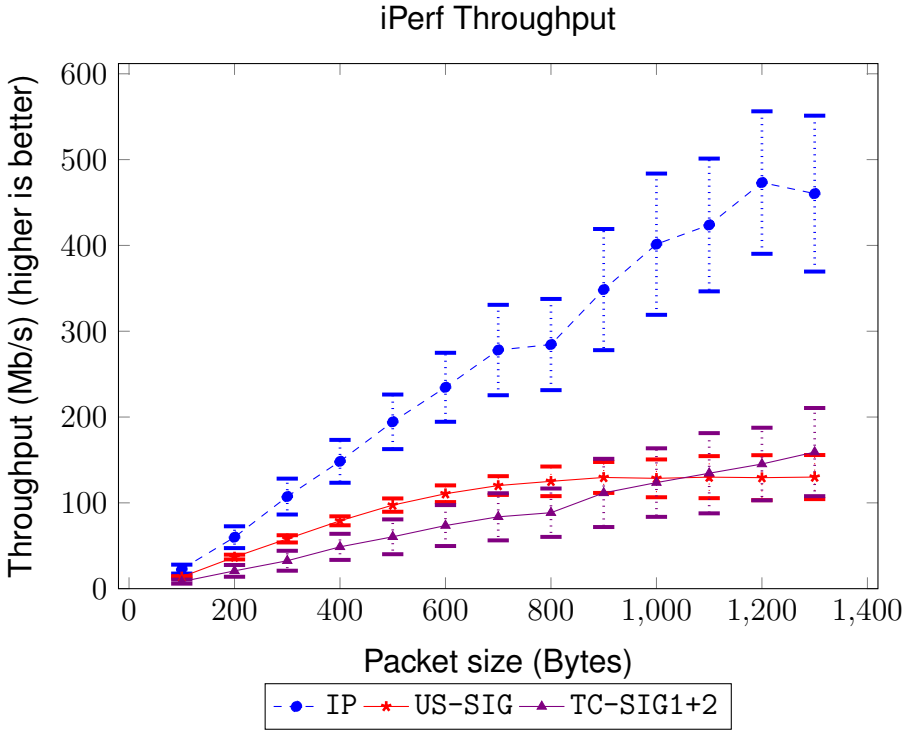


Figure 7.6: Maximum throughput reported by iPerf3 for different packet sizes. The values are averaged over 100 measurement repetitions and the error bars indicate the standard deviation for the 100 repetitions. A higher achieved throughput is the desired value. The error bars show the standard deviation.

The results confirm our expectation that the throughput would increase compared to the calculated RTT throughput. TC-SIG1+2 achieves a maximum throughput of 160 megabits per second and US-SIG 130 megabits per second. TC-SIG1+2 throughput is still lower than the native SCION maximum throughput, indicating that more performance gains can be made. Just as for fig. 7.5, we expect the increased processing delay for larger packet sizes (sec. 7.1) to cause the flattening of the throughput for the US-SIG (red line).

7.3 CPU Utilisation

We measure and analyse the CPU utilisation to determine the SIG's and TC-based SIG's computational load. Additionally, it allows us to determine any bottlenecks in the system.

We measure the CPU utilisation resulting in system, user and wait utilisation percentages. The system utilisation indicates time spent executing kernel code. The user utilisation indicates time spent executing user space code, whereas the wait utilisation indicates time spent waiting for input-output operations [61]. The input-output operations are disk and network operations. The wait utilisation for the tests we performed is presumably caused by networking since we do not write to disk. The maximum CPU utilisation measurable is 400% since we measure the utilisation per core and add up the values.

We start by discussing the user space SIG CPU utilisation. To be clear, each scenario has a user space SIG for TC-SIG1+2 and TC-SIG1; it runs parallel to the TC-based SIG. We expect the user space SIG implementation to be a bottleneck for the US-SIG scenario. Therefore, the CPU utilisation should be roughly 100% for that scenario, assuming that the user space SIG runs on a single CPU core. In contrast, we expect TC-SIG1+2 to hardly utilise CPU resources.

Fig. 7.7 shows the measured CPU utilisations for the IP (blue), US-SIG (red), TC-SIG1 (grey) and TC-SIG1+2 (purple) scenario. The user, system and wait percentages are graphed on top of each other, resulting in the total CPU utilisation. The red bars show that the US-SIG scenario depends on and heavily utilises the CPU. Additionally, we see that for the TC-SIG1 scenario, the CPU is utilised with a peek at a packet size of 1300 bytes. The utilisation of the TC-SIG1 scenario for packet sizes 100 to 1200 bytes is explained by our TC-based SIG implementation passing on messages to the user space SIG implementation. At the 1300 bytes packet size, the CPU utilisation indicates that all received packets are passed to the user space implementation, which aligns with the earlier RTT results.

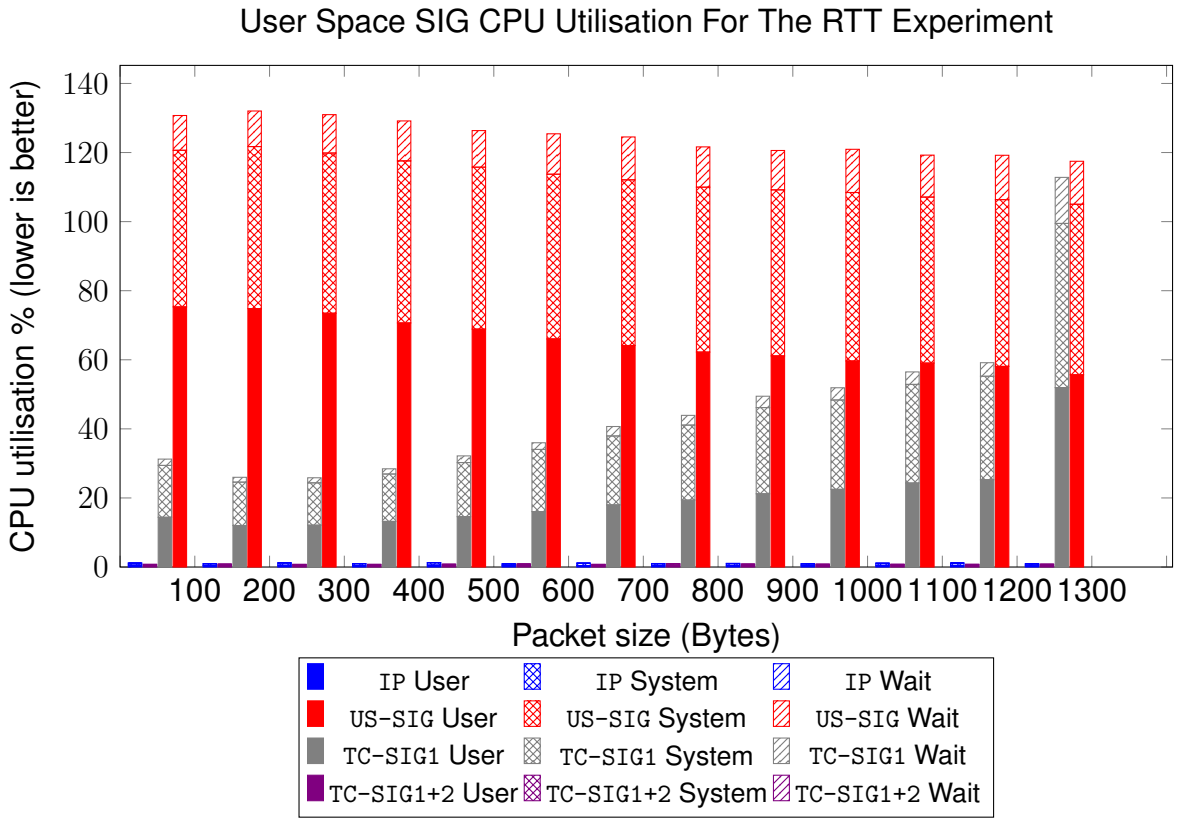


Figure 7.7: The measured CPU utilisation for the user space SIG for different payload sizes. The utilisation is specified in system, user and wait CPU utilisation percentages. The desired outcome is to have minimal CPU utilisation for the user space SIG. The figure clearly shows that TC-SIG1+2 does not utilise CPU resources.

Next, we discuss the dispatcher CPU utilisation. We consider the dispatcher since it is responsible for receiving the incoming packets and forwarding them to the user space SIG (see sec. 2.3). This responsibility makes the dispatcher likely to form a bottleneck for the user space SIG implementation. However, the dispatcher should not utilise any CPU resources for the TC-SIG1+2 scenario since the TC-based SIG implementation is attached directly to the network interface. Therefore, it sees and potentially redirects the packets before the dispatcher.

We show the measured CPU utilisation in fig. 7.8. Indeed the dispatcher does not utilise the CPU for TC-SIG1+2 (purple bars). The dispatcher does use CPU resources for the TC-SIG1 scenario (grey bars), following the same reasoning as the user space SIG CPU utilisation. Interestingly the dispatchers' system CPU utilisation for the US-SIG scenario (red crossed bar) increases linearly with the packet showing the same trend as the average RTT. Receiving and copying packets of increased size from kernel space to user space would explain this increase in system CPU utilisation.

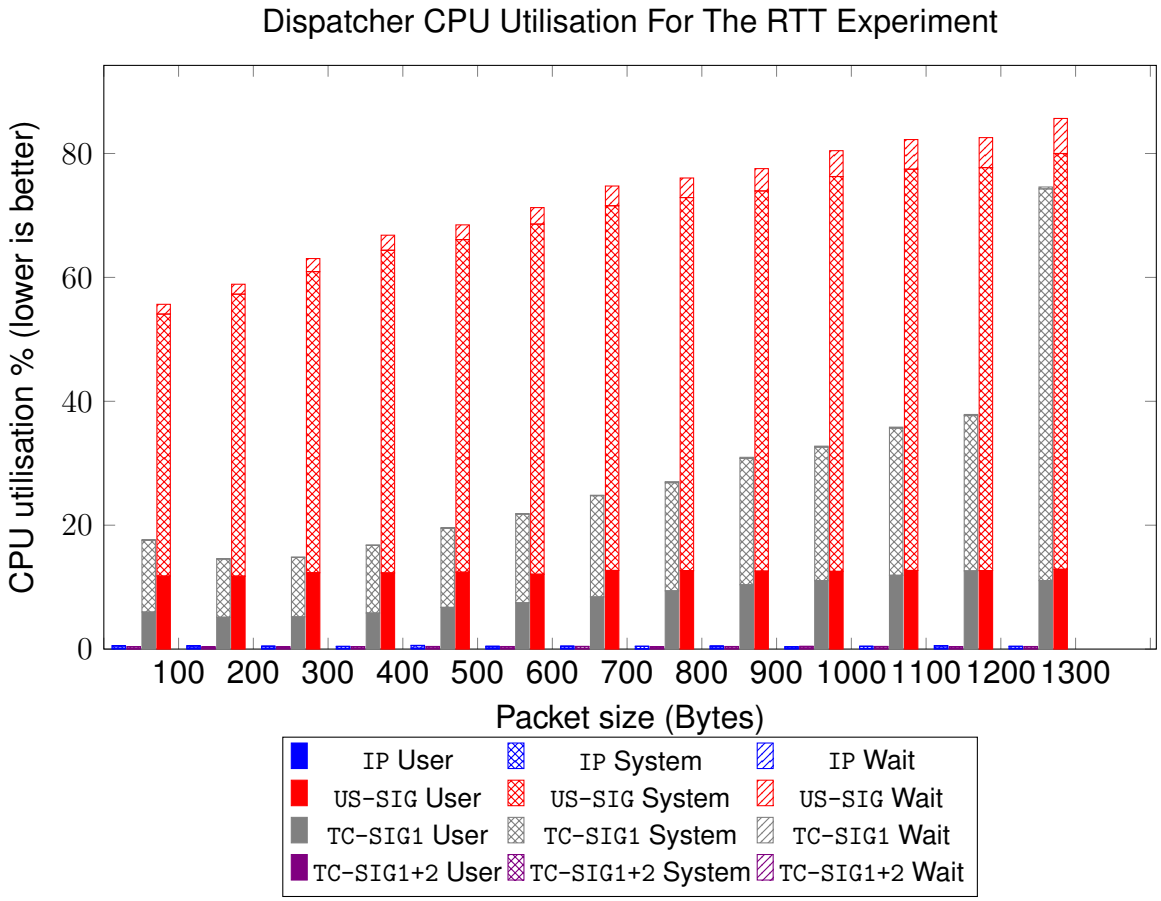


Figure 7.8: The measured CPU utilisation for the dispatcher for different payload sizes. The utilisation is specified in system, user and wait CPU utilisation percentages. The desired outcome is to have minimal CPU utilisation for the dispatcher. TC-SIG1+2 does not utilise any CPU resources. However, the system CPU utilisation increases linearly with the packet size for the US-SIG scenario.

Finally, we discuss the CPU utilisation of the ping command. The CPU utilisation of the ping command indicates whether the measurement tool (ping) forms the bottleneck or other components in the network (e.g., the user space SIG). We expect the utilisation to be 100% for the IP and TC-SIG1+2 scenarios since ping is a single-threaded process. 100% utilisation would indicate that ping is the bottleneck for these scenarios. On the other hand, we expect the CPU utilisation to be lower for the TC-SIG1 scenario and the lowest for the US-SIG scenario. These utilisations would indicate that ping is a bottleneck for the IP and TC-SIG1+2 scenarios. In contrast, the user space SIG implementation would be a bottleneck for the other scenarios.

Fig. 7.9 shows the measured CPU utilisation for the ping command. As expected, the IP scenario has approximately 100% CPU utilisation (blue bars). However, the TC-SIG1+2 scenario (purple bars) has a significantly smaller CPU utilisation. The utilisation for the TC-SIG1+2 scenario indicates that a component within the network forms a bottleneck by limiting how fast the ping command can send. This component

could be our TC-based SIG implementation, the SCION BR (in the BR-1 or BR-2 host) or both. TC-SIG1 and US-SIG scenarios show the expected limited CPU utilisation since the user space SIG bottlenecks them.

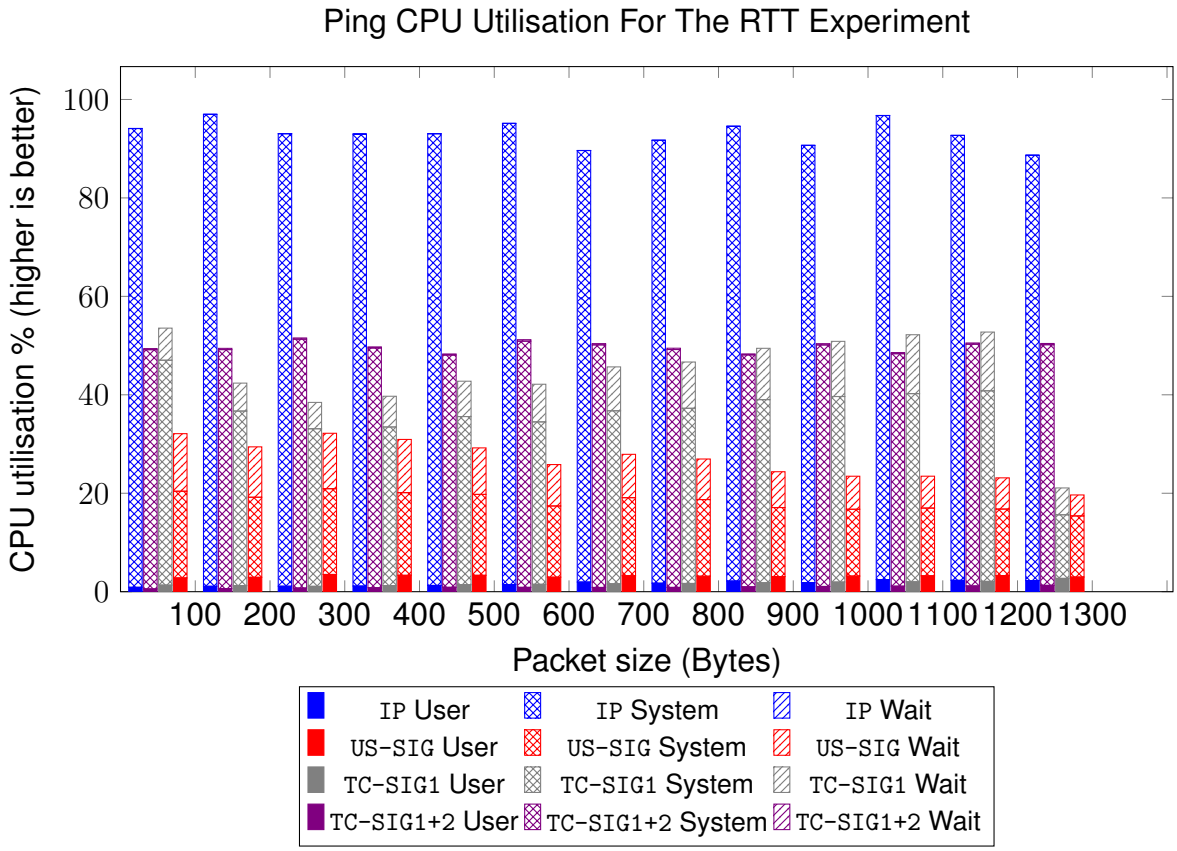


Figure 7.9: The CPU utilisation for the ping command for different payload sizes. The utilisation is specified in system, user and wait CPU utilisation percentages. The desired outcome is to have maximal CPU utilisation for the ping command since that indicates that the ping command is forming the bottleneck and not the SCION components.

7.4 Packet Loss

Packet loss is of interest to us since packet loss could indicate problems in the system and is a waste of networking resources. The value should be low to 0 for all scenarios since there is no congestion on our network.

Fig. 7.10 shows the packet loss measured during the RTT experiment. The TC-SIG1 scenario (grey line) shows a relatively high packet loss. After investigating the user space SIG logs on the SIG-2 host, we discovered the user space implementation dropped the packets. The logs did not contain the reason why the implementation dropped the packets.

A possible reason for the packet loss could be sequence number errors caused by the reordering of packets. The reordering could occur due to the parallel deploy-

ment and hybrid functionality of the user space SIG and TC-based SIG implementation in the SIG-1 host. The parallel deployment of the user space SIG and TC-based SIG in the SIG-1 hosts would not form a problem for the TC-SIG1+2 scenario since the TC-based SIG in the SIG-2 host would decapsulate the packets without dropping them due to sequence number errors.

The relatively high packet loss could also be caused by that the fact that the SIG-1 host using our TC-based SIG is capable of sending the packets faster than the not accelerated SIG-2 host can process.

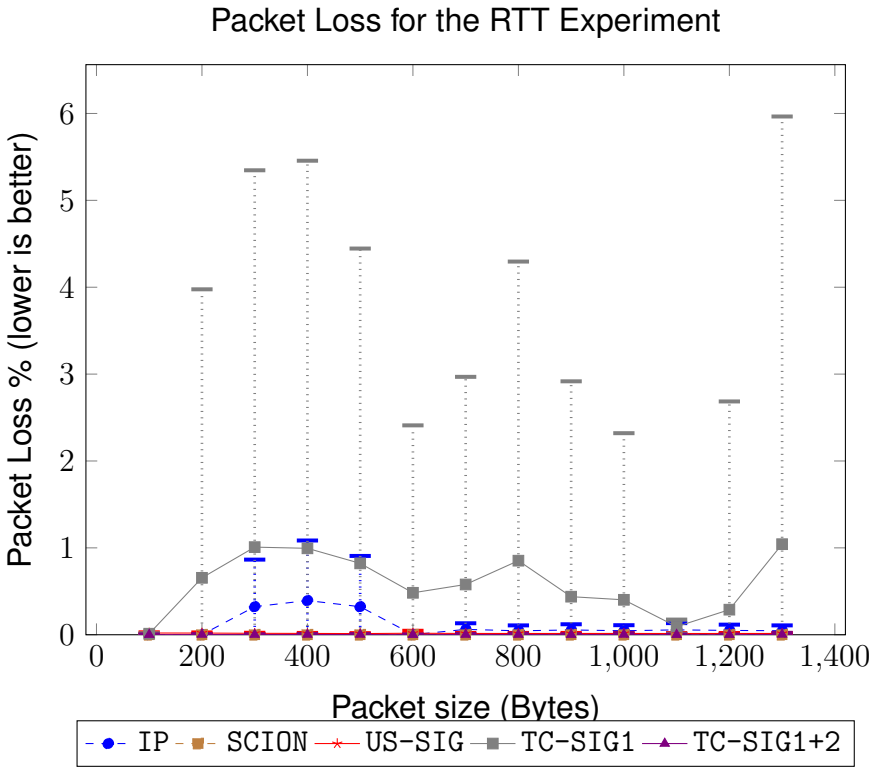


Figure 7.10: The measured package loss for the RTT experiment averaged over the 100 measurement repetitions. The desired outcome is a low packet loss value. The error bars show the standard deviation. The interoperable OPDP SIG scneario shows relatively high packet loss caused by dropped packets by the user space SIG implementation in the SIG-2 host.

Chapter 8

Discussion

In this chapter, we use our work on the OPDP implementations (chapter 5) and our experiments (chapters 6 and 7) to reflect on the P4 hardware constraints (sec. 8.1), our testbed design (sec. 8.2) the limitations of our OPDP SIG prototype (sec. 8.3), and the SIG framing protocol (sec. 8.4).

8.1 P4-based SIG

In sec. 5.2, we mentioned that we did not manage to realise a P4 implementation of the SIG because of the constraints of the P4 hardware. These constraints are not specific to the Intel Tofino but apply to all high-speed pipelined networking switches. The operations that the framing protocol requires (e.g., reassembly) discussed in sec. 8.4 limit the class of devices that can fully support the SIG.

The required SIG operations like packet reassembly usually occur at end-hosts and are non-normal behaviour for network switches. The Intel Connectivity Research Program forum confirms this statement by mentioning that fragmentation, defragmentation and packet reassembly are not in line with the high-speed design in the ”Research and Publication Review Guidelines”. Additionally, subject matter experts also confirmed this via an e-mail consultation.

As mentioned in sec. 5.2 a hybrid design could be possible for the Tofino. The design would run the user space implementation on the CPU of the Tofino-based system and forward the packets which require unsupported operations to user space, like reassembly. However, we do not believe this would be an efficient use of the Tofino hardware since congestion on the CPU port could occur since the CPU (user space) would quickly become the bottleneck. The physical ports do not only operate with higher throughput; they also outnumber the CPU port since there is only one CPU port and many physical ports.

8.2 Testbed

We focused on getting a first indication of the performance of an OPDP-based SIG. We therefore setup a simple testbed (see sec. 6.2). Our testbed only contains two ASes. As a result, the path header consists of only two HFs making up one fixed path. Therefore, we did not test our prototype in an environment with varying paths or HFs. We expect that our prototype would still be functional since it does not act on the HFs. It treats the HF information as a binary blob it uses to encapsulate the packets. The TTL time for eBPF map entries set in the user program would dictate how fast the program would adapt to path changes. A lower TTL would result in a more responsive system at the cost of letting more packets through to the user space implementation.

In addition, we perform all our tests in a controlled environment with direct virtual links between the hosts. This controlled environment has the advantage of easy development and deployment, but it does not simulate congestion or packet loss caused by other actors on the network. A controlled environment without congestion is desired to perform accurate and reproducible measurements for our specific purpose to compare US-SIG, TC-SIG1, and TC-SIG1+2. However, it does not represent a deployment on the Internet where congestion and packet loss could occur.

Our completely virtualised measurement setup influences the test results compared to deploying our TC-based SIG prototype directly on hardware. However, we expect the results to become more favourable since we could utilise hardware offloading for our OPDP using the same source code as the virtual deployment. Additionally, there would be less overhead due to the virtualisation.

8.3 Experiments with a TC-based SIG

As mentioned in sec. 7.2, we ran into packet reordering problems for the interoperable OPDP SIG scenario. After analysing packet captures, we found that packets handled by our OPDP SIG could overtake packets forwarded to the user space SIG. The reordering of the packets creates several problems. Firstly, the reordering of SIG frames results in packets being dropped by the user space SIG receiving the frames due to sequence number errors. Secondly, reordered packets causes significant performance degradation for TCP [62]. We found the performance degradation to be so severe that iPerf3 would not return the results correctly.

We set the UDP checksum to 0 as a workaround for the kernel dropping packets due to an invalid checksum. This workaround was a temporary solution to achieve measurable results within our limited time. Correctly calculating the checksum instead of bypassing the check is strongly advised before implementing our prototype

in a production environment. This calculation will negatively influence the performance of our prototype since more computation is required. We expect the negative influence to be minor, but more required computation decreases performance per definition. Therefore, our accelerated prototype should still outperform the current SIG implementation.

Our prototype has a variable `MAX_SIZE` which defines the maximum size of the binary data header it can handle. This limitation is due to the maximum allowed stack size of eBPF programs. Packets with a header size exceeding the `MAX_SIZE` would still be processed correctly by the user space SIG implementation. To be clear, our solution does not drop any packets. The incompatible packets would not be fast-tracked. In sec. 9.3 we describe an approach to possibly remove this limitation.

We made a first attempt at optimising the SIG implementation. Although the results seem promising, we see room for more improvement. We discuss the possible improvements in sec. 9.3.

8.4 SIG Framing Protocol

In this section, we discuss how the design choices by the SCION developers of the SIG framing protocol limit OPDP deployments.

We will first discuss the SIG framing protocol design choices. The maximum header stack size is 1150 bytes: 14 (Ethernet) + 40 (IPv6) + 8 (UDP) + 1024 (SCION) + 8 (SCION/UDP) + 16 (Frame) + 40 (IPv6) = 1150, a significant amount of the common 1500 Maximum Transmission Unit (MTU) of the Internet. Typically a header would contain far fewer HFs than the maximum supported, resulting in a reduced packet overhead. However, the fact that typically headers contain fewer Hop Fields (HFs) does not relax the hardware constraint since the hardware should support the maximum header size defined by the protocol specification.

Aggregating IP packets and encapsulating multiple packets inside a single SIG frame has downsides. Grouping multiple packets results in a less straightforward decapsulation. In our P4 implementation of the SIG we learned that not all hardware packet processing pipelines can duplicate incoming packets, a requirement for decapsulating multiple IP packets from a single SIG frame. Furthermore, the operation adds latency to the first packet since it waits for the other aggregated packets.

The design decision of fragmenting packets with a MTU which is too large over multiple SIG frames results in required reassembly operations at the receiving side. This reassembly is expensive to perform at high line rates due to the necessary buffering at the receiving side. Sometimes this buffering is not even possible since the packets are processed in different pipelines within a hardware device. A device

with a many-to-one connection acting as an aggregation point results in a significant traffic load that should be processed efficiently with high throughput. In that scenario, it is not possible to buffer packets to reassemble them later. Network switches neither implement fragmentation nor defragmentation. IPv4 implements fragmentation inside routers, however RFC4963 [63] mentions reassembly errors at high data rates. IPv6 has moved the fragmentation to the source host [64] altogether. The SIG Framing protocol could function in the same manner.

Additionally, the sequence numbering might be unnecessary. The transport protocols of the encapsulated IP packets take care of the retransmission of missing packets. TCP has this functionality built-in, and UDP application developers consider possible packet loss in their design. The sequence numbering seems unnecessary if we compare the SIG framing protocol, a form of tunnelling, to existing IP tunnelling. The sequence number is required for the expensive reassembly operations described above since packets can get reordered. The downside of the included sequence numbering is the additional complexity it adds to the protocol.

Chapter 9

Conclusions, Recommendations and Future Work

In this chapter, we start by concluding our research by answering the research questions (sec. 9.1). Next, we provide our recommendations for protocol creators and OPDP technology developers (sec. 9.2). Finally, we discuss the possible future work to be performed (sec. 9.3).

9.1 Conclusions

Our research aimed to accelerate the performance of the SIG using state-of-the-art OPDPs. We have successfully created an open-source TC-based OPDP SIG prototype. The prototype is the first attempt at accelerating the SIG. Our prototype is user space implementation agnostic since it is self-learning because it acquires the required path data by analysing outgoing SIG frame packets.

Besides the OPDP prototype, we have also created an automated virtualised testbed setup for OPDP SIGs performance measurements. We performed detailed measurements and compared the performance of the US-SIG and TC-based SIG. In our analysis, we have found that our prototype increases the performance whilst reducing the load on the CPU, enabling the SIG to be better suited for high-throughput scenarios.

We will now answer our research questions one by one.

`[RQ1] To what extent can state-of-the-art OPDPs be used to accelerate the performance of a SIG?`

Based on our engineering and measurement work, we provide a first indication that a high-performance SIG is possible using a TC-based OPDP. However, we also learned that OPDP languages and platforms have several limitations that make it a

non-trivial exercise to implement and maintain the SIG for different OPDPs.

`[RQ2] What test setup do we require to perform the performance measurements on the SIG and OPDP SIG?`

The goal of the testbed is a minimal measurement setup to evaluate the performance of an OPDP SIG, with RTT and throughput as performance parameters. We established a minimal required measurement setup of two ASes with four virtualised hosts. Two hosts are used within each AS. In one host of the AS we deploy a SCION BR and in the other host we deploy a SIG. We have automated the creation of our virtualised measurement setup to increase reproducibility. Our virtualised testbed's network latency is neglectable compared to the SIG processing delay, which allows us to compare the processing delay between the TC-based SIG and US-SIG.

`[RQ3] Which OPDP technology is best suited to improve the SIG performance?`

Out of the three OPDPs we researched, we were able to create a functioning accelerated SIG prototype with a TC-based OPDP. We found the hardware-based P4 OPDP and the kernel-based XDP OPDP to be unsuitable for accelerating the SIG. P4 is unsuited since the required fragmentation and reassembly operations for the SIG are not supported. We found XDP not to be suited since we could not attach the XDP program to the ingress side of the interfaces, which is a requirement for redirecting the packets. We determined these incompatibilities by attempting to create prototypes for all OPDPs.

`[RQ4] What performance gains can be achieved using state-of-the-art OPDPs?`

In our measurements, we achieve a 64% throughput increase whilst reducing the average latency by 45% and the CPU utilisation by 99% for the TC-based OPDP SIG compared to the US-SIG. The CPU utilisation decrease is calculated for the utilisation of the user space SIG. Our prototype is a first attempt at accelerating the SIG using OPDP and has room for optimisations.

9.2 Recommendations

In this section, we provide recommendations for network protocol designers, eBPF developers and programmable network hardware vendors.

- Network protocol designers must consider that hardware switching has different constraints than software switching. We recommend avoiding fragmentation and defragmentation by leaving this up to the end host as discussed in

sec. 8.4. Variable field lengths should also be avoided if possible [51], and in general, the protocol should be as simple as possible. Software switching performs satisfactorily for some use cases, but the network layer should be compatible with hardware switching to scale.

- Our research gives a use case for attaching XDP programs (see sec. 5.3) to the egress side of interfaces and for ingress to ingress forwarding. We recommend to the XDP developers to support these features since that would allow us to implement the SIG using XDP. There is already work performed on this feature [54], but we highlight the need for this feature.

- We recommend that programmable network hardware vendors implement an encryption P4 extern (custom hardware helper) like there already is for CRC calculations. For the use case of the SIG, performing MAC calculations was not required. However, there is a requirement for per-packet signing for SCION packet forwarding in the SCION BR chapter 2 and probably more future Internet protocols. So to keep up with these developments, we see the need to efficiently perform encryption inside the network pipelines.

9.3 Future Work

Our research is the first attempt at optimising and measuring the performance of the SIG. In this section, we will review a few possible improvements for our OPDP prototype and measurement setup and discuss future measurement ideas.

9.3.1 OPDP Implementations for the SIG

The current prototype implementation is not yet fully optimised. A deeper analysis comparing the performance of different eBPF map types and helper functions for our prototype could further improve the performance. Besides experimenting with the map types, the key used for the map could be optimised, e.g. by splitting the map into two maps, one for IPv4 and one for IPv6, reducing the required key size.

Additionally, the Ethernet egress TC program could be replaced by a patched version of the user space SIG. The patched SIG fills the eBPF maps with the required path information. Removing the Ethernet egress TC program dependency most likely increases the performance with the downside of not being user space agnostic anymore.

A different approach to improving performance could be implementing the OPDP SIG using VPP or DPDK. VPP is the technology Anapaya uses for their closed-

source components. VPP and DPDK were outside the scope of our thesis due to time constraints.

Besides performance improvements, we also have some possible improvements in functionality for our prototype. Support for VLAN parsing should be added because this is currently not supported. As discussed in sec. 8.3 there is a `MAX_SIZE` limitation on the SCION headers out prototype can handle due to the eBPF stack size limit. This limitation could be resolved by storing the headers directly into an eBPF map.

9.3.2 Measurement Setup

An improvement for our testbed would be to increase the scale and topology. Currently, the testbed exists of only two ASes. This could easily be increased on the condition that there are enough computational resources available on the testbed host. A different take on the measurement setup would be to deploy our OPDP prototype in the SCIONLab [11] and measure its performance in a real-life scenario.

9.3.3 Measurements

An interesting future measurement would be to deploy our prototype on actual hardware to see how the performance relates to our virtualised scenario. Additionally, a test using a Netronome NIC [39] with TC offloading would enable us to see the performance gains on our prototype using offloaded TC. In sec. 7.1, we found that our prototype has a constant latency for increasing packet sizes, whereas the current user space SIG implementation increases linearly with the packet size. Therefore, it would be interesting to perform tests with jumbo frames (9000 bytes) to determine if the latency payload size relations stay the same. Our measurements are currently performed for packets of a specific size for multiple different sizes. However, a test with packets of mixed sizes would further increase representativeness.

Bibliography

[1] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, U. Kc Claffy Caida Uc, S. Diego, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named Data Networking," *ACM SIGCOMM Computer Communication*, 2014. [Online]. Available: http://trac.tools.ietf.org/group/irtf/trac/wiki/icnrg

[2] NGP, "GR NGP 009 - V1.1.1 - Next Generation Protocols (NGP); An example of a non-IP network protocol architecture based on RINA design principles," *ETSI GR NGP*, 2019.

[3] D. Barrera, L. Chuat, A. Perrig, R. M. Reischuk, and P. Szalachowski, "The SCION internet architecture," *Communications of the ACM*, vol. 60, no. 6, pp. 56–65, 6 2017.

[4] SCIONLab, "SCION IP Gateway (SIG) - SCIONLab Tutorials," 2020. [Online]. Available: https://docs.scionlab.org/content/apps/remote_sig.html

[5] L. Chuat, M. Legner, D. Basin, D. Hausheer, S. Hitz, P. Müller, and A. Perrig, *The Complete Guide to SCION From Design Principles to Formal Verification*. Zürich: ETH Zürich, 2 2022.

[6] Anapaya, "SCiON-Internet - The New Way to Connect," 2022. [Online]. Available: https://www.anapaya.net/scion-the-new-way-to-connect

[7] scionproto, "scionproto/scion: SCION Internet Architecture." [Online]. Available: https://github.com/scionproto/scion

[8] Cloudflare, "What is BGP hijacking? — Cloudflare." [Online]. Available: https://www.cloudflare.com/learning/security/glossary/bgp-hijacking/

[9] D. Pinto, "What will happen when the routing table hits 1024k? — APNIC Blog," 3 2021. [Online]. Available: https://blog.apnic.net/2021/03/03/what-will-happen-when-the-routing-table-hits-1024k/

[10] J. P. Wagner, D. Hausheer, and M. Gartner, "Improving Packet Processing Speed on SCION Endhosts," Ph.D. dissertation, Otto-von-Guericke-Universität Magdeburg, Magdeburg, 9 2021.

[11] SCIONLab, “SCIONLab.org,” 2020. [Online]. Available: https://www.scionlab.org/

[12] J. de Ruiter and C. Schutijser, “Nieuwe internetinfrastructuren: een inleiding tot SCION — SIDN Labs,” 2022. [Online]. Available: https://www.sidnlabs.nl/nieuws-en-blogs/nieuwe-internetinfrastructuren-een-inleiding-tot-scion

[13] “The Go Programming Language.” [Online]. Available: https://go.dev/

[14] SCION, “SCION IP Gateway Framing Protocol Specification — SCION documentation,” 2021. [Online]. Available: https://docs.scion.org/en/latest/protocols/sig.html

[15] “ASICs at the Edge.” [Online]. Available: https://blog.cloudflare.com/asics-at-the-edge/

[16] “Switch Architectures.” [Online]. Available: https://www.grotto-networking.com/BBSwitchArch.html

[17] “P4 – Language Consortium.” [Online]. Available: https://p4.org/

[18] Y. Lixiang, L. Wenfeng, C. Dazhao, L. Tianhou, W. Ruobing, S. Qi, and F. Ke, *The ART of LINUX KERNEL DESIGN*. CRC Press, 2014.

[19] R. P. França, M. Peluso, A. C. B. Monteiro, Y. Iano, R. Arthur, and V. Vieira Estrela, “Development of a kernel: A deeper look at the architecture of an operating system,” in *Smart Innovation, Systems and Technologies*, vol. 140. Springer Science and Business Media Deutschland GmbH, 2019, pp. 103–114.

[20] “Networking — The Linux Kernel documentation.” [Online]. Available: https://www.kernel.org/doc/html/latest/networking/index.html

[21] eBPF.io, “What is eBPF? An Introduction and Deep Dive into the eBPF Technology,” 2021. [Online]. Available: https://ebpf.io/what-is-ebpf#verification

[22] NLnetLabs, “XDP - The NLnet Labs Blog,” 2022. [Online]. Available: https://blog.nlnetlabs.nl/tag/xdp/

[23] D. R. Barach and E. Dresselhaus, “Vectorized Software Packet Forwarding,” 6 2011.

[24] FD.io, “FD.io - The Universal Dataplane.” [Online]. Available: https://fd.io/

[25] H. Bi and Z.-H. Wang, “DPDK-based Improvement of Packet Forwarding,” *ITM Web of Conferences*, 2016.

[26] DPDK, “Home - DPDK.” [Online]. Available: https://www.dpdk.org/

[27] C. Hesselman, P. Grosso, R. Holz, F. Kuipers, J. H. Xue, M. Jonker, J. de Ruiter, A. Sperotto, R. van Rijswijk-Deij, G. C. Moura, A. Pras, and C. de Laat, “A Responsible Internet to Increase Trust in the Digital World,” *Journal of Network and Systems Management*, vol. 28, no. 4, pp. 882–922, 10 2020.

[28] “ONF Programmable Networks Projects - Open Networking Foundation.” [Online]. Available: https://opennetworking.org/onf-sdn-projects/

[29] “P4˜16˜ Portable Switch Architecture (PSA),” 2021. [Online]. Available: https://p4.org/p4-spec/docs/PSA.html

[30] “GitHub - p4lang/behavioral-model: The reference P4 software switch.” [Online]. Available: https://github.com/p4lang/behavioral-model

[31] The P4.org Architecture Working Group, “P4 16 Portable Switch Architecture (PSA),” P4.org, Tech. Rep., 2018.

[32] Intel, “Intel® Tofino™ Series Programmable Ethernet Switch ASIC,” 2022. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html

[33] ——, “P4 16 Intel ® Tofino™ Native Architecture - Public Version,” Intel, Tech. Rep., 3 2021.

[34] V. Gurevich and A. Fingerhut, “P4 16 Programming for Intel® Tofino™ using Intel P4 Studio™,” 2021. [Online]. Available: https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Vladimir-Gurevich-Slides.pdf

[35] S. McCanne and V. Jacobson, “The BSD Packet Filter: A New Architecture for User-Level Packet Capture,” in *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, ser. USENIX’93. USA: USENIX Association, 1993, p. 2.

[36] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. Câmara, and L. F. Vieira, “Fast packet processing with EBPF and XDP: Concepts, code, challenges, and applications,” *ACM Computing Surveys*, vol. 53, no. 1, 2 2020.

[37] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, “The eXpress data path: Fast programmable packet processing in the operating system kernel,” in *CoNEXT 2018 - Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. Association for Computing Machinery, Inc, 12 2018, pp. 54–66.

[38] "AF_XDP — The Linux Kernel documentation." [Online]. Available: https://www.kernel.org/doc/html/v4.18/networking/af_xdp.html

[39] J. Kicinski and N. Viljoen, "eBPF Hardware Offload to SmartNICs: cls bpf and XDP," Netronome Systems, Cambridge, Tech. Rep., 2016.

[40] D. Chandrashekar and M. Riaz, "Performance Comparison of SCION with Routed IP on Virtual Machines," AALTO UNIVERSITY, Tech. Rep., 11 2020.

[41] M. Gartner, "Improving SCION Bittorrent with efficient Multipath Usage," Ph.D. dissertation, Otto-von-Guericke-Universität Magdeburg, 2020.

[42] L.-C. Schulz and D. Hausheer, "Offloading SCION Packet Forwarding to XDP BPF," Otto-von-Guericke-University Magdeburg, Tech. Rep., 2022. [Online]. Available: https://legacy.netdevconf.info/0x13/session.

[43] C. Neukom, "High-Performance File Transfer in SCION," Ph.D. dissertation, ETH Zurich, 2020. [Online]. Available: https://doi.org/10.3929/ethz-b-000443419

[44] NLnetLabs, "GitHub - NLnetLabs/XDPeriments: Example programs fot the Journeying into XDP blogs," 2022. [Online]. Available: https://github.com/NLnetLabs/XDPeriments/

[45] O. Yoachimik, "A deep-dive into Cloudflare's autonomous edge DDoS protection," 2021. [Online]. Available: https://blog.cloudflare.com/deep-dive-cloudflare-autonomous-edge-ddos-protection/

[46] D. Lee, "Faster Packet Processing in Linux: XDP," SOSCON, Tech. Rep., 2019.

[47] "NetFPGA." [Online]. Available: https://netfpga.org/

[48] K. Součková, A. Perrig, and B. Rothenberger, "FPGA-based line-rate packet forwarding for the SCION future Internet architecture," Ph.D. dissertation, ETH Zurich, 9 2019.

[49] "GitHub - AnotherKamila/scion-p4netfpga," 2019. [Online]. Available: https://github.com/AnotherKamila/scion-p4netfpga

[50] "GitHub - SIDN/p4-scion: A P4 implementation of the SCION protocol." [Online]. Available: https://github.com/SIDN/p4-scion

[51] J. de Ruiter and C. Schutijser, "Next-Generation Internet at Terabit Speed: SCION in P4," in *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '21. New

York, NY, USA: Association for Computing Machinery, 2021, pp. 119–125. [Online]. Available: https://doi.org/10.1145/3485983.3494839

[52] K. van Hove, “P4 Hybrid Routing for Next-Generation Networks Experimentation based on Open Source Software Defined Routing,” University of Twente, Tech. Rep., 10 2021.

[53] “tcpreplay(1) - Linux man page.” [Online]. Available: https://linux.die.net/man/1/tcpreplay

[54] “Add support for XDP in egress path [LWN.net].” [Online]. Available: https://lwn.net/Articles/813406/

[55] “ping(8) - Linux man page.” [Online]. Available: https://linux.die.net/man/8/ping

[56] “iPerf - iPerf3 and iPerf2 user documentation.” [Online]. Available: https://iperf.fr/iperf-doc.php

[57] “pidstat: Report statistics for tasks - Linux man page.” [Online]. Available: https://linux.die.net/man/1/pidstat

[58] “Bandwidth Tester - SCIONLab Tutorials.” [Online]. Available: https://docs.scionlab.org/content/apps/bwtester.html

[59] “Oracle VM VirtualBox.” [Online]. Available: https://www.virtualbox.org/

[60] “Vagrant by HashiCorp.” [Online]. Available: https://www.vagrantup.com/

[61] “top(1) - Linux manual page.” [Online]. Available: https://man7.org/linux/man-pages/man1/top.1.html

[62] J. Bellardo and S. Savage, “Measuring Packet Reordering,” *University of California at San Diego*, 2002.

[63] J. Heffner, M. Mathis, and B. Chandler, “IPv4 Reassembly Errors at High Data Rates,” *IETF*, 7 2007. [Online]. Available: https://www.rfc-editor.org/info/rfc4963

[64] S. Deering and R. Hinden, “Internet Protocol, Version 6 (IPv6) Specification RFC2460,” *IETF*, 12 1998. [Online]. Available: https://www.rfc-editor.org/info/rfc2460

Appendix A

List of Acronyms

AES Advanced Encryption Standard

API Application Programming Interface

AS Autonomous System

ASIC Application-Specific Integrated Circuit

BGP Border Gateway Protocol

BMv2 Behavioural Model version 2

BPF Berkeley Packet Filter

BR Border Router

COLIBRI Collaborative lightweight inter-domain bandwidth-reservation infrastructure

CRC Cyclic Redundancy Check

CS Control Service

DAG Directed Acyclic Graph

DDoS Distributed Denial of Service

DNS Domain Name System

ELF Executable and Linkable Format

EPIC Every Packet Is Checked

FPGA Field Programmable Gate Array

GPL General Public License

HF Hop Field

ICMP Internet Control Message Protocol

ICRP Intel Connectivity Research Program

IP Internet Protocol

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

ISA Instruction Set Architecture

ISD Isolation Domain

ISP Internet Service Provider

JIT Just in Time

LPM Longest Prefix Matching

MAC Message Authentication Code

MTU Maximum Transmission Unit

NDA Non-Disclosure Agreement

NIC Network Interface Card

OPDP Open Programmable Data Plane

PCB Path-segment Construction Beacon

PCFS Packet Carried Forwarding State

PSA Public Switch Architecture

QoS Quality of Service

qdisc queueing discipline

RTT Round Trip Time

SCION Scalability, Control, and Isolation On Next-Generation Networks

SDE Software Development Environment

SIDN Stichting Internet Domeinregistratie Nederland

SIG SCION IP Gateway

SLA Software License Agreement

SOSCON Samsung Open Source Conference

TC Traffic Control

TCP Transmission Control Protocol

TNA Tofino Native Architecture

TRC Trust Root Configuration

TTL Time To Live

UDP User Datagram Protocol

VM Virtual Machine

VPP Vector Packet Processing

WCET Worst-Case Execution Time

XDP eXpress Data Path