ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master Thesis

Network Security Group, Department of Computer Science, ETH Zurich

FPGA-based line-rate packet forwarding for the SCION future Internet architecture

by Kamila Součková

Spring 2019

ETH student ID: 16-932-584
E-mail address: skamila@ethz.ch

Supervisors: Prof. Dr. Adrian Perrig
Benjamin Rothenberger

Date of submission: August 31, 2019

Abstract

The SCION future Internet architecture is designed to provide route control, failure isolation, and explicit trust information for end-to-end communication. Its viability has already been demonstrated in production, but to work at Internet scale, it is necessary to make it perform at high bandwidth, reduce costs, and evaluate its suitability for eventual implementation in hardware. This project implements a SCION border router capable of line-rate packet forwarding, running on an FPGA target. We utilise P4, an emerging language for programming packet forwarding planes. This enables target independence, a quicker development cycle, and a more modular approach. Based on this work, we suggest changes to the SCION data plane that allow for more efficient processing in hardware.

Acknowledgements

I would like to thank my advisors Prof. Dr. Adrian Perrig and Benjamin Rothenberger for both their guidance and their trust in my self-directed work. Their advice and help with connecting me with various experts were invaluable, and necessary for the success of this project.

This work would not be possible without Prof. Nele Mentens and Jo Vliegen of KU Leuven, who provided us with an optimised AES module, and without David Sidler of the Systems Group at ETH Zurich, who helped to integrate it.

I am also very grateful to Philip Paeps for tirelessly answering my barrage of hardware-related questions. His help was essential for making progress quickly despite not having previous hardware design experience.

Many thanks for reviewing the draft of this thesis go to Stephen Shirley. His ever-present criticism was surprisingly constructive.

Finally, I wish to thank Andrés Hamann for bearing with me and for being an excellent rubber duck. Andrés is awesome.

Contents

1 Introduction

1.1 Why SCION?

Today's Internet, based on IP and BGP, was conceived in a very different environment, with very different requirements and assumptions. Meant to be a research network, it assumed that access was not widespread (as only large institutions could afford a connection), that all entities connected to it trusted each other (and had out-of-band means to deal with misbehaviour), and that it would not grow large enough for outages to become difficult to manage. Considering these facts, it has scaled surprisingly well, but nowadays we are living in a very different world.

Today, everyone is connected to the Internet and everyone has their data online. With the huge number of connected entities, many of which have conflicting interests, trust can no longer be global and implicit. Users want control over their data, ISPs want control over their services, and states want to protect their infrastructure from outside actors. At this scale, convergence after path failure takes too long, and large-scale outages due to BGP route leaks are happening almost weekly TODO cite. Furthermore, attacks on the Internet infrastructure, e.g. DDoS attacks or BGP route hijacks, are becoming more and more common.

In the future, the Internet will become even more critical. Automation (such as drones, self-driving cars, or remotely operated robots) will depend on the availability and security of the Internet. As more and more data moves online, trust will become even more of an issue. We will need ways to satisfy new requirements such as geo-fencing, which cannot be adequately fulfilled with the current Internet architecture. And as malicious actors get more and more sophisticated and powerful, their attacks will get even harder to stop. Therefore, we will need to move to an Internet architecture that is resilient to both unintentional failures and malicious attacks, can provide explicit trust information to the end users, and gives the end users as well as institutions control over whom they trust and what happens with their data.

There have been may attempts to fix the IP+BGP-based Internet TODO[cite RPKI, BGPsec], but patching an architecture designed with entirely different assumptions cannot solve all the problems [2]. Additionally, the IP-based approach also has scalability issues due to requiring very large routing tables, and this is bound to get worse with future growth. Therefore, the SCION project chooses to build a clean-slate Internet architecture, with the goals of scalability, route control, failure isolation, and explicit trust information in the design.

1.2 SCION right now

Thanks to a well-thought-out design, as well as "real-life" considerations such as deployment, transition strategy, and engagement with industry, SCION is currently running in production in several locations TODO[cite]. Additionally, a planet-wide research network for testing SCION is running thanks to SCIONLab TODO[cite SCIONLab paper]. These use cases have proven that SCION is in fact suitable for real-world usage. In the near future, several ISPs are likely to start offering SCION connections to their customers. Therefore, considerations like throughput, cost-effectiveness, and energy efficiency are of great interest to many in the SCION community.

The existing implementations of the SCION infrastructure, in particular the routers, run as applications on standard servers. This has enabled rapid development, as well as flexibility, and it will continue to be the primary approach until the protocol stabilises. However, with the current technologies, a software-based approach cannot achieve throughput above several tens of gigabits per second.[1] Therefore, large-scale deployments would require many such routers, which is not cost-effective. From the research perspective, whether SCION can even in principle run at very high throughputs, and what hurdles must be overcome, are still open questions.

Especially thanks to the real-world deployments, SCION is now mature enough to lend itself to implementing in hardware. Though the current version of the protocol is very unlikely to be the final one, it would be useful to get an initial hardware-based implementation, mainly in order to evaluate the protocol's suitability for hardware. A successful hardware-based implementation would also be a good answer to the "can it run at line rate?" question, which would further increase the probability of deployment with commercial ISPs.

1.3 Contributions of this project

The first contribution of this project is **implementing a proof-of-concept border router capable of forwarding SCION packets at line rate**. We target a NetFPGA SUME board, which (in the ideal case) allows us to forward up to 40 Gbps. The border router is the crucial building block of the SCION data plane. Therefore, having a high-speed prototype will open the doors to large-scale data transfers, as well as prove that infrastructure providers (such as ISPs) will be able to scale to meet the needs of many customers without incurring very high costs.

Additionally, we keep the work target-independent (as much as possible), and thereby create a "library for line-rate SCION" for a variety of targets and use-cases.

[1] For example, the latest performance report of DPDK by Intel (http://fast.dpdk.org/doc/perf/DPDK_19_05_Intel_NIC_performance_report.pdf) shows a maximum of 20 Gbps in the best case, with highly optimised hardware, for a simple LPM-based forwarding algorithm.

An implementation targeting an FPGA[2] is a good first step for evaluating the feasibility of building a hardware SCION router, as much of the design process is the same with FPGAs as with ASICs[3]. Therefore, the second contribution of this work is **confronting SCION with hardware design expertise, and opening the doors for building faster, more efficient designs in the future**. The knowledge gained in the process of implementation on the NetFPGA allows us to better understand the requirements of running SCION in custom hardware, and predict whether a very high-speed and energy-efficient ASIC implementation could be built. As being able to run in hardware is a hard requirement for the success of any network-layer Internet protocol, knowing more about this question is crucial to the development and deployment of SCION.

Last but not least, with the knowledge gained during the implementation, we have been able to **provide suggestions for the SCION protocol that would result in more efficient hardware implementations**. Running in hardware has been a consideration in the design of SCION from the beginning, but an actual implementation, rather than abstract reasoning, is likely to uncover opportunities for further improvement. Doing this early, before SCION has become widely deployed, allows us to implement any changes needed without having to worry too much about backwards-compatibility and enter the "real-life deployment" sphere with an already optimised protocol.

[2] Field-Programmable Gate Array: a chip with many programmable logic blocks and programmable interconnects that can be used for hardware prototyping.

[3] Application-Specific Integrated Circuit: an integrated circuit customised for a particular use. Since ASICs are custom-built, they can be highly cost-effective and energy-efficient. The initial development costs of ASICs are very high, so a sufficiently large deployment is needed to offset those.

2 Background and Related Work

In order to achieve this project's aims, we need to work in the intersection of several fields: among others network security, network protocols, software engineering, and hardware design. This chapter introduces what concerns us in each of the relevant areas, and the following chapters build on this information.

2.1 SCION data plane

SCION is a clean-slate Internet architecture designed for route control, failure isolation, and explicit trust information for end-to-end communication. For a comprehensive presentation of the SCION architecture, see [4]. In this section, I will introduce the aspects of the SCION data plane relevant for this work.

2.1.1 Packet-carried forwarding state

In order to scale without the need for state on routers (specifically large routing tables), SCION puts all information needed for packet forwarding into the packet header. In particular, the user-selected path that the packet needs to take through the network is present in the header, as a stack of *hop fields* (HFs). This means that the packet headers are variable-size and can be quite large – this is the trade-off SCION makes to avoid large and potentially inconsistent routing tables in routers.

Each HF in the path corresponds to an AS-level hop. It encodes the ingress and egress interface in its AS, verification information used by the AS to check that the HF has been issued by the AS and has not been modified, the expiration time for this HF (which must be checked), and some additional information. The exact HF format is described in [4]. In order to forward the packet, the router only needs to look at the "current" HF, i.e. the one meant for this AS. Therefore, the SCION header contains a pointer to the current HF. This is initially set to the first one in the path, and every egress border router increments it after processing it, so that a router can find its HF without parsing the whole path.

2.1.2 Hop field verification

Allowing the end host to assemble the path is great for giving the end host control, but for various reasons ASes may need to enforce routing policies. Therefore, end hosts must not be allowed to create "any" paths, only ones compliant with the policies of the ASes involved. In order to enforce this at line rate and without

needing much state on the routers, SCION requires the HFs to contain verification information. For standard SCION, this information is a cryptographic MAC keyed by an AS-specific secret key.[1] The AS distributes the HFs including this pre-computed MAC during path discovery, and therefore if a hop field in the end-host-created path contains a valid MAC, it is proof that the end host got this HF from the creator AS. The MAC for a HF is computed over itself and the HF occurring previously in the path (see figure 2.1), so that the end host cannot arbitrarily join multiple valid path segments into a single path if it has not been explicitly allowed. The MAC verification is based on AES-CMAC TODO[cite the RFC], because with dedicated hardware, computing AES can be done very efficiently at line rate. TODO[would be great to have something to cite here – look for what Adrian ended up citing when he asked me what to cite :D] Because the forwarding is otherwise rather computationally inexpensive, achieving a sufficiently fast HF verification routine will be one of the more interesting aspects of our implementation.

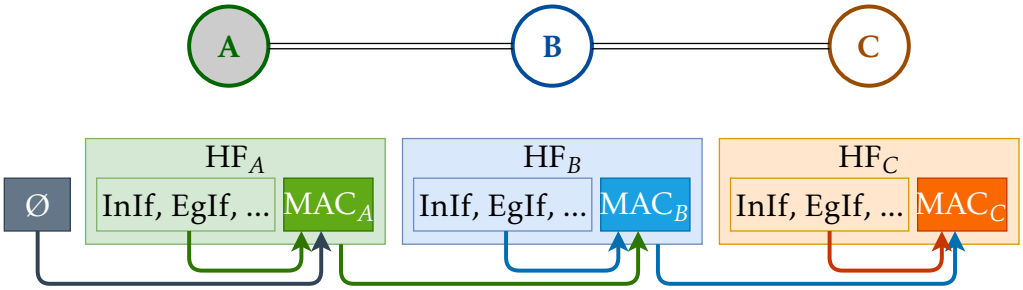

Figure 2.1: TODO somebody should write something here.

2.1.3 IP overlay

In order to enable inter-operability with existing networks, SCION packets may use an IP/UDP overlay. TODO Steve says I should replace that sentence with: " Scion is primarily an inter-domain protocol; as such it runs as an overlay on top of existing network protocols, such as UDP/IP." The software border router currently requires this, as it uses a UDP socket to get SCION packets. Therefore, our implementation must accept SCION within IP, and must send out packets wrapped in the correct IP/UDP overlay, i.e. the next hop's IP address and UDP port. In the future, SCION will also run directly on top of L2 (when given a point-to-point link). Our implementation should therefore either support both, or make it easy to add direct SCION over Ethernet links in the future.

[1] In addition to "standard SCION", SCION also supports packets with end-to-end bandwidth reservations, and source-authenticated packets. For these, the verification information is different. In this work, we only implement the "standard SCION" mode, and will mention the other modes only very briefly.

2.1.4 Putting it all together: Border router behaviour

In accordance with the above, our border router must do the following:

1. Extract and parse the current HF according to the offset information in the offset header.
2. Verify the ingress interface and the incoming IP overlay.
3. Check that the HF is not expired.
4. Validate the HF MAC.
5. Update the HF offset information.
6. Update the IP overlay header.
7. Select the output port corresponding to the egress interface and send the packet out.

(Note that for simplicity, handling of traffic from/to the local AS is omitted here.)

2.1.5 Related work

The reference software router is available at https://github.com/scionproto/scion/. Since this runs in software and is not optimised for performance, it can only achieve a few Gbps on normal hardware.

We are aware of ongoing work by various groups: one creating a high-speed software implementation (several tens of Gbps), and two hardware-based implementations: one targeting an FPGA and programmed in Verilog, and one targeting a programmable switch (due to the limitations of the hardware, this one will support "SCION with caveats"). At the time of writing, none of this work has been published.

2.2 High-speed packet forwarding

There are multiple approaches that enable high-speed packet processing.

On the software side, the main approaches are eBPF and DPDK.

eBPF is an open-source in-kernel user-programmable virtual machine, originally introduced in BSD and later ported to GNU/Linux. It allows the user to implement their own packet processing rules without losing performance due to copying to user space and back. The performance of eBPF on modern hardware is on the order of 10 Gbps.

DPDK, the Data Plane Development Kit, is a set of data plane libraries and network interface card (NIC) drivers that enables fast packet processing in user space. It abstracts the hardware and software environment in order to enable

target-independent implementations. Its performance on modern hardware is also on the order of 10 Gbps.

To achieve higher efficiency and cost-effectiveness, using a hardware approach is often preferred. For mature protocols (i.e. ones that are very unlikely to change), it is preferred to build a custom ASIC, i.e. a chip with the exact circuitry required to perform the function. Examples of ASICs used in networking are Ethernet controllers (also on NICs in PCs) or IP prefix matching tables in dedicated router hardware.

ASICs have very good performance characteristics and they are cost-effective at large quantities, but the one-time development costs of ASICs are extremely high and if the protocol changes, an ASIC cannot be reprogrammed. Therefore, for new protocols such as SCION, we need to look at programmable hardware. One flavour is programmable network switches: a fixed structure with some programmable elements. An OpenFlow switch is one example: the processing pipeline is fixed, but one can to an extent choose which fields are matched and what action is performed based on the match. Recently, more flexible programmable switches based on the P4 language (see 2.4) have become available. These have a fixed architecture, i.e. the general "shape" of the processing pipeline: an example would be "parser → checksum verification → match-action pipeline → checksum calculation → deparser". The functionality within each of these blocks is programmable to an extent, but still limited by the capabilities of the underlying hardware. In exchange for these limitations, the hardware can be optimised and can achieve throughput up to several Tbps.

Unfortunately, currently available programmable switches do not have all capabilities required for SCION: notably, the high-performance switches miss a cryptographic functions module, without which we cannot efficiently implement the AES-based HF MAC. Further, programmable switches do not provide much insight into the internals of the execution (and any such insight would anyway be limited to the specific switch), so with these, we would not be able to discover the bottlenecks and find opportunities for optimising the SCION protocol for hardware in general.

A good trade-off between flexibility and performance, which additionally shows more of the hardware design process, are **FPGAs**. Designing for an FPGA gives us the full power (and responsibility) of designing fully custom hardware: we are not limited to whatever the manufacturer included, and if we need an AES module, we can add an AES module. The full programmability of the circuit comes at the cost of performance: assuming a good design, a single FPGA can handle on the order of 100 Gbps (which is about an order of magnitude above software, and about an order of magnitude below programmable switches and ASICs).

When designing for an FPGA, we have visibility into the various parts of the system, and can see which parts are "difficult" to implement efficiently. Because the design process for FPGAs is the same as the first steps for creating an ASIC, it is reasonable to assume that things problematic on FPGAs would also be problematic

on an ASIC. This allows us to make good guesses about the difficulties in hardware in general, and therefore propose generally useful changes to the SCION protocol.

Due to hardware availability, the flexibility vs. performance trade-off, as well as the insights that can be gained in the design process, we have chosen an FPGA for our work.

2.3 NetFPGA

The NetFPGA is a family of open-source hardware and software for prototyping of network devices. They are FPGA-based devices with several high-speed Ethernet ports. Being FPGA-based implies that these devices are fully programmable. Therefore, we will be able to include all the required functionality, such as parsing of SCION packets, AES-based MAC computation, timestamp validation, and communication with the control plane.

Their newest (as of 2019) product, the NetFPGA SUME [5], has 4 ports with 10G Ethernet, thus enabling a total throughput of 40 Gbps. It incorporates Xilinx's Virtex-7 690T FPGA, which is a high-end FPGA suitable for large projects. Due to this, and the availability of a P4 toolchain (see 2.4) targeting the NetFPGA SUME, we have chosen to use this hardware for our project.

2.4 P4

P4 is a programming language for specifying packet processing pipelines. Building on previous software-defined networking efforts, it allows switches to be reprogrammed by the end user. It is target-independent, and therefore can in principle be used for programming anything from software switches to high-performance ASICs with minimal changes to the program. It is general enough to express the processing of almost any reasonable network protocol, including SCION, and high-level enough to spare us from target-dependent implementation details such as manually managing pipelining.

For programming FPGAs, P4 enables a much more high-level approach compared to traditional hardware description languages (HDLs) such as VHDL or Verilog. While the traditional HDLs are more general and give the user far more control, P4's focus on a single purpose (packet processing) enables it to abstract away many common features of packet processing pipelines, and lets the programmer focus on the interesting features. P4 programs generally consist of a **parser** stage (encoded as a state machine), a **match-action pipeline** (encoded as an imperative program that makes use of **tables**, i.e. structures written by the control plane that look up a key and execute an associated action such as modifying a header or setting the output port), and a **deparser** that outputs the modified packet header.

P4's choices about which features are not supported (such as unbounded loops or arbitrary array access) enable the P4 compiler to guarantee line rate processing ("if it compiles, it will run fast"). Our initial examination of P4 suggests that it provides appropriate abstractions, and that the benefits of a faster development cycle, maintainability for software engineers, and target independence outweigh the loss of fine control for the use case of building a SCION router prototype.

P4 is a relatively new language (the specification for $P4_{16}$, the version we use, was published in May 2017). Therefore, the first interesting question is P4's suitability for describing non-traditional network protocols such as SCION. Beyond those concerns, the toolchains for targeting specific hardware, and especially the NetFPGA, are at this point experimental and not well tested (as we will see in chapter 4). Nevertheless, the abstractions offered by P4 may be worth it, especially because the "new, untested technology" problems will likely get less severe in the future, and therefore having a P4 implementation for SCION will become more and more useful over time.

2.5 Introduction to designing for FPGAs

The design process for an FPGA is very different from software engineering. Though this is a difficult task, in this section we attempt to provide a **brief** overview of it from the perspective of a software engineer.

2.5.1 How to think about programming FPGAs

In software, the fundamental unit of a program is the instruction: one creates programs by describing the instructions, or steps, that happen one after another and transform the data as desired. Therefore, in software engineering, data is processed serially, and it "stays in one place, while the operations performed on it change in time". The programmer thinks in terms of events: data arrives, then we do something with it. In this way, software engineering is reactive and adaptive to the data: the CPU performs different functions depending on the runtime data. The typical limiting factor in software engineering is also time: we want our programs to run fast, and therefore need to limit the number of instructions.[2] This is a useful way to think about software, because a PC has one (or a few) general-purpose CPUs, which execute different and data-dependent instructions in every step.

In contrast, in FPGAs (and hardware design in general), the circuit always stays the same. One cannot perform an addition and then a multiplication without

[2] This is of course a simplified description, with networked systems and parallel programming, as well as the growing ratio of CPU to I/O speeds, being the obvious discrepancies. However, we believe that fundamentally, this a good, even if not complete, picture of how software engineers think about programming, and it is useful to contrast this with how to think about FPGAs.

moving the data, because an addition circuit always adds, and cannot multiply. The function of each part of the circuit is fixed when programming the FPGA, and stays the same during the execution, unable to react to what the data might "need". Thus, in order to perform more than just the simplest functions, data must move to a different part of the circuit in order to proceed to the next step of processing. However, in contrast to CPUs performing only one instruction at a time, all the different parts of a circuit are available all the time, and many different (or the same!) tasks can be performed at the same time without competing for CPU time.

This leads to the idea of *pipelining*: if the data has to move through different parts of the circuit for each processing step, it would be a waste to keep the rest of the circuit idle. Therefore, in order to process more data, we should start processing the second item before we're done processing the first one. For example, if the processing has 2 stages, we can feed new data to the first stage on every cycle, even though each item spends 2 cycles in the system. Thus, pipelining can be used to cheaply increase throughput (and has no effect on latency). Further, when there are complex steps in the processing, the circuit can often be duplicated, so that (with some buffering) the pipeline can keep moving despite the slow step.

From this, it follows that the limiting resource in FPGA design is circuit area: more complex processing does not cost us throughput (and thus, for most practical purposes, is not "slower"), but adds area. FPGAs have limited area directly proportional to the cost of the FPGA, and thus we need to limit ourselves in area usage. Furthermore, large circuit area brings forth issues with signal propagation: the speed of signal propagation is finite, and thus depending on the physical layout of the circuit, synchronisation and signal timing requirements may become difficult to satisfy. Problems with satisfying timing requirements require design optimisation and, if that is not enough, increasing the allotted time for each step and therefore decreasing throughput.

In summary, unlike with software, where programmers "think in time and make things happen", hardware designers need to "think in space and put things where they need to be". The layout and connections of the circuit determine the paths of the data, and thereby the operations performed on it. Data locality becomes an important concern, operations that take an unpredictable amount of time are to be avoided, and the pipeline needs to keep going at all costs. In return, for many high-performance applications, the flexibility and parallel processing power available through wise use of the available area is well worth it.

2.5.2 Overview of the design process

In software engineering, going from source code to a running program is usually a one-step process: one compiles the code and gets a runnable binary. For FPGAs, this is quite a bit more involved. This is the outline of the process:

1. **Logic description**: writing the source code for the logic of the project. The logic of the program is described using a *hardware description language*

(HDL), such as Verilog or VHDL. Compiling from a higher-level language, such as Bluespec [3] or, in our case, P4, is also an option. One can also link various self-contained modules (similarly to using libraries in software development) into their logic. An example of that in this project is our usage of a generic AES module (the "library"), which we then employ to check SCION HF MACs (the "business logic"). After writing the logic of the program, a *behavioural simulation* should be run to check logic correctness: this simulates the program on a high level, with logical inputs and outputs.

2. **Describing the full design**: taking the various modules (including the one(s) written in step 1) and connecting them to form the complete system. This includes connecting standard components for interacting with "the real world" outside of the FPGA. In our case, the router code from step 1 handles L1 and up, but connects (over a simple standardised bus) with a module for Ethernet that handles the physical layer, as well as with modules for PCI and other communication with control plane. The description of modules and connections from this step is usually called the "netlist".

3. **Synthesis**: Taking the high-level description of the system from step 2 and "compiling" it into gate-level logic. This step includes optimisation of the logic. The programmer may set options/hints for the synthesizer, but does not fully control the process – it is similar to how a C compiler produces optimised assembly not meant for human eyes.

4. **Implementation**: placing and routing the gate-level logic onto the FPGA. In this step, physical placement of the various parts is decided, and routes for signals are created. Thus, this is where the final design, and the exact resource consumption, become apparent. The programmer may provide hints or express preferences for various trade-offs, but this step is largely automated. Because this is a difficult optimisation problem, this step usually takes a rather long time (for our project, it is about 3-4 hours). When we have the final design from this step, signal propagation issues, especially timing, can (and should) be checked. Once one has the full design, it is a good idea to run a full low-level simulation, which simulates the execution on the level of signals and circuits.

5. **Generating the bitfile and flashing it onto the FPGA**: Creating a binary file in a format specific for the FPGA, and loading it onto the FPGA. This file configures the FPGA's cells to the desired circuitry.

2.5.3 The hard problem in FPGA design: Meeting timing

In FPGA design, writing the logic of a program is only the first step. As mentioned in the previous section, it is only after the program has been turned into a circuit (the "implementation" step) that one can be sure that it will work. Often,

especially when the program is complicated, one will instead find that the circuit was created, but exhibits timing violations.

When signal propagates through a circuit, the various elements such as gates and also routes (i.e. "wires") will delay it by a tiny bit: the speed of light is finite, and the gates need a bit of time to stabilise on the correct value. These signal propagation delays accumulate; and if the signal takes too long to arrive on the output, the circuit will not work correctly. This problem is referred to as a timing violation (because the signal delay violates a timing constraint required by other parts of the system), and it is often the most difficult problem when designing an FPGA program.

The usual reason for a timing violation is logic which is too complex to execute within one clock cycle. For example, an implementation of a round of AES is a somewhat complex computation that requires quite a few gates, and thus it may add a significant signal propagation delay: for our implementation, this is about 0.4 ns. Thus, if we attempted to run 10 rounds of AES in one cycle, the total delay would be 4 ns. If we attempted to run such a circuit at a 300 MHz clock, we would only have 3.3 ns per cycle (frequency 300 MHz = period 3.3 ns) and thus the signal would arrive too late and the circuit would not work. However, running the circuit at 200 MHz, where one clock cycle is 5 ns, would work (and we would have 1 ns left if we needed to fit in some extra operations). Thus, when fixing a timing violation, we have two options: either we reduce the number of operations per cycle, or we lower the clock frequency. Reducing the number of operations per cycle (while not changing the functionality) means that our program will need more cycles and thus will have increased latency (but if we use pipelining, we can avoid losing throughput). Lowering the clock frequency lowers not only the latency, but also the throughput (as fewer bits per second can be processed). Conversely, if our design achieves low delays, we may be able to increase the clock frequency and thus also throughput.

Timing is a complicated problem and there are no silver bullets. Lowering the delays in one part of the design is often a trade-off, as it may cost more area or even exchange lower delays in one place for higher delays in another place. However, what makes or breaks a design is the worst path (as that is what limits the maximum clock frequency), not the sum, and thus such trade-offs are a useful tool.

3 Design overview

The border router's algorithm, as outlined in section 2.1.4, is as follows (more details have been added here):

1. Identify the packet as SCION and parse the SCION common header.
   If the packet is not SCION, send it to software.
   If the parsing fails, drop this and generate an SCMP[1] error packet.
2. Extract and parse the current HF according to the offset information in the offset header.
   If the extraction or parsing fails, drop this and generate an SCMP error packet.
3. Verify the ingress interface and the incoming IP overlay.
   If incorrect, drop this and generate an SCMP error packet.
4. Check that the HF is not expired.
   If incorrect, drop this and generate an SCMP error packet.
5. Validate the HF MAC.
   If incorrect, drop this and generate an SCMP error packet.
6. If the packet destination is in this AS, forward it according to the intra-AS routing.
7. Update the HF offset information.
8. Update the IP overlay header according to an egress interface ⇒ IP overlay data table, including recomputing IP and UDP checksums.
9. Select the output port corresponding to the egress interface and send the packet out.

Note that the fast path is the interesting part of this project, while the error generation (which needs rate-limiting and may change significantly in the future) and local delivery (which needs additional configuration and state on the router

[1] SCION Control Message Protocol: the equivalent of ICMP in SCION. These packets should be rate-limited, and in future versions of SCION, they may become source-authenticated (i.e. require additional cryptographic operations).

for intra-AS routing) are extra work not essential to prove the viability of SCION in hardware. Therefore, a key design decision is to structure this project as a "really smart NIC": all packets which can be completely processed in the FPGA will be processed in the FPGA, and any packets which cannot be completely processed (for any reason) will be sent to the OS unmodified, through what looks like a normal network interface to the OS. Our FPGA will, further, pass through any packets arriving through the OS-visible "fake" interface. Thus, our implementation does not need to know anything about the unprocessable packets and any replies. This, together with SCION's mostly stateless design, enables us to run the software SCION router unmodified and completely unaware of the fact that the FPGA taking care of most of the traffic.[2] Thus the code running on the FPGA can be kept relatively simple, as it only needs to handle things that must happen at line rate: any SCION control plane messages, error handling, local delivery, or any of the intra-AS L3 packets such as ARP will transparently "just work". Additionally, with this approach, we can start with the very simple "pass everything through to the OS" program, and incrementally move more and more functionality into the FPGA. This greatly saves development effort and thus enables us to complete this project within the time scale of a Master thesis.

With this approach, the control plane for our project consists of two separate programs: the unmodified SW router, to which we sent any unhandled packets but otherwise do not interact with, and a thin wrapper that reads the SCION configuration files and writes the configuration into the NetFPGA, as well as provides a few supporting functions. Details about the control plane can be found in section 4.1.4.

Figure 3.1 shows the components of our design.

All the highlighted modules in the figure are loosely coupled in order to ease maintaining, extending, and building on top of this project. Thanks to the idea of transparently passing unprocessed packets to the OS, the processing pipeline can stay relatively simple, and the control plane software can be just a thin wrapper running alongside the unmodified SCION router. Furthermore, if the control plane gets extended or changed, it is sufficient to upgrade the software router and none of this project's code needs to be changed, as long as there are no changes in the forwarding logic (which isn't likely to change as often).

[2] Note that this would not be quite true in production, due to issues like runtime configuration changes and traffic shaping functionality. However, for a proof of concept, this approach works very well.

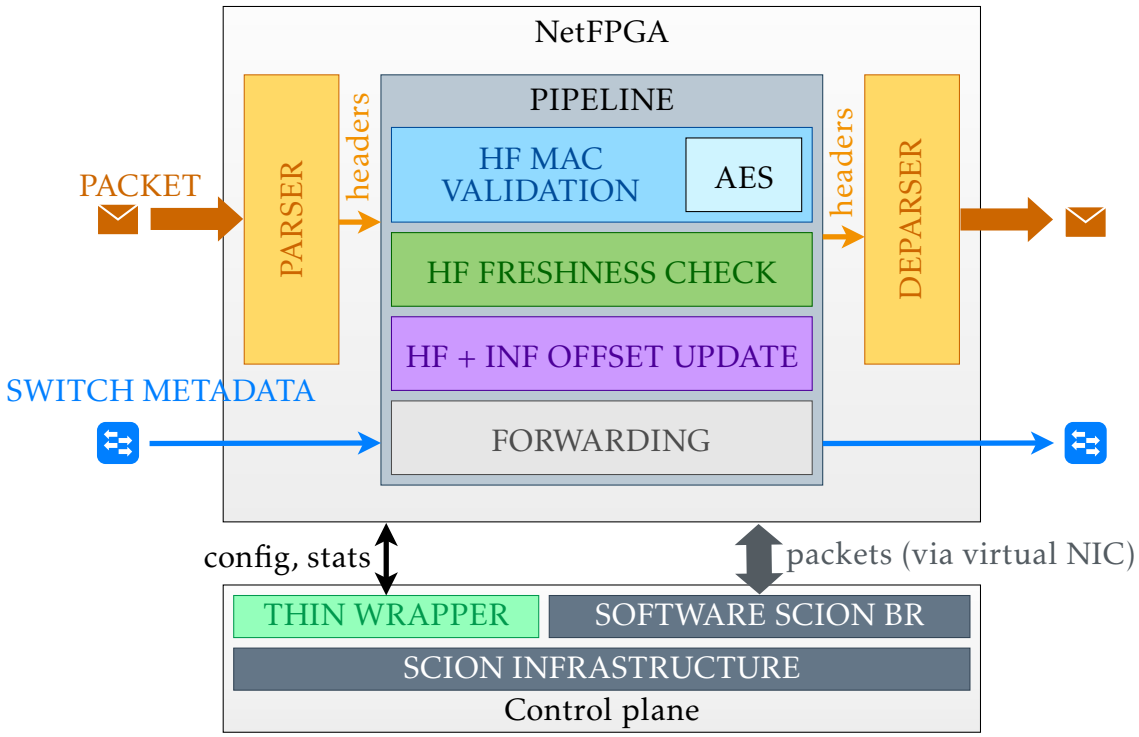

Figure 3.1: TODO somebody should write something here.

4 Implementation Challenges

The problems we encountered during the implementation were too numerous to list completely. Therefore, in this section, we present a select few particularly interesting, challenging, or instructive issues, and our solutions for them.

TODO dear reviewers, is the order of these sections good?

4.1 Software engineering aspects

From the software engineering perspective, this project is somewhat on the larger side given the modest time and personnel assignment of a single Master thesis. Therefore, in order to speed up development, we needed to do a few interesting things. This section introduces those.

4.1.1 Project building workflow

The standard workflow for building a project for FPGAs is very time-consuming, complicated, inconvenient, impossible to perform without training or lengthy documentation, and requires manual interaction in multiple steps. In the case of this project, it takes over 6 hours to go from source code to a binary that can be flashed onto the hardware. (See section 2.5.2 for background information on why this is so.) Originally, these 6 hours were interleaved with frequent manual checks that required opening the FPGA development environment GUI and running multiple long commands with input manually copied from previous steps. We do not consider that to be a sensible workflow, and if we had kept it that way, we would not be able to make progress fast enough to get anywhere with a project as complex as this. Therefore, one of the first things we worked on, before delving into the project-specific issues, was to improve the workflow.

Due to this, as well as due to supporting multiple targets as described in 4.1.3, a lot of thought went into organising the project's source code repository, and into scripting the build process. This required vastly more effort than anticipated, because the numerous NetFPGA scripts make unnecessary and often unsatisfiable assumptions about the project's structure. However, it was worth it: at the moment, the project can be built with `make build`, which requires no manual interaction (nor lengthy documentation). This frees the programmer from frequently checking in on the build, allowing them to spend those 6 hours more productively. Further, running `make` in the project directory lists (automatically

generated, and therefore always up-to-date) tasks and build settings that the programmer may need, thereby reducing the need for out-of-band and thus inevitably out-of-date workflow documentation.

4.1.2 Modularity and incremental development

Though one may be tempted to just "make things work" as fast as possible, larger projects quickly start to have problems if one does not spend extra effort on thinking about the design of the project. Keeping the code base well thought out may seem like an unnecessary luxury at the beginning, when the project is small and the programmer is excited to see things working. And it can in fact be a good idea to leverage that excitement: in our case, we often first wrote very messy code, as we were figuring out how things work and what approach would be best. Before having a good overview of the possible approaches for different aspects of the project, any design created would not survive contact with reality. However, left unchecked, development of new features without stepping back and re-considering the design will lead to code that is not only difficult to read, but also difficult to modify or extend. Thus, in this project we periodically interleaved development with refactoring and code cleanup. As it goes with academic projects, due to lack of resources, the project's source code could be further improved. However, even the limited efforts to keep the mess under control made a large difference, and ultimately they were the deciding factor that enabled us to go beyond a completely bare-bones implementation.

Modular code base

In order to make it easy to find things, enable or disable different functionality, and ultimately also make our code usable as a library for processing SCION packets in different projects, we separated different features into independent, loosely coupled modules. Thus, the different parts of packet parsing, MAC verification, timestamp validation, routing, overlay handling, and monitoring are separate, and can be mixed and matched as needed and debugged separately. Furthermore, modularity is in many cases a requirement for achieving portability/target independence: for example, the MAC verification routine needs an AES encryption routine, which will be implemented and accessed differently on different platforms. Also, the the HF format can be (within some constraints) chosen by the AS. Thus, the sub-parsers within the packet parser, as well as the HF MAC verification routine, also need to be easily customisable. Figure 4.1 shows the different components of the systems, and figure 4.2 shows the sub-modules of the parser.

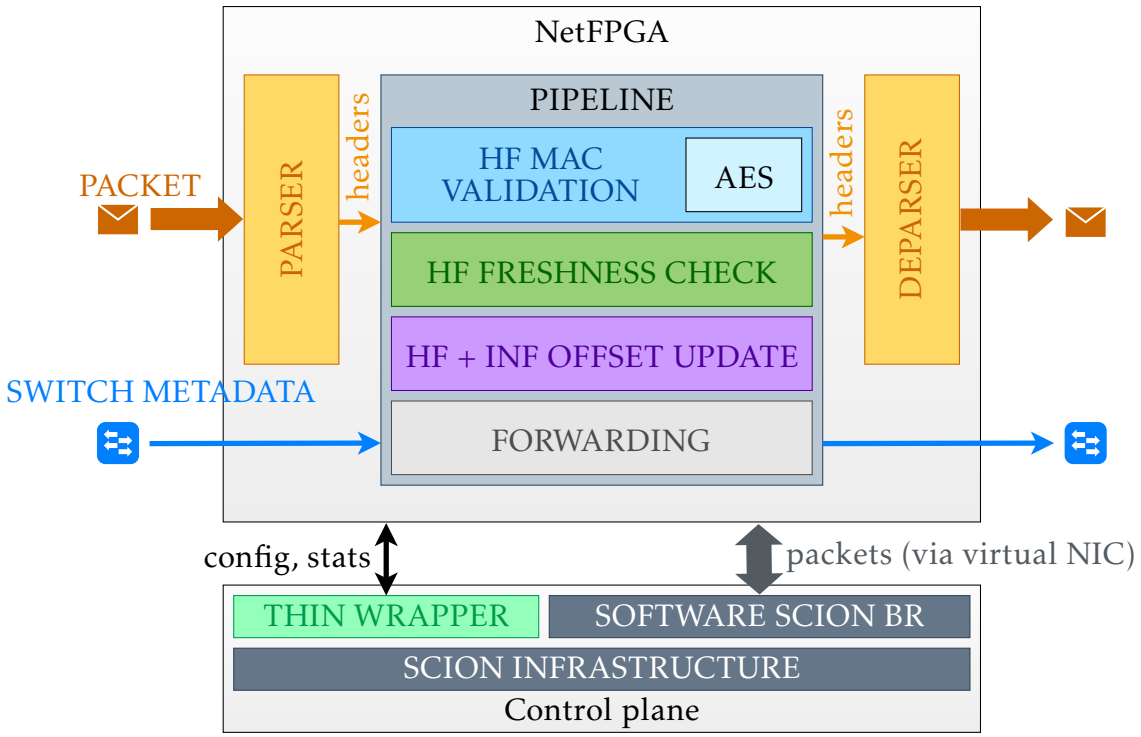

Figure 4.1: The major components of the system. (Same as figure 3.1.)

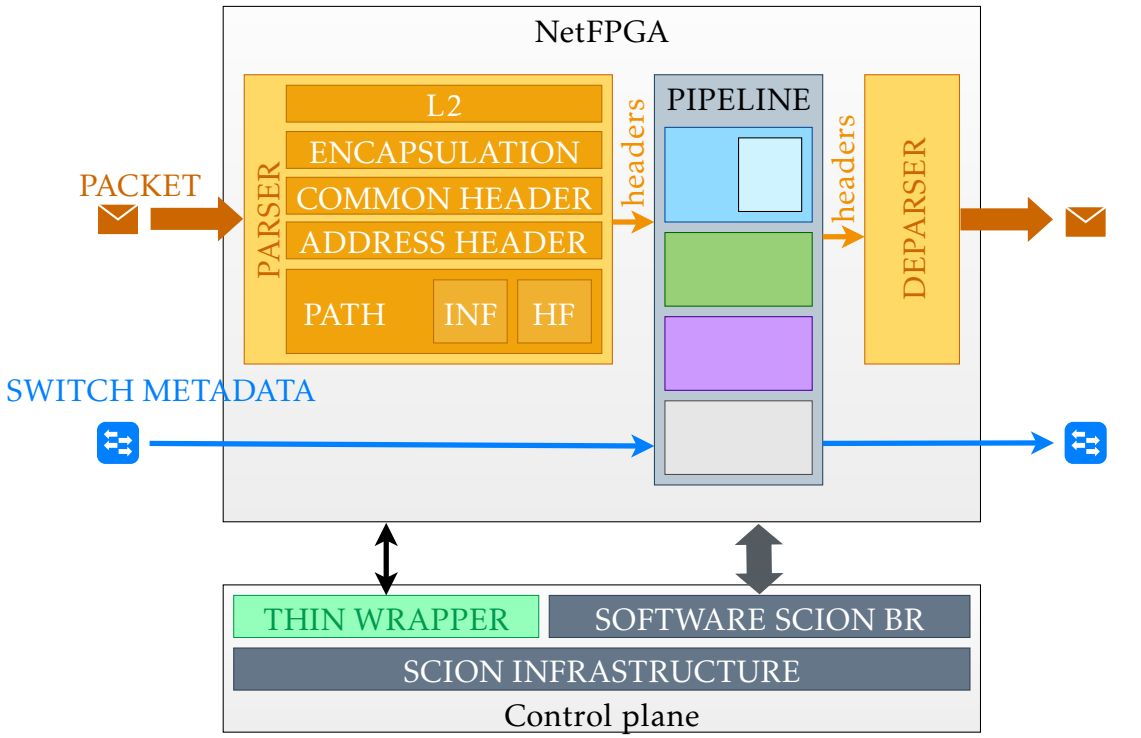

Figure 4.2: The sub-parsers of the parser component.

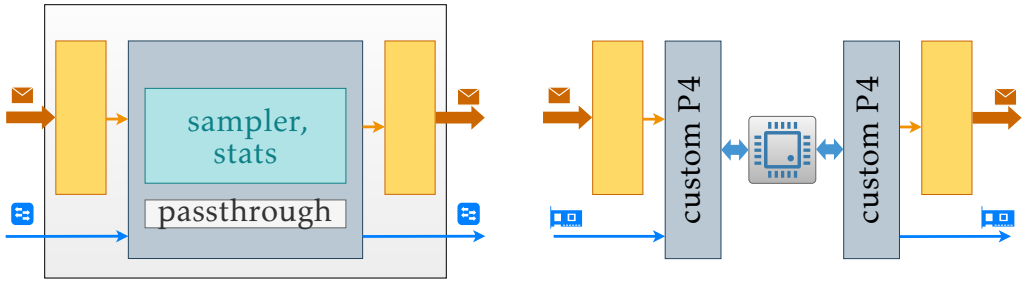

Figure 4.3: Example use cases for our SCION library: NetFPGA-based network monitoring system (left) and a SmartNIC-enabled end host used for high-throughput data processing (right).

SCION as a library

Thanks to the modularity described in the previous section, our project could be easily customised for other use cases. As an example, we could create a SCION-aware in-band network monitoring system by keeping our SCION parser, HF validation (if needed), and deparser, but instead of routing packets (and updating HF and INF offsets in the header), we would collect samples/statistics, and pass the packets through unmodified. This would be easy to achieve, as the different parts of our project do not depend on one another, and thus can be easily swapped out as needed.

Another example, where the target independence from section 4.1.3 also comes into play, would be a special-purpose host that processes large amounts of data that needs to be sent out over a SCION network. Such an end host with a P4-capable SmartNIC could re-use our parser and deparser, add any custom processing needed, and use DMA to transfer data to and from the CPU. Thus, it could achieve extremely high goodput, as not even the SCION headers would need to be created on the host and thus the full DMA bandwidth could be used for application data.

Figure 4.3 illustrates these example use cases.

Enabling incremental development by sending unhandled traffic to the CPU

As mentioned in the design overview (section 3). we chose to present the NetFPGA as a NIC to the host system, run an unmodified software router listening on the NetFPGA's "fake" DMA-based interfaces, and thus be able to transparently send traffic we are unable to process to the CPU. We also directly send out any traffic received on the soft interfaces, thus transparently enabling also IP traffic and ARP (which is relevant for the SCION IP overlay). Figure 4.4 illustrates this design.

Besides simplifying the handling of control packets and errors, this approach also facilitates testing features with real traffic while the system is not yet complete. Thanks to it, we are able to test with real traffic despite not implementing the UP flag (reversed path segments) or any form of error packet generation. Thanks to it

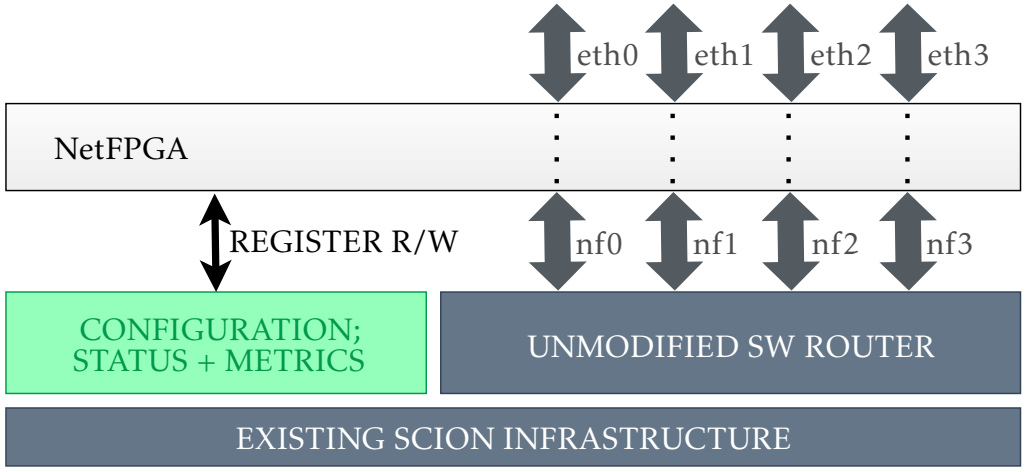

Figure 4.4: We map the DMA-based interfaces visible to the host to the physical ports on the NetFPGA.

we were also able to make sure that MAC checking works on real packets while other features (such as handling the IP overlay) were not ready yet.

In order to make this approach work with as little effort as possible, we needed to move communication with our control plane (such as configuration and monitoring) out of band – so that the only packets traversing through the DMA-based interfaces are coming from or intended for the outside world. Thus, we only use registers (memory) accessible from the host through DMA, and no control packets. This simplifies the data plane design, as its responsibilities are reduced: it only needs to assume that the registers contain the right values (or, in the case of monitoring, keep the metrics registers up to date), and not worry about updates. It requires the control plane to run on the physical host rather than remotely, but if needed, it would be possible to only run a thin wrapper on the host and make it "translate" control packets into DMA writes/reads. Thus, this approach provides multiple advantages, and has been very helpful during the development.

4.1.3 Target independence/portability

Writing portable bare-metal software is a non-trivial task with no general solutions, as every project faces different challenges in this area depending on the project's aims, and the number and diversity of the targeted platforms. In our case, P4 is in theory a target-independent language, and the same code can be compiled for many different targets, from software targets such as a user-space switch or a DPDK application, to hardware targets such as FPGAs and programmable switches. However, in practice, every target has different limitations, supports a different subset of the P4 language, and requires different extra resources (i.e. files needed to build the program for this target).

For this project, we chose to support the open-source reference P4 software switch and the NetFPGA SUME board, and keep the door open for easily adding other targets in the future. In the ideal case, we would be able to work within the intersection of the P4 features supported by all the intended targets, and write unified code for the vast majority of the functionality. However, unfortunately (though not surprisingly), in the case of the NetFPGA's proof-of-concept toolchain, the limitations and unsupported P4 features turned out to be significant (see sections 4.2 and 4.3). Therefore, confining ourselves to the subset of P4 supported by both the reference compiler and the NetFPGA compiler would not allow us to implement SCION. As such, a single, unified code base could not be achieved and we frequently needed to write different code for each target.

In order to avoid a combinatorial explosion of different implementations, we chose to not depend on targets in our code, but on features. In other words, our code's `#ifdef#` statements (which conditionally enable or disable code) were not of the form `#ifdef TARGET_NETFPGA`, but e.g. `#ifdef TARGET_SUPPORTS_VAR_LEN_PARSING`. This means that we will not necessarily need to modify existing code when we add a new target, we will just need to define the features it supports. We provide an easy way to check for features required by the project's code, so that adding a new platform requires as little effort as possible: running `make find-all-target-supports` prints all `TARGET_SUPPORTS_*` features that the code uses.

In order to accommodate for diverse build processes and supporting files, while avoiding duplication where possible, we have created a two-level structure in our repository that captures the various types of differences among targets that we anticipate: the supporting files, as well as Makefiles that define the build process, live under either `platforms/<platform>` if architecture-independent, or `platforms/<platform>/<architecture>` if architecture-specific.

The differences between the software switch and the NetFPGA can be handled cleanly with our approach, which proves that it does indeed work in reality.

4.1.4 Control plane

While the focus of this project is on the data plane, we needed to create a minimal control plane, so that we could test the full functionality required of the data plane. The design of the control plane is also interesting from the modularity viewpoint. Therefore, this section presents a brief overview of the design our control plane.

As explained in section 3 and shown on figure 4.4, we are using the unmodified software SCION router to handle packets we cannot process in the data plane, as well as any errors. Therefore, this responsibility is taken care of, and our control plane only needs to provide the following functions:

- **Loading the BR configuration into the NetFPGA**: the AS key for HF verification, the SCION interface ID to physical port mapping, and the IP overlay data for every interface. Also setting the Ethernet address (MAC address) for

its interfaces according to the DMA-based host interfaces, so that ARP/NDP can be handled by the host transparently.

- **Providing a clock**: as mentioned in section 4.4.2, we need to frequently update a register in order to tell the NetFPGA the current time, so that it can validate timestamps in the path's info fields.

- **Exporting metrics**: frequently reading registers where the NetFPGA keeps various counters, and making the data available from the outside. We chose to export the data in a format suitable for the Prometheus monitoring system[1], and thus run an HTTP server that, upon request, reads the registers and returns the data.

These features are completely independent, but need a level of coordination: the clock needs to be updated periodically, while the metrics need to be read on demand (whenever Prometheus requests the metrics page). Thus, a program combining all of these features would either introduce unnecessary coupling between the independent modules (in case of a simple single-threaded program) or introduce unnecessary coupling and add some concurrency bugs (in case of a multi-threaded one).

Thus, in order to keep our modules independent (unlike in a traditional single-threaded program) and easily composable (unlike in the case of multi-threaded code that would need locks to protect access to the NetFPGA registers), we chose to use an event-driven single-threaded approach using the Twisted Python framework[2]. At the core of Twisted is its event loop, implemented as part of the library and outside of the user's control. The user programs the event handlers needed, in our case e.g. a handler for "half a second has passed, we need to update the clock register" and "a request came to the HTTP server, we need to fetch and return metrics". Twisted uses cooperative multitasking, so only one handler runs at every time, and it hands off control back to Twisted's event loop when done. The event loop keeps track of events, so this way, the handlers can be completely unaware of each other, and no race conditions or concurrent writes can occur.

Because this approach allows us to make the modules trivially composable, we were able to keep the various handlers completely independent of each other. This enabled us to quickly add features without worrying about breaking existing ones. Even more interestingly, we have been able to make the modules complementary to the P4 modules, and therefore have kept clean module boundaries across the whole stack. Thus, when changes need to be made, or only a part of this project needs to be used (as in the library paradigm), it is possible to also reuse the needed parts of the control plane.

[1] https://prometheus.io
[2] https://twistedmatrix.com

4.2 Working around compiler and toolchain bugs

Some hurdles are expected when working with cutting-edge technology. However, the state of the NetFPGA's P4 toolchain was enough of an issue to warrant a separate section in this thesis. Therefore, in this section we unfortunately present a subset of the problems encountered.

Note that the parser-related problems (and their solutions, and the problems in the solutions), which constituted a large fraction of the problems, were interesting enough that we discuss them separately in section 4.3. We do not re-iterate those issues here, so see also that section for a complete picture.

TODO reorder the sections

4.2.1 Very high tooling complexity

The tooling around the P4-NetFPGA project is a jumble of scripts with no rhyme or reason. Besides the extremely complicated build process alluded to in 4.1.1, there were several other issues. We summarise them here.

No way to cleanly add a custom module

NetFPGA scripts expect information about all external non-P4 modules to live in one file distributed with the NetFPGA repository. This assumption is also present in several other NetFPGA scripts. Thus, adding a project-specific external module, such as our AES implementation, turned out to be a difficult problem. We spent significant effort trying to do it cleanly, but we found no good options. Raising the issue with the NetFPGA developers got vague agreement, but our proposal for a concrete patch did not receive a reply. Thus, building this project still requires modifying a Python file that ships with the P4-NetFPGA distribution.

No way to do low-level debugging

There is currently no way to inspect the internal state of the hardware. In theory, it should be mostly unnecessary to debug the hardware, as the whole-system circuit-level simulation, which enables one to look at the signals in the circuit, should reveal the vast majority of problems. In practice, though, due to the very high complexity of the scripts, we were not able to run the whole-system simulation for our design. Luckily, the issues we ran into were either visible in the high-level behavioural simulation, or we were able to solve them by means of a few days of thinking rather than by inspecting the system. However, we strongly believe that simplifying the usage of the behavioural simulation would be a worthwhile goal.

4.2.2 Packet generation helper script utterly broken

The NetFPGA example repository comes with a sample script for generating packets and metadata and testing the code in the behavioural simulation. This script is completely broken: it does not even produce the right number of bytes. This error has apparently gone unnoticed because other parts of the toolchain pad the bytes in questions with zeroes if there are too few (instead of raising an error in order to make the problem visible), and so one does not notice the problem if testing only the cases where the missing bytes are all zero. Due to the extremely low code quality, attempting to debug and fix the script was very quickly abandoned. There is no documentation for the metadata format, though, and thus we needed to partially reverse engineer the script it in order to find out what exactly it was attempting to do (even if incorrectly). Reading the code in question has left the author with permanent trauma. It would be better to not include buggy example scripts at all, especially when they serve as an excuse for not adding any documentation.

4.2.3 Changes to the code have highly unpredictable effects

Whenever one writes new code for the NetFPGA, the likelihood of encountering a more or less obvious compiler bug is nearly 1. Beyond that, there are no certainties.

In the good cases, the compiler exits with a "Compiler Bug" message, which can be more or less related to the actual problem (often a notable amount of guesswork and experience with previously encountered compiler bugs is needed to decipher it). An example of this would be the case of attempting to use an "if" statement in the wrong place, which results in the message "Compiler Bug: $\text{action}_{\text{split}}$ unimplemented".

In the bad cases, the program compiles successfully, and then does something surprising in the simulation. This may or may not be caught immediately, as test cases for this particular flavour of surprising may not be present. In that case, the problem may become apparent only several commits later, when another feature depending on it is added. This makes it very difficult to track down problems, and causes constant anxiety for the programmer. As an example, an issue with packet corruption (described in 4.3.2) was not visible in the simulation at all, and, together with the long compilation times, it took about a week to track down the cause.

4.2.4 Silent failures, no way to detect errors early

Almost every part of the very complex toolchain contains something that fails silently: from device initialisation scripts through scripts that fill tables to the many stages of compilation and simulation. A prime example of this is the HDL compilation step that must be performed at the beginning of the simulation or

synthesis: many of the NetFPGA-provided HDL files generate compilation errors, but they are not fatal and somehow, the project works anyway. Thus, in order to let the project build, the NetFPGA build scripts simply silence all failures from the HDL compiler. The consequence of this is that if the HDL code generated from P4 or an external module contains an error, this will not be obvious. The compilation process will continue, either failing 15 minutes later (thus costing the programmer extra time), or in some cases, it will not fail at all, use stale intermediate data instead of the erroneous parts, and do something unexpected, thus causing utter confusion and more wasted time. In order to preserve our own sanity, we ended up replacing all scripts that weren't closely intertwined with other ones. Unfortunately, there were not many of those, and thus we had to train ourselves to distinguish fatal errors from non-fatal ones in several thousand lines of compilation logs.

4.2.5 Table matching broken under unclear circumstances

We use a table to match the SCION interface ID with the physical port and the IP overlay settings. We had tested table matching previously in a toy project when getting to know the toolchain, so we did not expect any problems with it. However, during the development, table matching suddenly stopped working. We triple-checked that the keys in the table and in the packet data were correct, but that was not the problem. As a sanity check, we went back to the known working toy project, and made sure that tables still worked there, which they did. Having no indication of what was wrong, we proceeded to copy over parts of our code between the two projects in order to narrow down the possibilities. It was largely by accident that we discovered that if we include an extra dummy table at the beginning of the program, the actually relevant table would magically start working again. In the end we discovered that only the "first" table match was broken: if we included two tables, the one that was accessed first in the control flow of the program would fail to match, and the second one would work.

We reported this problem to the NetFPGA developers, and worked with them to find out what was going on and why this was only happening in our project.

Finally, we found out what was going on: it turns out that calling an extern function (i.e. escaping from P4 into a Verilog/VHDL module) corrupts some state in the P4 module and breaks the following table. Our guess is that the use of the table then restores the corrupted state, and after that the execution proceeds as expected. This behaviour is rather concerning, as the P4 module should in theory be completely independent from other modules, and not even the NetFPGA developers know why this is happening.

In our case we need to call extern functions in order to validate the HF: as described in section 4.4, we need an extern function to get the current time (needed to validate the timestamp) and to compute AES-128 (in order to check the HF MAC). As we want to send the unmodified packet to the SW if any errors are encountered, these checks should happen as soon as possible. Specifically, we

would like to do them before the use of the table, as using the table modifies the packet (in P4, the "match-action", i.e. looking up a key in a table and performing an associated action based on the match, is an atomic operation). Therefore, our first idea was to add the dummy table, so that we could keep the control flow of "first call externs, then fix the corruption using the dummy table, then access the actually relevant table".

However, the extra table costs about 0.1 ns of time, which is a non-negligible amount (see section 4.5 for a detailed discussion). Thus, when optimising timing, we switched to a different approach: because we are using the `packet_mod` deparser described in 4.3, we are able to choose very late in the program (specifically, in the deparser) to either commit to the packet modifications or keep the old data. As such, we are able to use the following approach: First, we access the table (which may modify the in-program representation of the packet, but does not affect the result of the checks). Then we call the externs and depending on the results of the checks, we may or may not set a "can modify" flag. We then check this flag in the deparser before committing to the in-program packet modifications in the real packet on the wire. This way we are able to use tables together with externs without using the somewhat expensive dummy table workaround.

4.2.6 High difficulty of writing cross-platform code

The NetFPGA only supports a particular subset of P4, and any documentation on P4 support is incomplete. During the work on this project, we encountered several unsupported features not mentioned anywhere, which we added to the NetFPGA documentation[3]. Knowing about the unsupported features is only the start, though: the next step is writing code without them. Dancing around the limitations is a non-trivial exercise, and adding workarounds has cost us a lot of effort. Besides the parsing-related ones, in particular error handling and use of externs needed extra code and much trial and error.

4.2.7 Control plane API issues

The NetFPGA repository contains a C API for accessing the NetFPGA's tables and registers via DMA, and a Python wrapper for the C API. Both the C API and the Python wrapper have unexpected limitations.

In P4, the programmer can define registers of any size. However, the API only supports 32-bit registers, and thus only those can be used for anything that involves the control plane. We attempted to extend the API, but this limitation is in fact deep in the NetFPGA's hardware design, and thus would be very time-consuming to fix. Therefore, we had to change our program to use 32-bit registers only. For example, in the case of the AS key used for HF MAC verification, which

[3] https://github.com/NetFPGA/P4-NetFPGA-public/wiki/Workflow-Overview#limitations

is 128 bits long, we had to use 4 32-bit registers, and the key is written and read in parts. While this can be done for our proof of concept implementation, in the real world a non-atomic write of the key would cause problems and thus additional workarounds would be needed.

The table API also had issues. In P4, a table is a key-value mapping where the value is an (*action*, *data*) pair. The data format is given by the action signature. As an example, a table with 2 actions, `action1(param1)` and `action2(param2, param3)` should contain values of the "shape" `action1 | param1` and `action2 | param2 | param3` (where | stands for concatenation). This is given by the P4 standard and expected by every P4 programmer. However, that is not how the NetFPGA API works: attempting to use the API in this way produces an obscure error message. Instead, the API expects the action data to be a union of all possible action data, with zeroes everywhere but in the relevant fields. For the example above, this means that we need to use `action1 | param1 | 0 | 0` for action 1 and `action2 | 0 | param2 | param3` for action 2. This deviation from the standard is not documented anywhere and, frankly, makes no sense. The order of parameters is also not easy to discover, and even if they internally choose to represent the tables this way (which seems like a waste of precious area), this implementation detail should not leak out and cause a deviation from the expected (and far more reasonable) API. Thus, we wrote our own Python wrapper for the C API, which handles this internally and exposes the standard P4 API.

The Python wrappers for the C API were not at all usable for our project. They hard-code assumptions about the location of metadata about the P4 program (which cannot be satisfied for any sensibly-structured project), and, more importantly, they print an error message to the terminal instead of indicating the error to the caller. Thus, errors from the Python API cannot be handled in any sensible way, and it was easier to completely rewrite it. Further, while the C API only accepts input in "hex string" format (an unusual and error-prone choice for C, it would be much simpler and more natural to take raw bytes), the Python wrapper for the table API attempts to convert common string notations for bytes, such as `11:22:33:44:55:66` (MAC address format) and `1.2.3.4` (IP address format). However, it does so incorrectly – it somehow works for 6-byte colon-separated hex digits and 4-byte dot-separated decimal digits, but returns surprising values for other sizes. Thus, we used the opportunity when writing our own wrappers and also added proper parsing.

Our rewrite implements all of the original Python API's functionality for registers, and enough of the table functionality to be usable by us. It includes dealing with the clumsy non-standard table API described above, proper error handling, correct input parsing, and fewer limiting assumptions. Notably, despite the added functionality, in terms of the number of lines of code, our rewrite is about one third of the original wrappers.

4.3 Implementing the parser

4.3.1 Handling the variable-length SCION path

TODO remove "P4-SDNet" because that was never introduced, just call it "the compiler" or something Unfortunately, the SDNet P4 compiler implements only a subset of P4, which has created unexpected challenges when implementing the SCION packet parser. The biggest issue was that at the time of writing, with P4-SDNet the parser cannot work with any variable-length data: this includes not only `varbit<n>` types, but also header unions, and – most importantly – header stacks. Furthermore, it is unable to even skip over variable-length parts of the packet. In summary, all packet offsets must be compile-time constants. This poses a problem for SCION: the path in the packet is variable length (it can contain any number of hops).[4] Therefore, the parser cannot easily deal with the path in the SCION packet.[5]

To get around this problem, we employed the following steps:

First of all, we opted to design the parser so that it parses only the actually needed data: while the path can be arbitrarily long, any single BR only needs to process $O(1)$ hop fields (usually one, or two in case of a shortcut path). This not only makes it possible to compile for the NetFPGA, but also improves performance on other targets.

Next, we needed to solve the problem of emitting headers we have skipped: using only the standard features of P4, it is impossible to deparse parts of the header which have not been parsed. (The payload is copied without being parsed, but the payload is defined as anything **after** the last thing we parsed – so if we skip parsing something in the header, it is lost and we are unable to emit it on the output interface.)

The solution on the NetFPGA is to use the non-standard `packet_mod` feature of P4-SDNet, as this (unlike the standard deparsers) allows to modify the existing header instead of creating it anew, thereby allowing us to not lose the skipped parts of the header. The `packet_mod` feature mirrors the structure of the parser in the deparser: it allows us to define a state machine that produces the output header by skipping over or updating the input header as needed.

[4] The SCION host addresses are also variable-length, as the address type tag in the common header defines what kind of address it is. For this case, we opted to use the C preprocessor to conditionally replace the union with a struct with three fixed-length fields (one for each possible type of address), only one of which is parsed and made valid for a given packet. (This causes a small increase in FPGA area usage, but any other solution would be significantly more complex, so this is a good trade-off.)

[5] In fact, it would be possible to parse the whole SCION packet by using the C preprocessor to "unroll" the parser at compile time, and parse the path segments into `struct`'s with fields such as `hop1`, `hop2`, ... (size fixed at compile time). However, this would make actually using the fields very difficult, as I would need more preprocessor magic to index into such a struct; and additionally it would drastically increase my FPGA area usage.

Using the `packet_mod` feature is not straightforward, either: we needed to switch to the `XilinxStreamSwitch` architecture, as this feature is not available in the `SimpleSumeSwitch` architecture that is the default on the NetFPGA. This required modifying the Verilog wrappers provided by the NetFPGA developers, which was not trivial: the P4-SDNet compiler is closed-source and the few lines of documentation in the wrappers were actively misleading. Therefore, much trial and error was required to understand what was going on and how to change it.

Obviously, using a non-standard SDNet-only feature means that with this approach, the program would not compile on a standard P4 compiler. However, standard P4 compilers (unlike the P4-SDNet compiler) tend to implement parsing variable-length headers. Therefore, we are able to emulate `packet_mod` for standard compilers by adding an extra struct to keep track of the "skipped" parts of the headers, and instead of skipping them, we parse them into the (variable-length) fields of this extra struct.

We can use the C preprocessor to hide this difference, thereby keeping our code portable while being able to use `packet_mod` where available. This not only allows us to parse the variable-length SCION packet on the currently incomplete P4-SDNet compiler, but also allows us to harness the performance benefits of the `packet_mod` feature on any target where it is available.

To make this approach work, the last step is to turn all packet offsets, including the ones used only for skipping, into compile-time constants, so that P4-SDNet is willing to compile the code.

Fortunately, SCION hop fields are always multiples of 8 bytes. Therefore, the obvious approach is to add a parser loop that skips over 8-byte chunks until it gets to the needed place. Unfortunately, when attempting to compile this, the P4-SDNet compiler exhibited a rather curious behaviour: due to what we assume to be an off by one error in the compiler, the loop compiled into invalid SDNet code and was not accepted by the later stages of the build process. We attempted to isolate the problem to get a better understanding of it, but the compiler bug manifested only in very specific conditions, and thus we were unable to find a workaround for it. Therefore, the loop approach did not work.

The final approach (which did work, though at a cost) was as follows:

If we assume a maximum path length K, there are only $O(K)$ many options for how many different sizes we might need to skip. Therefore, we can use the C preprocessor to "unroll" the parser and create separate states for skipping $1, 2, ..., K-1$ hop fields. This causes an $O(K)$ increase in FPGA area usage, but no latency increase – our logic becomes wider, but not deeper.

Unfortunately, the FPGA area usage increase comes with an additional problem: the amount of RAM needed to build the design increases with area, and for this case, we were unable to build the parser for paths larger than 16 with 32GB of RAM. While "get more RAM" would be a viable approach, it is not very elegant, and the FPGA area usage is also a cost that is worth bringing down. Therefore, we came up with a two stage skip idea to reduce these: if we want to support a max path length K blocks (let's say $K = 64$), we create two skips: the first one will skip

over "big" blocks of size $\sqrt{K}$ at a time, and the second one will skip over normal-sized blocks. This way, we can skip over any number of blocks $N, 0 \leq N < K$ by first skipping over $\sqrt{K} * \left\lfloor \frac{N}{\sqrt{K}} \right\rfloor$ "big" blocks and then skipping over $N \mod \sqrt{K}$ "normal-sized" blocks. Instead of requiring $O(K)$ FPGA area (and therefore RAM), this requires only $O(\sqrt{K})$ and thus enables much longer paths at reasonable costs.

4.3.2 Unexplained packet corruption when handling parser errors

As explained in section 4.1.2, our parser is composed out of several sub-parsers. Thus, we need a way for errors from the sub-parsers to propagate up through the parent parsers. The traditional way to achieve such error propagation is by using exceptions, but exceptions are problematic in many contexts and are not supported by P4. However, in the case of P4 parsers, a behaviour equivalent to exceptions is available via the `reject` keyword: P4 parsers are state machines, and transitioning to the special state `reject` causes the parser to immediately finish parsing, including any parent parsers. Therefore, this is the natural way to handle parser errors and we attempted to use this in our project.

According to the NetFPGA developers, this is supported and should not cause any problems. However, in our project, which consists of a somewhat deep hierarchy of sub-parsers, that was not what we observed: upon encountering the `reject` keyword, the pipeline would proceed as expected, but on the output, the outgoing packet would get corrupted and contain zeroes instead of the original data. We made sure that it was due to the `reject` statement: when we changed our code to exit from sub-parsers without `reject` (i.e. via the `accept` special state, which exits the sub-parser but not the parent parsers) and added manual error propagation, this behaviour was no longer observed.

Due to unavailability of the compiler source code and time constraints, we did not investigate this bug further. However, two options come to mind: it is either a problem that manifests only in complex parser hierarchies (which, given the previous section, would not be surprising at all), or it is a bug only present in the experimental `XilinxStreamSwitch` architecture (the use of which we explain in the previous section), not the standard `SimpleSumeSwitch` one.

4.3.3 The final parser design

The final design is shown on figures 4.5–4.10: the top-level parser on figure 4.5 only calls the encapsulation parser (figure 4.6) and the SCION header parser (to the right), and the SCION header parser in turn calls several other sub-parsers as shown on the figures. This hierarchical structure is not only useful from the modularity aspect, but also helps avoid code duplication: the parametric sub-parsers used for skipping bytes, as well as the host address sub-parser, are instantiated multiple times in the design. Most importantly though, it makes it

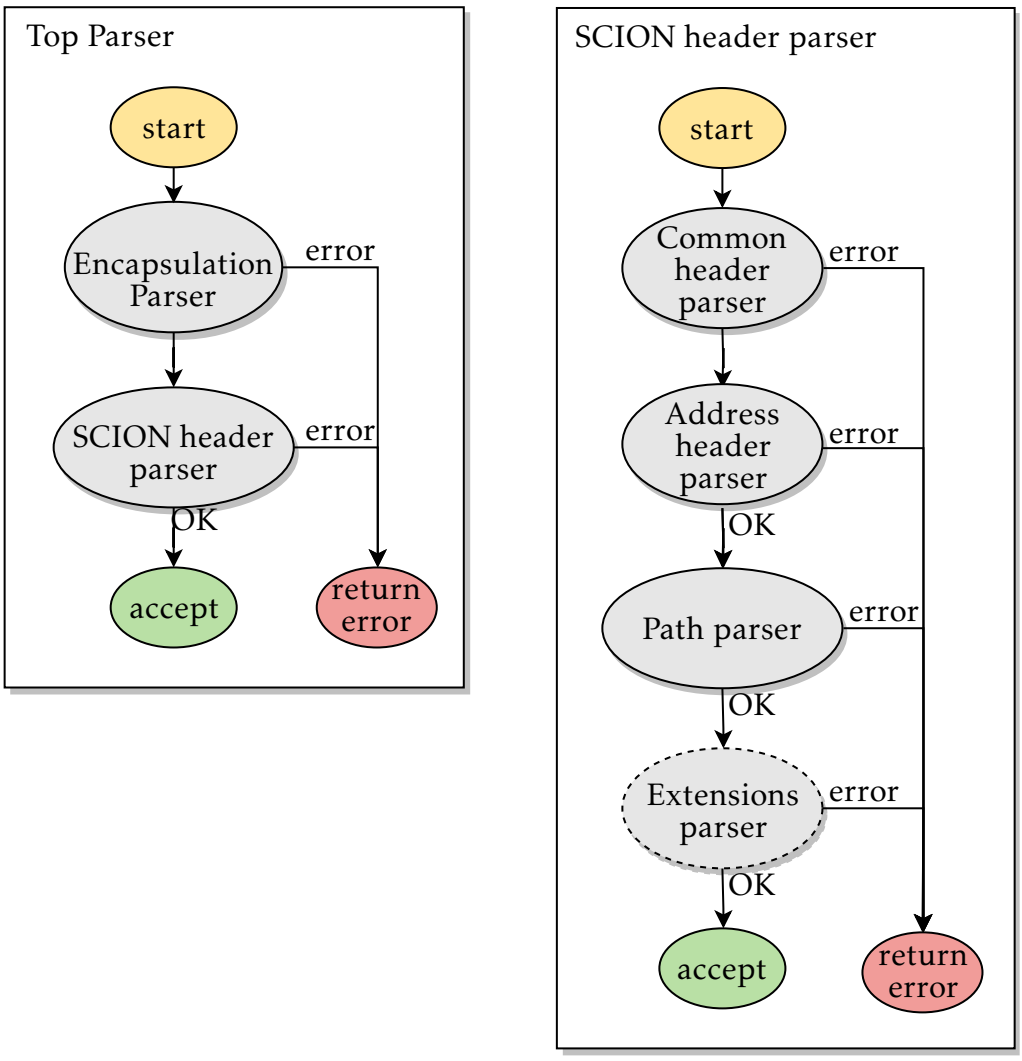

Figure 4.5: Left: The top-level parser state machine: IP encapsulation and the SCION header. Right: State machine for parsing the SCION header. (The extensions parser is currently unimplemented.)

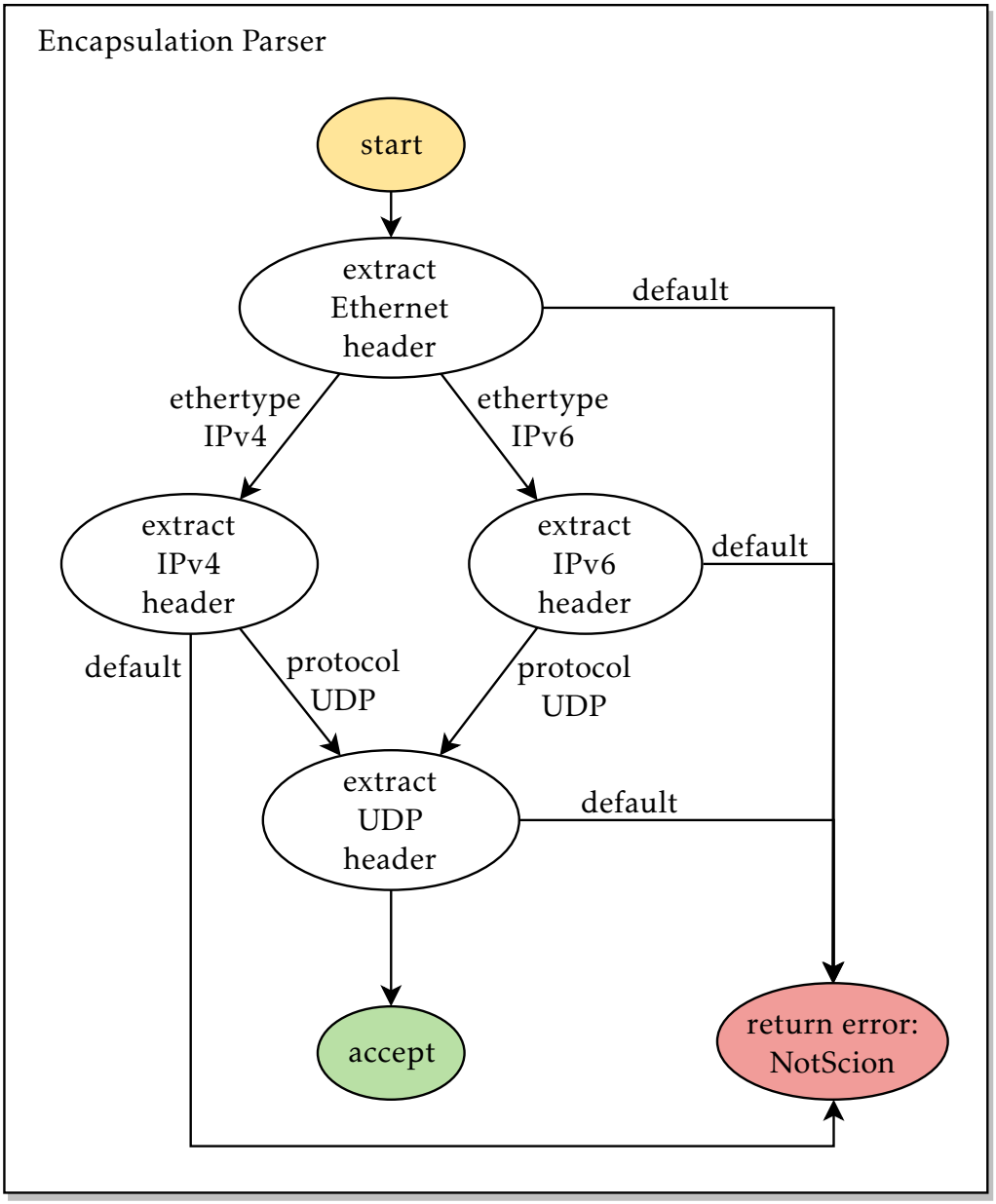

Figure 4.6: IP/UDP encapsulation parser state machine.

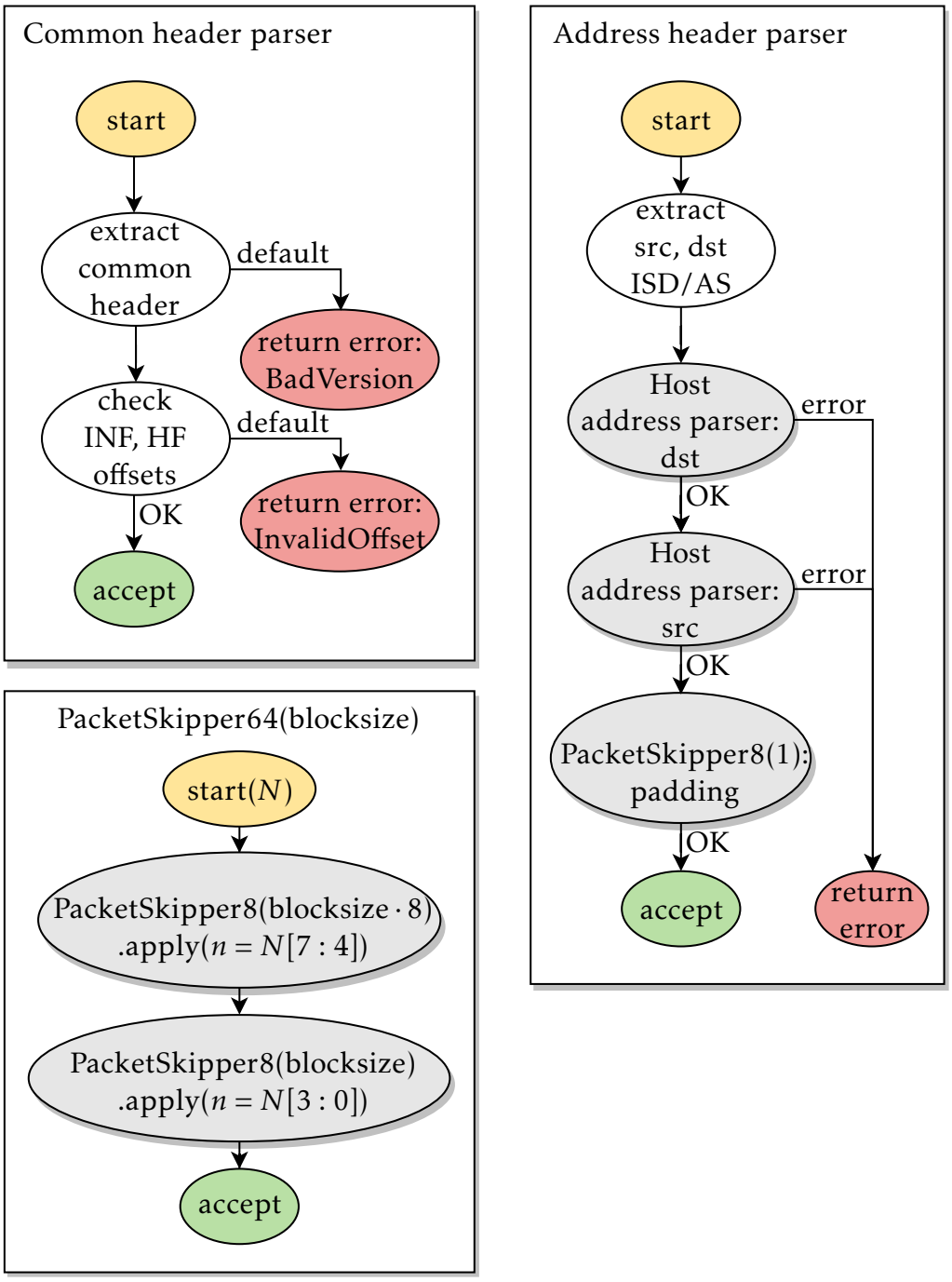

Figure 4.7: Top left: the SCION common header parser. Right: The address header parser. Bottom left: The NetFPGA implementation of a helper for skipping over 0–63 blocks (`blocksize` is a compile-time parameter).

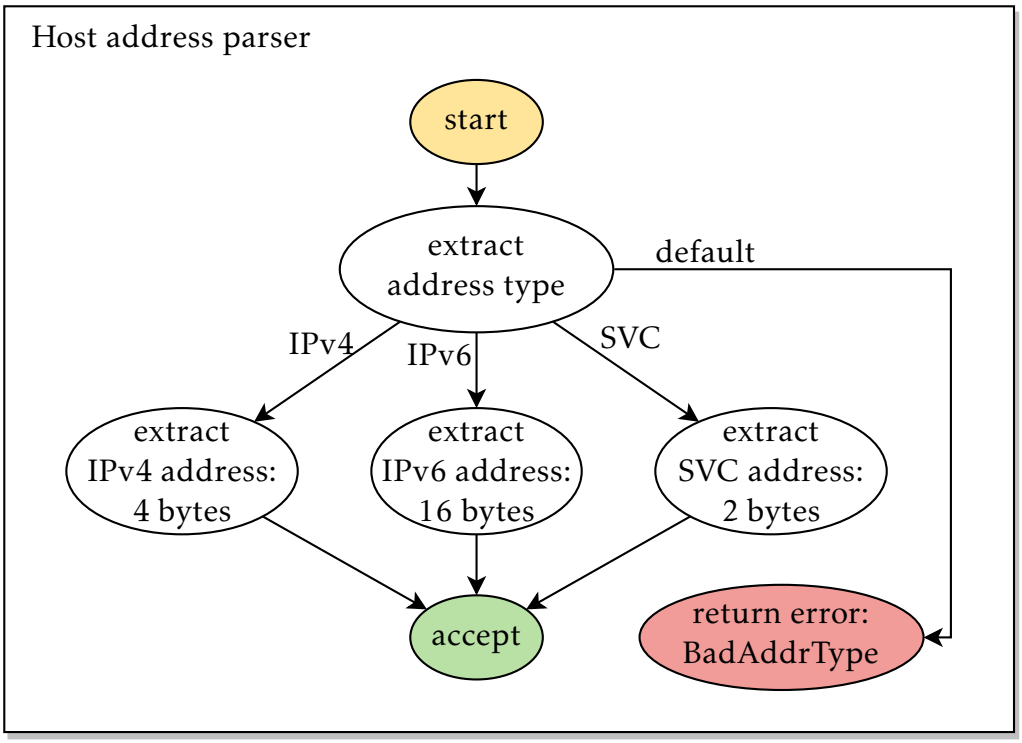

Figure 4.8: The host address parser: branches on address type.

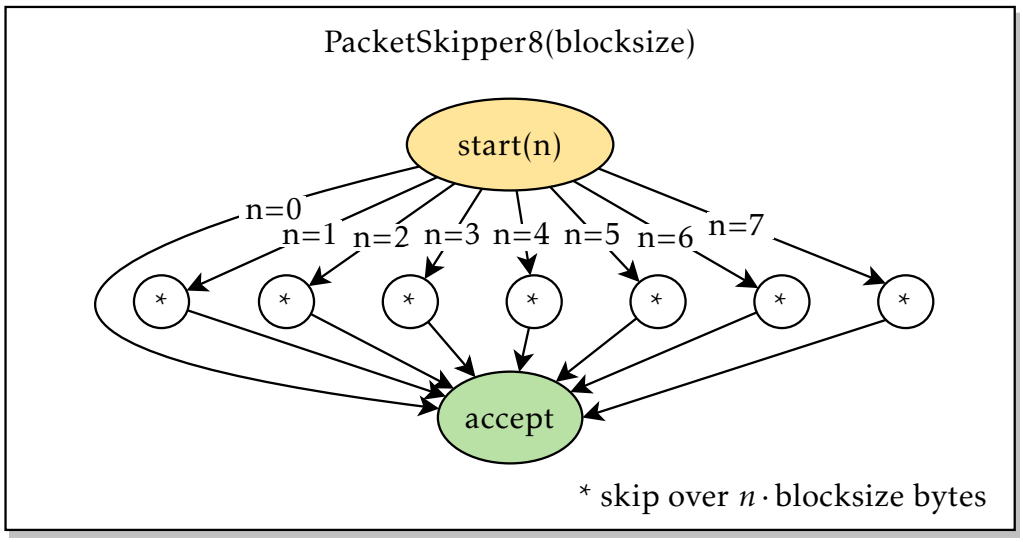

Figure 4.9: The NetFPGA implementation of a helper for skipping over 0–7 blocks (`blocksize` is a compile-time parameter).

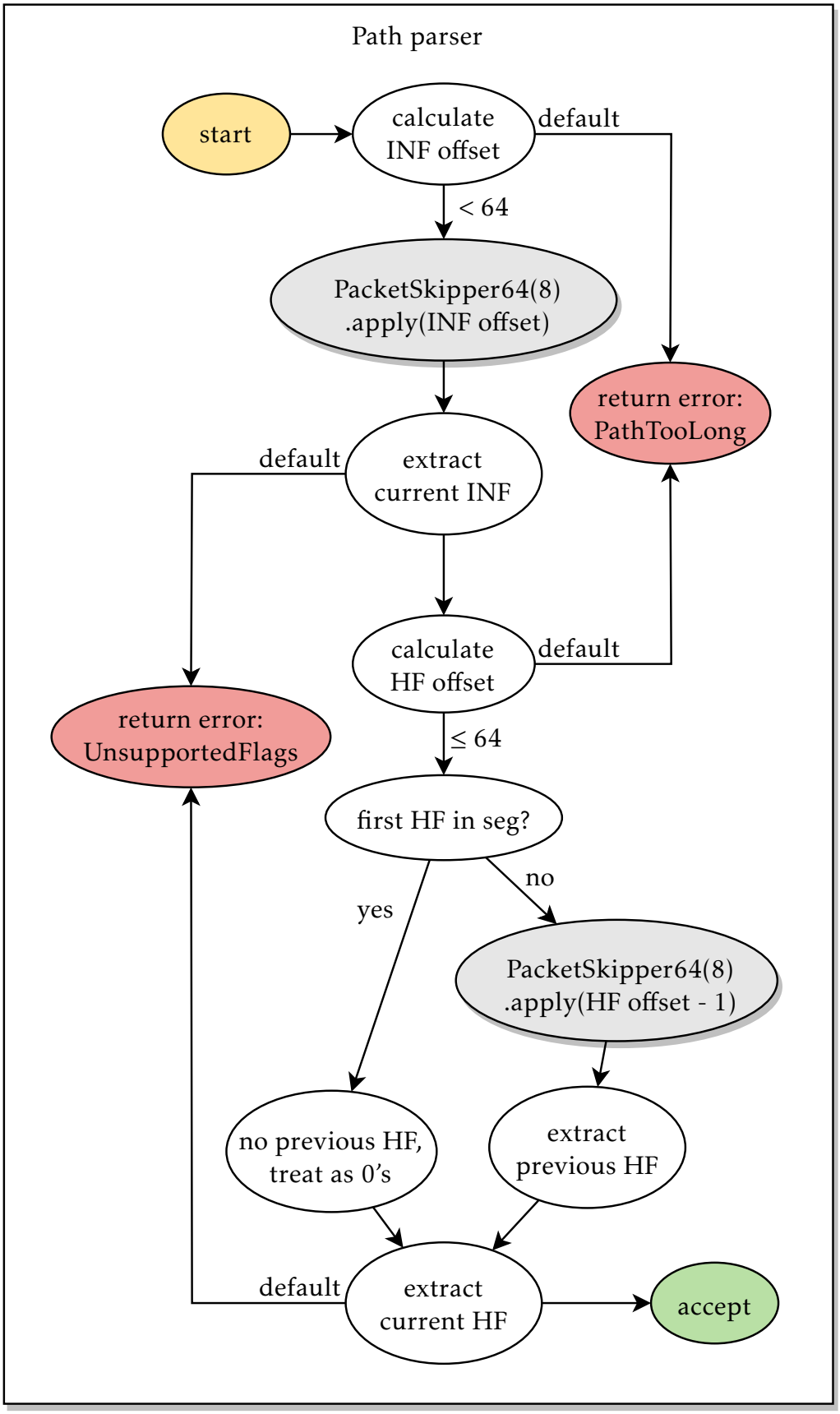

Figure 4.10: The SCION path parser.

possible for a human to reason about the correctness of the code: the flattened state machine would be impossible to even draw in any reasonable way (we tried). This way, one can abstract away the details of the sub-parsers when looking at the big picture, and thus have a better idea of the system.

4.4 Hop field validation

Before a SCION packet can be forwarded, we need to validate that this path, specifically using this hop field, is allowed. There are three parts to that: we need to check that the hop field used in the path is not expired, that the packet arrived at the ingress interface specified in the hop field, and that it was in fact created (and thereby enabled) by our AS, not a malicious party by checking the HF MAC. Here we briefly discuss each of those steps.

4.4.1 MAC verification

The MAC in SCION HFs is an AES-based MAC over the hop field and the previous hop field (excluding some flags) plus a timestamp, as outlined in section 2.1 and detailed in [4]. Therefore, to check the MAC, we need to add AES functionality to our P4 program.

Adding the AES external module

While implementing AES in P4 would in theory be feasible, in practice such an implementation would have severe performance problems. Therefore, we instead chose to include an external module in our FPGA design, and give access to it to our P4 code. Giving P4 access to external features is done via *externs*: an extern is a P4 function interface without implementation, and the compiler uses it to plug in an external module.

The first step was to find out how externs work: since the NetFPGA's documentation is rather scarce, the interface required for extern modules to work correctly with the P4 compiler was unclear and we needed to learn about it by trial and error. We did this by implementing a simple no-op module, which we then switched for a basic proof-of-concept AES-128 implementation.[6] This appeared to work – the AES encryption result was correct, and everything worked in the behavioural simulation. However, when deployed on real hardware, the NetFPGA would process the first packet (and successfully use AES), and then it would get "stuck" and not output any more packets. After long hours of debugging, we decided to invest the time to learn to debug using waveforms, i.e. the low-level simulation of the FPGA logic. Here, we found out that the AES module (which required 5 cycles to compute the AES) was occasionally being called every 4 cycles, despite indicating

[6] Unfortunately, we do not know the original source: we got this implementation from another student without any information despite asking.

to P4 (via an output bit) that it was not done yet. We therefore concluded that although this was not mentioned anywhere in the NetFPGA docs, P4 may input data in any cycle, not only after the previous computation has finished.

As such, the externs must be pipelined: in addition to the reset, clock, input data, and output data signals, externs must keep track of the signal indicating input validity, and output the result for every valid input in a FIFO manner. An "output valid" bit should be set whenever the result of a valid input is ready – this bit signals to P4 that it can take the data and resume execution of the caller pipeline. We confirmed this conclusion with the NetFPGA developers and sent a patch to the documentation.[7]

Checking the MAC

Once our AES extern was working, we were able to implement the MAC checking. Because we currently do not support shortcut paths with `VRFY_ONLY` HFs, or "long" HFs[8], we always need to calculate the MAC of one HF + previous HF chain per packet, which is at most one 128-bit block.

SCION uses the AES-CMAC function as specified by RFC 4493. The unconstrained version of AES-CMAC requires two AES-128-ECB operations for subkey derivation, plus one per block of data. In the special case of a single block, we only use the first subkey, so single-block AES-CMAC requires only two AES operations. Therefore, my implementation uses two AES cores.

Because SCION uses the same key for all HFs of a given AS, it would be possible to do the subkey derivation only when updating the key (which could be done by the control plane). With this optimisation, we would only need a single AES core per block. Thus, with the current HF size of 8 bytes, SCION can be forwarded with a single AES operation per packet.

4.4.2 Timestamp validation

In addition to checking that the HF MAC is valid, we also need to check that this HF is still current, and not expired. This is done by checking the timestamp and expiry interval encoded in the info field of the path segment. (This timestamp is also included in the MAC calculation, and thereby cannot be changed by the end host in an attempt to use an old path.) As the NetFPGA does not have an easily accessible clock, our implementation uses a register accessible from the control plane, and simply assumes that the control plane writes the current time into it often enough. SCION requires only second-level precision, so this is sufficiently

[7] https://github.com/NetFPGA/P4-NetFPGA-public/wiki/Workflow-Overview#p4-netfpga-extern-library

[8] The current version of SCION supports a "continue" flag in the hop field, which indicates that the hop field is longer than 8 bytes. There is no limit on the HF size. Because this causes significant issues in hardware implementations, the next version of SCION will only allow fixed-size hop fields.

precise. Real-world implementations should include an internal clock, so that the router behaves correctly even if the control plane is disconnected for a short time.

4.5 Area and timing constraints

This section requires a basic understanding of FPGA design. Refer back to section 2.5 for a brief introduction. TODO is that note needed?

The various costly workarounds as explained in the previous sections of this chapter, together with some questionable design choices in the P4-NetFPGA base design[9], take up almost all of the FPGA area.[10] Due to this, the place and route step of the design process has difficulty satisfying timing constraints.

Originally, in the full design, the longest path executed in one cycle had a delay of over 6 nanoseconds. The P4-generated module runs on a 200MHz clock, which requires all delays to be under 5.0 ns (200 MHz $\Rightarrow$ 5 ns per cycle). After more than a month of optimisation, we have been able to decrease the delay by a bit over 1 ns (i.e. 20%). However, we have not been able to cross the threshold: the largest delay in the full design is still 5.08 ns. At this point, the FPGA is too full for any further optimisations to help: all of our attempts just shifted the problem from one place to another, but did not solve it.

Note that this problem is more difficult to fix in our situation than in "normal" FPGA design: the P4 compiler decides how to divide the logic into stages, and because we have more logic than is usual due to the various workarounds, the compiler's decisions are not very good for our situation. If we had full control over the design, it would be possible to split the logic on the longest path into two stages, sacrificing a cycle in order to reset the delay (though that would create problems of its own). In other words, the particular difficulty with solving our case arises from the trade-off of using a high-level rather than a low-level language: by using P4, we cannot control the details of the pipeline, but on the other hand, we do not have to control the details of the pipeline.

Due to these issues, we have not been able to test our full design on real hardware; however, we have tested it in simulation, and we have tested two designs with a subset of features enabled, as discussed in the analysis in section 6.

When attempting to solve this problem, we have learned a great deal about writing P4 code with better timing characteristics. We present some of our insights in section 6.2.

[9] The P4-NetFPGA design includes a full Microblaze microprocessor in the FPGA. This alone takes up about 30% of the FPGA area.

[10] For context, the Virtex 7 FPGA present in the NetFPGA is often used for prototyping CPUs. To fill it up is a rather unexpected feat.

5 Evaluation

Due to inadequacies in the P4-NetFPGA toolchain as explained in 4 and especially in 4.5, we have not been able to test the full system. Therefore, we do not present results for the full design: instead, we show the data for a partial design, and we draw conclusions based on these and our knowledge of how composing systems in FPGAs affects performance.

5.1 Method

The aim of the evaluation is to determine the maximum throughput achievable with our system, as well as gain insight into the limitations of the system. In order to do that, we need to generate a large amount of traffic. We first attempted to do this using a host with a 10G NIC (which we had been using for debugging), but the NIC was not able to approach line rate with small frames: we only got about 6.2 Gbps over a direct host-to-host connection with 115B frames. Thus, we used the Spirent TestCenter traffic generator, which (though cumbersome to work with) is able to achieve line rate even with minimum-sized packets.

Due to the limitations of the traffic generator, we were only able to use a small set of deterministically pre-generated SCION frames, not complex randomly generated traffic patterns. However, this does not make it "easier" for the NetFPGA, as it processes each packet completely independently (except for queuing). It also ensures measurement reproducibility, and thus can be viewed as a good thing.

As mentioned in section 4.5, the full system does not meet timing constraints and thus cannot be tested in one piece. Therefore, we evaluated two subsets of the full system: a program with everything besides updating the IP overlay (i.e. parsing, HF validation, routing), and a program with everything besides HF MAC validity check (i.e. parsing, HF timestamp check, routing, IP overlay update). The two behaved identically (as can be expected with an FPGA design that meets timing), and thus we only present one set of numbers.

In all cases, the ingress traffic was sent on all 4 ports, distributed evenly. The output traffic distribution was also even (as determined by the egress interface IDs pre-set in our packets). All of the measurements were performed multiple times and showed consistent results. Each measurement was run for at least 45 seconds (sometimes more, up to 20 minutes), and we checked that the throughput was steady for the vast majority of all measurements.

5.2 Throughput

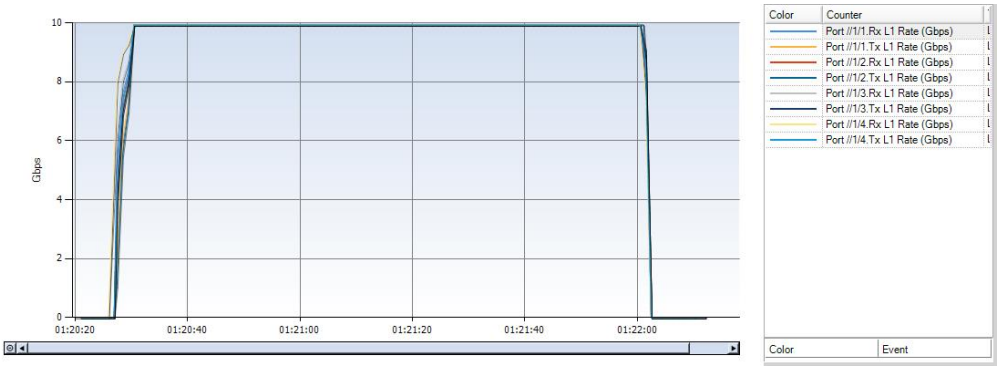

Figure 5.1: Throughput per port over time, with 1500B frames. Stable at 9.93 Gbps per port, i.e. **39.72 Gbps** in total.

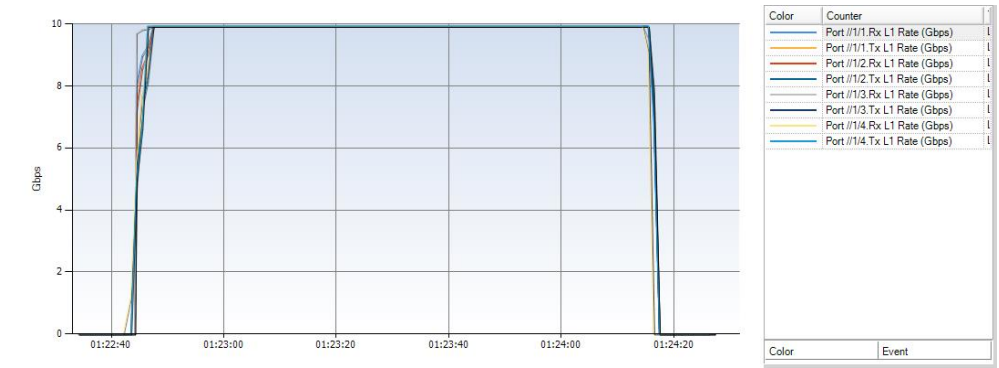

Figure 5.2: Throughput per port over time, with 115B frames. Stable at 9.93 Gbps per port, i.e. **39.72 Gbps** in total.

Plots 5.1 and 5.2 show the throughput per port, as measured by the traffic generator, for 1500B and 115B frames, respectively.[1] For both frame sizes, after the traffic ramp-up both the transmit and the receive rate stabilised at 9.93 Gbps per port, i.e. **39.72 Gbps** in total.

5.3 Queue sizes and packet loss

Our program exports statistics about frame counts and queue sizes, as seen by the NetFPGA. We can use this data to understand whether packet loss is occurring. Tables 5.1 and 5.2

[1] We apologise for the low quality of the graphs. Unfortunately, the traffic generator we used has no way of exporting time series data, and thus we had to resort to screenshots.

show the packet loss observed throughout the duration of the experiments, for 1500B and 115B frames. The frame counts as reported by the NetFPGA are consistent with the frame counts reported by the traffic generator.

Table 5.1: Packet loss with varying load, with 1500B frames.

Load / Gbps	Tx frames	Rx frames	Loss
8	39094558	39094558	0
16	79240424	79240424	0
24	122369587	122369587	0
32	157131302	157131302	0
40	198083742	198083742	0

Table 5.2: Packet loss with varying load, with 115B frames.

Load / Gbps	Tx frames	Rx frames	Loss
8	334798373	334798373	0
16	674862021	674862021	0
24	990142312	990142312	0
32	2125564720	2125564720	0
40	1631918616	1631918616	0

Note that the 0% loss in all measurements may look like an error, but that is not the case – refer to section 6.1 for an explanation.

6 Analysis

6.1 Discussion of the measurements

6.1.1 Throughput

Our measurements show that the various parts of our design, in particular the per-packet cryptographic MAC check, run at line rate (i.e. 40 Gbps). Further, the small maximum queue sizes, which do not grow indefinitely, indicate that the processing is not the bottleneck and our design could handle more traffic with faster network interfaces. Though this is not the full production-ready SCION router design, these results indicate that maximum throughput can be obtained with the full design too: with pipelining, adding independent processing steps does not increase throughput (see section 2.5). As the missing steps are independent, and computationally less expensive than the parts we tested, we can be sure that the full design would still run at 40 Gbps. Thus, we conclude that running SCION at line rate can be achieved (and has been achieved by this project for the relevant parts).

6.1.2 Packet loss (or lack thereof)

The observed zero packet loss is a surprising result, and when we obtained it, we first thought we had made an experimental error. However, that turned out not to be the case. This result in fact makes sense: in general, packet loss happens because there is no buffer space for queuing packets on the input or on the output, either due to processing delays or due to uneven distribution of traffic. In our case, there are no processing delays because on the FPGA everything happens in constant time, with the exact number of cycles known at compile time. Further, the traffic generator outputs packets evenly and deterministically, not in bursts, and thus on the input, the traffic distribution is very even both in time and per port. Since we pre-set the egress interface IDs in the generated traffic to also be evenly distributed, the output traffic is also distributed evenly. Thus, neither the input nor the output buffers get full, and we are able to achieve zero packet loss.

6.2 Guidelines for high-speed packet processing

6.2.1 Types of problems suitable for P4

P4 is not a general-purpose language, but it does allow quite general programs for packet processing. P4 programs are expressed as imperative steps for processing a single packet, and thus are best used for tasks where packets are processed mostly independently: state shared among multiple packets is limited to a few primitives, such as register arrays. In order to enable processing at line rate, P4 limits the control flow statements available: there are no loops (i.e. one can only write programs that execute in constant time), no locally-scoped arrays, and even branching statements are only usable in specific places. Thus, P4 is best suited for problems that do not require a complicated control flow or a large amount of state shared between packets. The advantage of this is that programs satisfying these constraints can be expressed quite simply, and automatically run at line rate (assuming the timing is met – see below), with the details of the design, such as pipelining, handled by the compiler.

Of note is that different P4 targets may have different limitations. For example, an FPGA target can (assuming a good toolchain) run a larger set of programs than e.g. an ultra-high-speed P4-programmable switch: although P4 is supposedly target-independent, in reality different targets may have different limitations. Thus, P4 programs handling complex protocols (such as SCION) may be better suited for FPGA-based implementations than programmable switches.

6.2.2 Writing P4 code that meets timing

This section requires a basic understanding of FPGA design. Refer back to section 2.5 for a brief introduction. TODO is that note needed?

During the optimisation of our code we learned a great deal about writing P4 code that often leads to lower signal propagation delays on the NetFPGA. Since timing is not an exact science, it is difficult to extrapolate this knowledge. Nevertheless, the following sections list some of our insights.

Reducing data dependencies

The structure of a P4 program, and especially the flow of data, has a large effect on the data paths in the resulting hardware design, and thus also on timing. Data processed in many interdependent steps, where the input from a previous step is needed to execute the next step, will accumulate more delays and thus result in tighter timing.

This general observation may be enough to hint at concrete ways to improve a P4 program. Additionally, it can also be specifically applied in two common cases: `inout` parameters and branches.

P4 provides several methods for separating large programs into smaller pieces. The most interesting ones are actions, functions, and controls. (Parsers use their own keyword and a different syntax for the body, but for the purposes of structuring P4 programs, they are equivalent to controls.) All of these can accept parameters, which are copied by value. These parameters have a direction: they can be marked `in` if they should be copied from the caller scope at the beginning of the block's execution, `out` when they should be copied out to the caller at the end, and `inout` if they should be copied both on the way in and out. While declaring parameters `inout` is often quicker, this comes at a cost: it introduces a long path where the same data needs to go into and out of the block, accumulating all the timing delays. When a parameter is only `in` or only `out`, the compiler may find more opportunities for optimising logic, splitting and parallelising a computation, or similar. In our case, we originally had an `inout` parameter passed along through all of our sub-parsers, and since the SCION parser is on the critical path, this was causing issues. Cleaning up the sub-parsers to use an `in` parameter or an `out` parameter, and only when really needed, helped us save about 0.2 ns on the critical path, which is quite noticeable.

The case of branching is quite similar: code that conditionally should or should not execute based on a result of a complex computation needs to wait for that computation, thus accumulating the delays from it. Code like `if [long computation 1] then [long computation 2]` is likely to have worse timing results than a rewrite of the form `[long computation 1]; [long computation 2]; [combine the results]`. This may look counter-intuitive: in the first case, `[long computation 2]` will only be performed when needed, while in the second case, we perform both computations even if we don't need to. However, this thinking only applies in software engineering, not hardware engineering: in P4, if the compiler infers that the two computations are independent, may parallelise them, thus using more FPGA area but not more time. The accumulated timing delays will also be only the maximum of the delays from the long computations plus the delay from the combining, while in the original code, the delay from computation 1 propagated into computation 2 and thus the total delay was the sum of the two, not the maximum plus a bit.

P4 tables

Matching against a CAM table (the `exact` table type) adds about 0.1 ns. Out of the elementary operations in P4 that we tested, this is the most expensive one. (Note that we have not tested TCAM, i.e. `lpm` tables. We assume that those would be even more expensive.) Thus, a design with many tables may run into timing problems. Therefore, it may be worth it to consider "squishing" those tables into one big table: instead of having something like `table1.apply(); table2.apply();`, consider making one bigger table with a composite key and a composite action that combines the two smaller tables into one. Obviously, this may require restructuring the control flow of the program, which may change

other things, so this is not a "one size fits all" solution. Furthermore, the NetFPGA stores action parameters inefficiently: if a table has multiple actions, on the NetFPGA the entry will need as many bytes as the sum of the sizes of parameters of all actions, not the maximum as one would expect. Thus, a table with multiple actions can end up being very large and taking up a lot of FPGA area, thereby making routing more difficult and potentially causing timing issues in other parts of the design.

Externs can be expensive

Obviously, the implementation of an extern affects how much delay it adds: for example, our AES implementation adds about 0.4 ns. Less obviously, even the simpler of the externs included with the P4-NetFPGA repository are more expensive than we had expected. A register write adds about 0.03 ns, which is small but adds up. Thus a program that relies very heavily on even simple externs may run into problems: it is not too hard to imagine a program that uses a register write 20 times, which would add up to a delay of more than 0.5 ns, i.e. more than 10% of the available 5 ns.

6.3 Implications for the SCION protocol

The information in this section arose through discussion with multiple interested parties. Thus, in this section we summarise the discussion, but do not claim sole credit for the content.

6.3.1 What SCION does right

The most obvious hardware-relevant choice in SCION, and the one that drove much of the design process, is packet-carried forwarding state. Besides avoiding entire classes of problems by eliminating state on routers, this also saves circuit area used for lookup tables. In the current Internet, IP prefix table size is quickly becoming a limiting factor. Equipment able to handle the tens of thousands of prefixes needed for backbone routing is extremely expensive, because large content-matching tables are difficult to manufacture. Further, the large area and number of parallel comparisons needed to match every packet in such huge tables translate into high electricity consumption. Thus, exchanging a large CAM table for an AES core is a good decision that is likely to significantly decrease hardware manufacturing costs and power requirements.

Another very relevant consideration in hardware is avoiding variable sizes. The SCION header is variable-length due to the need to put forwarding state into the packet (so that it does not need to be on the routers, as per the previous paragraph). However, given this fundamental need, it does specify header alignment requirements, which makes this at least somewhat easier to handle. Thus, given

that we need a variable-length header, being strict about header alignment is a good idea.

Thirdly, and perhaps most importantly, despite the variable-length paths, and despite providing path-level authorisation, SCION only requires each border router to only process a constant amount of data. The path authorisation property can be verified by MAC-ing only a constant number of blocks, and the SCION common header contains offsets to the current hop field. Therefore, every border router can immediately jump to that part of the header without parsing or processing the rest of the path. This is absolutely necessary for being able to obtain line rate.

6.3.2 What could be improved

Define maximum sizes

Currently, the maximum path length in the packet header is unreasonably large (the 8-bit offset fields allow for up to 256*8 = 2kB offset), and the hop field length is virtually unlimited (the continue flag signals whether we need to keep reading, and there is no maximum length defined). In hardware, in order to enable pipelining, we need to create buffers of the "worst case" size. This consumes FPGA area (thereby increasing costs and possibly energy consumption) and, due to processing in fixed-size blocks, increases latency. Therefore, these should be limited. We propose the following:

- limiting the maximum number of hops to some "reasonable" value, such as 64
- limiting the maximum hop field size to e.g. 16 bytes

Avoid variable lengths

The SCION header contains a large amount of information, and thus, for maximising goodput, it allows for variable lengths in many cases. At the moment, SCION limits the variability to an extent: all header sections, as well as the size of hop fields, must be a multiple of 8 bytes. This is a good start. However, not knowing the size of important data structures, such as hop fields, in advance, unnecessarily complicates matters. While there are multiple cases of this issue in the SCION header (another notable one are the host address fields), the hop fields, due to their involvement in the routing and the cryptographic MAC checking, are the most critical case. Thus, we propose that a single size for the hop fields should be selected, instead of allowing multiple options.

Avoid implicit lengths

Because SCION is L3-agnostic, the source and destination end-host addresses are not fixed-size. Currently, the address type field in the header implicitly indicates

the length. Though the intermediate routers do not need to understand the end host addresses, they need to know the length, so that they can parse the header correctly. This means that all routers on the path need to have a lookup table that maps all known types to the address length. This table may get unnecessarily large if more address types are added, and, more importantly, whenever a new address type is added, all routers in the network must be upgraded to know about it. Since upgrading equipment, especially hardware, is very expensive, we propose that the address length should be explicitly encoded in the header alongside the type.

Consider re-thinking MAC chaining for peering paths

The MAC chaining in SCION is used in order to ensure path authorisation: a path should only be usable if the on-path ASes have explicitly allowed it. While MAC chaining is an elegant way to achieve this property, it is not the only option. The number of blocks that need to be MAC-ed is a relevant consideration for hardware design, and thus it might be beneficial to evaluate different options for achieving path authorisation, especially those that would result in fewer blocks that need to be MAC-ed. We do not have a strong recommendation in this case, as the security implications of changing the MAC scheme must be carefully considered; however, we do believe that further investigation may prove useful.

6.3.3 A look at the future

Hardware-based SCION routers

A basic SCION border router does not require any state other than configuration, so the packet processing can be easily parallelised using multiple FPGAs. The only thing that needs to scale vertically is the switching itself, which can be done with speeds of several hundred Gbps with relatively affordable hardware (as long as we don't require the hardware to be programmable). Therefore, using several higher-throughput FPGA-enabled NICs for the packet verification and update, plus a non-programmable MPLS or IP switch for the switching (i.e. to handle buffering/queueing at high speeds), we could in principle scale to very high rates even without creating an ASIC. As SCION becomes more prevalent in the real world, this might be of particular interest to Internet exchange points (IXPs).

The SCION protocol

An upcoming SCION data plane redesign is currently in the works, and it will incorporate some of our suggestions from the previous section. In particular, we are aware of the following upcoming changes:

- The hop field size will be fixed, likely to 12 bytes. (Thus, the alignment requirements will be changed to 4-byte rather than 8-byte alignment.)

- A maximum path length will be defined, somewhere on the order of 64 hops. This should be sufficient, as AS-level paths in the current Internet tend to be around 10 hops long. We anticipate that on some platforms, it may be necessary to limit the path length to 32 hops, but future-proofing the specification by settling on 64 is the safer option.
- The address length will be included in the header. The current proposals contain a two-bit field (which can encode 4 different lengths), and a two-bit field for the address type scoped by the length (so that the address type is uniquely identified by the length + type pair).
- Ongoing discussion for about the HF MAC chaining is in progress.

7 Future work

Completing the feature set / Adjusting for newer SCION versions

Due to difficulties with the NetFPGA's P4 toolchain, we currently do not implement everything in the current version of SCION, which means that a larger than necessary fraction of traffic must be handled by the CPU. Therefore, an obvious avenue for improvement would be to implement the missing parts of SCION, namely handling UP segments, shortcuts and peering paths. The handling of shortcuts and peering links may be simplified in the next version of the protocol, which will require modifications to this project's code anyway. Therefore, it is better to first accommodate the new protocol version (once it is finalised), and add support for all link types afterwards.

Additionally, while "plain SCION" is already a significant improvement over today's Internet, features built on top of it could provide even more benefits. Examples of such systems are COLIBRI, a system for bandwidth reservation that guarantees bandwidth even in the face of network overload and DDoS attacks, and EPIC, a source authentication mechanism designed to run at line rate.

It would be interesting to see how it would go to implement these more complex schemes in hardware, and whether high throughput can still be achieved without unreasonable costs.

Researching the resource consumption

A core part of the design of SCION is the assumption that per-packet cryptography is cheaper than state on routers (e.g. in the form of routing tables), especially as the Internet continues to scale up. While this is very likely true for configuration and therefore also personnel costs, it is less obvious for hardware manufacturing costs, as well as energy consumption. Costs of running network equipment are nowadays largely due to electricity, plus there are environmental concerns to consider, and therefore an exploration of SCION's power requirements would be very valuable.

In theory, not requiring large prefix match tables such as those in IP routers decreases the chip area and therefore its manufacturing cost and power consumption. However, the added cryptographic verification of every packet is a complex operation that may increase it. Thus, the overall effect of the two is not certain.

It has been theoretically predicted that SCION may be able to use overall less energy per packet [1], but measuring it and seeing it in practice would be much more tangible.

The initial results of this project indicate that performing the AES operations may mildly increase in-device packet processing latency (by the order of nanoseconds, which is negligible), but is cheaper in terms of FPGA area and thereby also energy requirements. However, due to the discussed issues with the NetFPGA toolchain, we have not been able to look into the details. It would be of great interest to find concrete numbers and show the exact improvements that can be obtained by using SCION's approach. Moreover, with a concrete hardware design at hand, we may be able to learn more about the trade-offs and find opportunities for further power consumption-related optimisation of the SCION protocol.

Other targets, higher throughput, other high-speed SCION applications

Due to keeping portability in mind during the development, we anticipate that porting our border router to other targets should not be too difficult. Having a single SCION implementation usable in multiple scenarios could be very useful in the future. Some targets of interest are for example future FPGA-based NICs, programmable switches, or the P4 to DPDK compiler. Especially the last one would be a useful direction to go in, as DPDK runs on standard PCs, and therefore porting for the P4 to DPDK compiler would give us a high-speed switch that does not require special hardware.

With the NetFPGA as our hardware target, we are limited to 40 Gbps by its 4 10G ports. If not for that limitation, our design could achieve up to 51 Gbps without any change (it processes 256 bits per cycle at 200 MHz clock ⇒ 256*200e6 = 51e9 TODO fix the numbers). If not for the numerous costly workarounds, our design could run at a higher frequency, and with greater parallelism. Therefore, by porting this project to a faster target with a better toolchain, we might be able to achieve several hundred Gbps with a single FPGA of similar parameters as what we are using now.

Thanks to the emphasis on modularity, our "SCION as a library" code could also be used in many applications different from the border router. Apart from the applications in themselves, working on these would also help us improve the code base by challenging our assumptions about modularity and generality.

8 Conclusion

Our work has proven that high-security Internet routing protocols, and especially SCION, can feasibly be forwarded at high speeds. We see no major obstacles for implementing a SCION border router with very high throughput (on the order of terabits per second). The minor issues identified, such as defining maximum sizes, fixing the hop field size, avoiding implicit lengths, and considering the trade-offs of MAC chaining, will be addressed by an upcoming data plane redesign that incorporates our suggestions. Therefore, we conclude that SCION is suitable for hardware, and thus can be considered for large-scale deployments.

This project also shows that FPGA-based programmable data planes can be made to perform at line rate even for relatively complex protocols. Further, P4 is an adequate language for programming such, and though it has some issues, it is a useful tool that simplifies development and maintenance, and thus enables researchers, enthusiasts, and commercial entities to take their ideas into the real world. We are looking forward to further developments in the P4/SDN ecosystem, especially more mature hardware targets and toolchains. Once they are available, we believe that there is much promise in extending this work with those.

Bibliography

[1] C. Chen, D. Barrera, and A. Perrig. Modeling data-plane power consumption of future internet architectures. *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, Nov 2016.

[2] S. Goldberg. Why is it taking so long to secure internet routing? *Communications of the ACM*, 57(10):56–63, Sep 2014.

[3] R. S. Nikhil and Arvind. What is bluespec? *ACM SIGDA Newsletter*, 38(23):1–1, Dec 2008.

[4] A. Perrig, P. Szalachowski, R. M. Reischuk, and L. Chuat. *SCION: A Secure Internet Architecture*. Springer International Publishing, 2017.

[5] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 34(5):32–41, Sep 2014.

A [1/3] Glossary

meep, do I really need this? and if yes, where do I put it? TODO

A.1 TODO FPGA

A.2 TODO NIC

A.3 DONE target

A hardware or software platform on which to run a program, for example a software switch or an FPGA. In the context of P4, a target is a platform and *switch architecture* pair, as the same platform may support more than one architecture: for example, the NetFPGA SUME platform supports the `SimpleSumeSwitch` and `XilinxStreamSwitch` architectures.