Universiteit Leiden
The Netherlands

Bachelor Computer Science

Practical Equivalence Checking of P4 Packet Parsers

Jort van Leenen
J.P.van.Leenen@umail.leidenuniv.nl

Supervisors:
Dr. T.W.J. Kappé and Dr. J.J.M. Martens

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
https://liacs.leidenuniv.nl

08/07/2025

Abstract

In computer networks, packets function as structured containers for transmitting information. Packet parsers validate these packets in devices like packet switches and firewalls. P4 is a domain-specific programming language that can specify such parsers in software-defined networks. As with any software system, bugs may be present, or improvements are to be made. These subsequent modifications may introduce discrepancies relative to the intended specification. To address this, we introduce OCTOPUS, a tool for the practical equivalence checking of packet parsers written in P4. Here, practical signifies a runtime suitable for execution on a typical, modern personal computer. The tool additionally produces a certificate that enables manual verification of the result's correctness.

Contents

1. Introduction

The Internet can be considered the largest engineered system by humankind [1, Ch. 1]. By 2023, over two-thirds of the global population was expected to have Internet access, with networked devices projected to exceed three times the global population [2]. Predictions indicate this trend will persist [2]. As the Internet continues to grow more interconnected and diverse, it remains a key focus of active research.

Like any other extensive system, the Internet relies on many subcomponents for its functioning. One of these subareas, including its ongoing research, concerns packet parsers. In a *packet-switched network*, such as the Internet, hosts (*i.e.* devices) communicate by sending packets through a network of communication links and packet switches. A *packet* is a structured container of information comprising two components: a header and a payload [1, Ch. 1]. The *packet switch* is responsible for forwarding packets to their destinations, indicated in the header field(s).

As an example, consider the illustration of a UDP packet with its header and payload components in Figure 1. It depicts how a UDP packet's header consists of four fields, including a destination (port). The data field represents the payload of a UDP packet. Due to the layered organisation of protocols [1, Sec. 1.5], *i.e.* the Internet's *protocol stack*, an example of a payload in a UDP packet could very well be another packet.

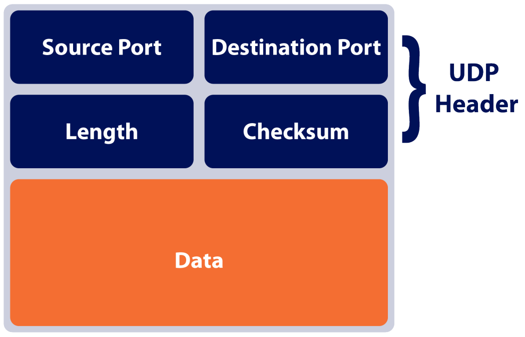


Figure 1: An illustration of the structured information as contained by a UDP packet. The top four blocks represent the fields comprising the packet's header. The bottom block represents the packet's payload.

A packet switch receives a packet as a stream of bits on its incoming link(s). It then has to decide whether the packet is valid, and, if so, to which outgoing link(s) to forward it. To make this decision, the stream of bits must be parsed into the packet's components and respective field(s). This is performed by the packet switch's *packet parser*.

Packet parsers are not limited to use in packet switches; they are also essential in another class of devices known as *middleboxes*. These devices operate on the data path between source and destination hosts and perform functions beyond standard packet forwarding [1, Sec. 4.5]. A subcategory of middleboxes performs security services. For instance, consider a *firewall*, which may block traffic based on specific values of header fields. Similarly, an *intrusion detection system* (IDS) could inspect header fields and payload content, possibly up to a particular layer or "depth". By their very nature, these devices also depend heavily on packet parsing.

Given the central role of packet parsers in both enabling communication and enforcing security, it is critical that they behave according to their functional specifications. Nonetheless, implementation bugs do still occur, and these can lead to incorrect behaviour or security vulnerabilities. Searching for the keyword "packet" in the CVE database [3], a widely used reference for publicly disclosed cybersecurity issues, returns various results. Consider `CVE-2020-1350`, a vulnerability where, due to an integer overflow in a function responsible for parsing DNS response types, remote code execution (RCE) became possible in a wide range of WINDOWS SERVER editions. Another example is *Ripple20* (`CVE-2020-11896`–`CVE-2020-11914`), a set of vulnerabilities discovered in the IP stack implementation found in a lot of embedded systems. The consequences ranged from RCE to denial of service (DoS) in a variety of devices, including medical applications. Both examples underline the importance of applying formal methods to ensure software correctness [4].

P4 is a domain-specific programming language used to define packet processors, including the behaviour of their packet parsers. As with any software system, P4 programs are susceptible to bugs. They may also undergo modifications over time. These changes can introduce unintended deviations from the intended specification.

To address this problem of unintended deviations, we present Octopus, a tool developed for practical equivalence checking of packet parsers written in P4. The tool aims to verify, within reasonable computational bounds, whether two P4 parsers are behaviourally equivalent. This is relevant as it allows one to compare different revisions of the same parser, representing the possible subsequent modifications. In addition, Octopus generates certificates that facilitate verification of the correctness of its findings. As an example, suppose that a version of a P4 packet parser is deemed correct concerning the intended specification. Now, a programmer has rewritten (parts of) the code in an attempt to improve its performance. Octopus can be used to check whether the two implementations are behaviourally equivalent, *i.e.* no bugs or other deviations from the specification were introduced.

1.1. Thesis Overview

This thesis investigates the problem of practical equivalence checking for packet parsers defined in P4. The primary question we aim to answer is: given an intentional modification to a parser, is its behaviour modulo this deliberate change preserved, *i.e.* equivalent? Beyond constructing a tool for equivalence checking, our goal is to provide guarantees on the correctness of the results produced by this tool. We will attempt to do so by generating certificates alongside the equivalence checks that back the tool's findings.

We begin by presenting the necessary background, including relevant foundational and theoretical concepts, as well as a review of related work, to position our contribution within the existing research landscape. Building on this foundation, we present our methodology for achieving practical equivalence checking. We then describe the design and architecture of the tool, along with its usage and supported syntax. Finally, we evaluate the performance of the tool in comparison with related work, draw conclusions from our findings, and outline directions for future research.

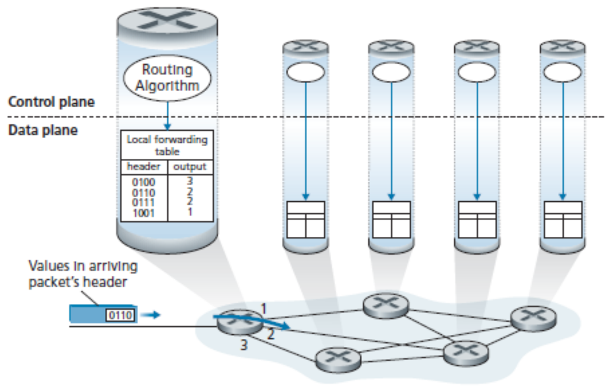
2. Background and Related Work

Software-Defined Networking Within the Internet's protocol stack, the *network layer* is responsible for enabling communication between hosts; at this layer, packets are transferred from a sending to a receiving host. To achieve this, the network layer can be viewed as consisting of two interacting parts: the data plane and the control plane. Within what is considered the *data plane*, it is determined whether and how to forward a packet from an incoming to an outgoing link. Forwarding decisions are made using a *forwarding table*, which maps one or more packet header fields to an output link. These decisions are local to each router. In contrast, the *control plane* embodies network-wide logic that coordinates these local actions to establish end-to-end delivery paths.

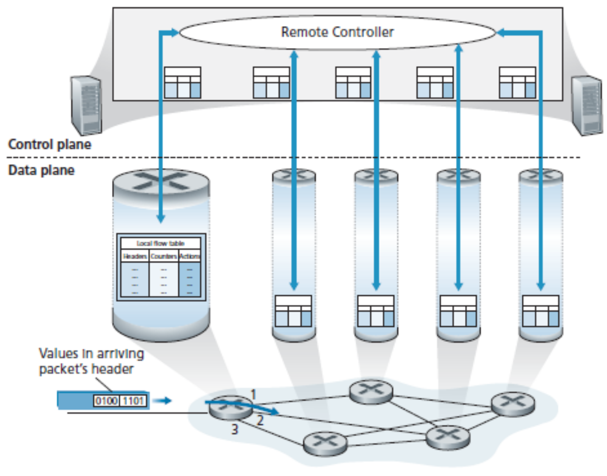
Software-defined networking (SDN) is an emergent paradigm within the field of computer networking. Its main promise is better manageability [5]. In a traditional computer network, the control and data planes are vertically integrated, monolithically on each device, and configurable through an interface, often a vendor-specific CLI or GUI. See Figure 2a for an illustration. On the other hand, in SDN, the control and data planes are separated. A logically centralised controller implements the control plane. The devices in the data plane, such as routers and switches, become simple forwarding devices. This is also where SDN finds its name. Namely, the control plane, which forms the heart of the network, is defined by a software-based controller. In practice, this is often achieved through a distributed control system to maximise performance, scalability, and reliability [6], [7]. See Figure 2b for an illustration.

OpenFlow and P4 Recall that forwarding tables are used in the data plane of a traditional network. In SDN, the software-based controller enables a more universal paradigm called *generalised forwarding* [1, Sec. 4.4]. This approach employs *match-plus-action* tables, which enable more expressive packet matching across multiple fields, protocols, and layers, and which support advanced actions such as load balancing and firewalling.

In practice, the OpenFlow protocol [8] pioneered the match-plus-action table abstraction. It remains a widely used protocol for defining communication between the controller and the data plane [5]. The protocol is used between an SDN controller and SDN-enabled devices; essentially, the tables defined by OpenFlow act as an API through which the behaviour of a packet switch can be programmed. While expressive, this model offers only limited programmability. For example, OpenFlow specifies a fixed set of protocols, restricting extensibility. One could envision a more expressive model that incorporates constructs familiar from general-purpose programming languages, such as variables, conditional logic, and functions. To this end, a domain-specific programming language for *programming protocol-independent packet processors* (P4) [9] was introduced. P4 enables the specification of packet parsing, processing, and forwarding logic in supported devices, without relying on fixed rules, such as those imposed by OpenFlow.



(a) A network where the control and data planes are vertically integrated [1, Ch. 4].



(b) A network where the control and data planes are separated [1, Ch. 4].

Figure 2: A comparison of two network types. In (a), a network is depicted in which the control and data planes are vertically integrated, *i.e.* monolithically on each device. On the other hand, (b) shows a network where the control and data planes are separated. It is an example of a software-defined network.

Verification and Formal Semantics of P4 As conceptualised in Section 1, software may deviate from its intended specification, potentially leading to security vulnerabilities or other issues. Programs written in P4 are, by definition, also vulnerable to these problems. Formal methods were proposed as one means to mitigate such risks. Applying formal methods to P4 is particularly promising due to the language's expressiveness and the inherent complexity of networked systems, which can allow subtle bugs to go unnoticed and/or untested by developers [4].

To assist developers in verifying the functional correctness of their P4 programs, several push-button verifiers have been developed. These tools, given an annotated P4 program, are capable of checking user-specified properties with minimal manual intervention. Examples include p4v [10], ASSERT-P4 [11], and Aquila [12]. Each allows the specification of functional properties through a manner of annotation. The annotated programs are then translated into intermediate representations (IRs), such as guarded command language (GCL) [13] and C, and verified using formal techniques: verification condition generation in the cases of P4V and AQUILA, and symbolic execution in the case of ASSERT-P4.

Another approach to checking the correctness of a P4 program is employed by VERA [14]. It operates fully automatically and can detect a wide range of practical bugs. However, it is limited to analysing concrete table snapshots and does not reason over the whole space of possible table entries.

A separate class of tools focuses on verifying the correctness of the translation to concrete network configurations. This area of research is often referred to as *toolstack verification*. These tools have goals different to those of OCTOPUS, which is concerned with parser equivalence rather than end-to-end translation correctness. An example is P4PKTGEN [15], which uses symbolic execution to generate test packets along with their expected processing outcomes. These packets are then put into a software switch (BMV2 [16]), and it is observed whether the actual behaviour aligns with the predictions.

Each of the previous tools relies on ad hoc models of semantics, or utilises an existing implementation, such as P4C[1], the reference compiler of P4. Effort has been made to provide formal semantics for P4, for example, in P4K [17], which specifies these in the K framework [18]. The downside of P4K is that it uses the $P4_{14}$ language specification, which has a significantly different syntax and semantics compared to the modern $P4_{16}$ [19]. A more recent development is PETR4 [20], which defines formal semantics for a subset of $P4_{16}$. Major contributors to the P4 language (specification) have worked on this research. The goal is to extend this approach to the full language and to write a verified compiler for P4. By ensuring correctness upstream in the compiler, the tools that rely on it can also be considered more trustworthy, as a consequence.

Leapfrog From the discussed tools, OCTOPUS is most similar to the functional verifiers. However, as mentioned, we are interested in the relational property of behavioural equivalence between two parsers. All the tools we have seen so far work on a single program instead. In our direction, only a single work has conducted research for P4, namely LEAPFROG [21]. As a result, it is the most relevant.

LEAPFROG, implemented in ROCQ, a proof assistant, is a behavioural equivalence checker of P4 packet parsers. The tool accepts a custom syntax representing a subset of P4 functionality. The semantics of this syntax enable a written program to be modelled as a deterministic finite automaton (DFA). Then, by writing packet parsers in this syntax, equivalence checking becomes a matter of proving the equivalence of two DFAs. By formulating equivalence as a (backwards) bisimulation [22] between these underlying DFAs, and using ROCQ together with an SMT solver (see Section 8) for formal verification, equivalence can be formally proven. While this approach is effective, it requires a substantial amount of time and computational resources. For example, the implementation of a real-world parser required more than 500 GB of memory and had a runtime of nearly a day, until a timeout occurred due to the machine running out of memory. This is the result of generating a proof object on the fly, which also contains a lot of redundant information. What if such a proof is not constructed, but the equivalence is merely checked? This could significantly improve the algorithm's runtime, possibly allowing it to run on regular PC equipment instead of requiring high-memory servers.

3. Research Question

The principal research question (PRQ) addressed in this thesis is as follows.

(PRQ) **What is the impact of not constructing a proof in parallel when checking the behavioural equivalence of P4 packet parsers?**

In our context, impact has two equally valid and practical interpretations, leading to the formulation of two sub-questions (RQ1, RQ2) in support of the PRQ.

(RQ1) What effect does avoiding proof construction have on the runtime of P4 packet parser behavioural equivalence checking?

[1] https://github.com/p4lang/p4c

(RQ2) To what degree can an implementation that does not construct a proof produce a certificate that allows an external program to verify its correctness?

4. Methodology

To investigate a Leapfrog implementation that does not construct a proof in parallel, we must reimplement their approach. This should be done in a general-purpose programming language that does not construct a proof object like Rocq builds during computation. A lot of work with P4 has been done in C++. For example, the reference compiler of P4 programs, P4C, is written in C++. However, instead, we will use Python. The reason for this is twofold:

- Python is widely used and well-supported, with readily available packages on PyPI for the SMT solvers required in our work, such as Z3 [23] and cvc5 [24], as well as a library package that serves as an abstraction over using different solvers (see Section 8). In contrast, the C++ ecosystem lacks a unified source for all required solvers. For instance, vcpkg does not provide a package for cvc5 at the time of writing.

- We do not expect a significant performance loss from using Python instead of C++. A substantial portion of the runtime will be spent within external SMT solvers, which are implemented as efficient C or C++ libraries and accessed via Python bindings. As such, the performance-critical components remain largely unaffected by the choice of host language.

Besides using a different programming language, it is also worth noting that, while our goal is to reimplement the approach proposed by Leapfrog, our approach to bisimulation differs in a key aspect. Rather than using a backwards bisimulation strategy as in the original work, we adopt a forward bisimulation approach. Further details are provided in Section 7.

We need to ensure that we obtain results that can be compared to those obtained by Leapfrog. As a result, our evaluation methodology consists of two separate experiment runs. In the first run, we execute the benchmarks of Leapfrog in an identical manner as its authors did. Then, after we have written our implementation, we run the same benchmarks with our tool. We perform this task on the same machine under the same conditions, using the same utilities. Ultimately, we obtain findings that can be compared. We will use the outcome of the testing methodology to make a statement about the runtime of our implementation, answering **RQ1**.

The correctness of our implementation of Leapfrog will not be proved. Additionally, our algorithm does not construct a proof that verifies the correctness of its findings. Thus, without modifying the algorithm, one is limited to trusting the implementation and its results. A solution would be to extend the algorithm and have it generate a certificate alongside its decision on equality (**RQ2**). This *certificate* could consist of an encoding of a bisimulation when it deems two parser programs to be equivalent, and a counterexample trace otherwise.

Such a certificate can then be used by an external program to verify the correctness of the certificate's accompanying result without having to fully rerun the analysis. A limitation of this solution is that it cannot guarantee that our implementation produces a correct result for every input. However, we do not consider this an issue, as the individual statements regarding the equivalence of parsers can be verified. This is what is essential for practical applications.

After addressing both sub-questions, we can formulate an answer to the main research question (**PRQ**) of this work. In doing so, we assign equal weight to the outcomes of both sub-questions. This reflects the view that, for our tool to be practical, it must be both easy to run and capable of producing verifiable results that can be trusted.

5. Overview

In this and the upcoming sections, we will look at our implementation of Octopus. It is available as an open-source project under the MIT license and can be found on GitHub[2]. The `README` file in the repository provides a more detailed overview of how to use the tool in practice. To aid practical usage, a Docker image has also been pushed to Docker Hub[3]. It is ready-to-use, comes with two SMT solvers installed (Z3 and cvc5), and includes the benchmarks that are part of our testing methodology, facilitating easy verification of our findings.

We divide the description of the tool into three components, each corresponding to a distinct stage in the processing pipeline. These are examined in the following order:

1. The *frontend* (Section 6), which is responsible for obtaining the intermediate representation (IR) of a parser block with its associated types from a P4 program. Afterwards, it parses this IR into a custom object.

2. The *midend* (Section 7), consisting of a DFA implementation, which serves as an abstraction over the custom object produced by the frontend.

3. The *backend* (Section 7, Section 8), ultimately responsible for performing bisimulation, either on top of the DFA abstraction (naive bisimulation, Section 7) or directly over the parser block representations built by the frontend (symbolic bisimulation, Section 8).

6. Frontend

Octopus is to be provided with two file paths: either to P4 programs, or to their IR in JSON format, as generated by P4C, the P4 reference compiler. If provided IR JSONs, it attempts to parse these into Python objects directly. When supplied two P4 programs, two calls are made to the p4c-graphs command-line tool. This CLI application is a backend of P4C which outputs the IR in JSON format that can then be parsed into a Python object on success.

Once Python objects for both P4 programs are obtained, they are parsed into `ParserProgram` objects. These represent solely the parser block within a P4 program with its types annotated. This conversion is performed by a hand-rolled parser, implemented as a class hierarchy. An illustration of the full pipeline from P4 program to `ParserProgram` can be seen in Figure 3.

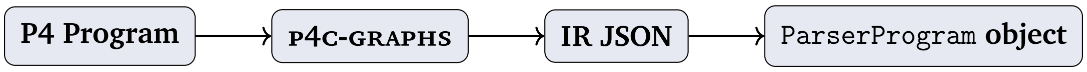


Figure 3: The preprocessing pipeline as used by Octopus. It is run twice, once for each input. Note that entry to the pipeline can occur at either “P4 Program”, or “IR JSON”. Once two `ParserProgram` objects are obtained, we can move on to checking for their equivalence.

P4C has support for the full P4 language, including both $P4_{14}$, as well as $P4_{16}$ [19], the most recent revision of the P4 language. This is not the case for Octopus, which supports only a subset of $P4_{16}$. It should be noted, though, that P4C can be used to translate $P4_{14}$ into $P4_{16}$. So, as long as the $P4_{16}$ syntax to which the $P4_{14}$ source is translated is supported, Octopus could accept such programs as well. Whether a language feature is included or excluded from Octopus's subset comes down to three criteria:

1. Is the feature relevant for the parser block in a P4 program? If not, then it is irrelevant to Octopus and thus ignored.

[2]https://github.com/jortvanleenen/Octopus
[3]hub.docker.com/repository/docker/jortvanleenen/octopus

2. Is the feature translatable into a DFA-centric approach? If not, then it falls outside the scope of this research. An example of this includes `varbit`, a variable-length bitstring type. Its behaviour resembles that of a stack-like data structure, which may require modelling with a pushdown automaton (PDA) rather than a DFA. As a result, features like `varbit` are excluded from this work.

3. Is the feature essential to the implementation? Certain features of P4 serve primarily as syntactic conveniences. For example, the `enum` type can be represented equivalently using fixed-size bit-vectors within the context of a packet parser. Due to time constraints, such non-essential features have been omitted from this work.

Based on the criteria above, we define a formal grammar that captures the syntax accepted by OCTOPUS; see Listing 3. This grammar is written to closely mirror the formal grammar specified in [19], ensuring comparability. Our grammar, and thus accepted syntax, fully covers the syntax and semantics supported by LEAPFROG, making it at least equally expressive. In addition, it extends upon their language by introducing two extra operators: bit-vector AND and bit-vector shift-right. These extensions demonstrate the ease with which OCTOPUS's language subset can be expanded, owing to its modular and hierarchically structured implementation. Further details are provided in Section 9.3.

Features that fall under the categories described by points 2 and 3, namely those that are either non-translatable into a DFA-centric approach or non-essential to the implementation, will trigger a warning from OCTOPUS at runtime. The tool will attempt to recover gracefully, but will raise an exception if recovery fails. This behaviour contrasts with that for features falling under point 1. By default, OCTOPUS will quietly ignore them, as some of which are considered essential for the compilability of the P4 program. Such features are safe to ignore as they are not relevant to the semantics of parser blocks. This distinction reflects the broader observation that parser blocks represent only one component of a complete P4 program.

To support this separation of concerns between the syntax required for the actual parser analysis and the syntax just required for program compilation, we define a minimal template, as shown in Listing 4. This template serves as a lightweight framework within which arbitrary parser blocks can be embedded. It was used for all benchmarks in this work, demonstrating its general applicability. On a similar note, the formal grammar defined earlier (Listing 3) deliberately excludes language constructs unrelated to parser semantics.

More informally, we restrict the $P4_{16}$ language according to the following conventions:

- OCTOPUS expects headers to consist solely of fixed-width bit-vectors. These headers must eventually be grouped into a `struct` that represents the parsed packet upon completion of the parser.

- A parser block must take exactly two parameters: one variable of type `packet_in`, representing the input stream, and one `out`-directed variable of the aforementioned `struct` type.

- Within a parser block, local variables and instantiations are disallowed. Only assignment statements are permitted, possibly using a method call to extract bits from the stream. Each parser state must include at least one extraction; states consisting solely of transitions are not allowed. We also disallow nested blocks and direct application.

- Slice expressions are not permitted on the left-hand side of assignments (*i.e.* as `lvalues`).

- Set types are not supported. Consequently, the mask and range operators are disallowed in select expressions. List expressions, used as syntactic shorthand, are also excluded because they are not essential to our implementation.

- Only a limited subset of operators is supported: slicing, concatenation, bit-vector shift-right, and bit-vector AND.

We adhere to the semantics of P4 as defined in [19], and formalised in [20]. As such, we do not redefine it here.

7. Naive Bisimulation

As illustrated in Figure 3, we obtain two `ParserProgram` objects representing the two parser blocks with their associated types. This section illustrates how one can transition from this implementation to a DFA, which is defined as follows [25]. It is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

1. Q is a finite set called the *states*,
2. Σ is a finite set called the *alphabet*,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the *transition function*,
4. $q_0 \in Q$ is the *initial state*,
5. $F \subseteq Q$, is the set of *accepting states*.

A DFA works as follows. We begin in an initial state and process an input string symbol by symbol from a fixed alphabet. At each step, the current state and the input symbol determine a unique transition to the next state. The machine continues until all input symbols are consumed. If it halts in an accepting state, the input is accepted; otherwise, due to the absence of a valid transition or ending in a non-accepting state, it is rejected.

We will directly base our translation from `ParserProgram` to DFA on P4A as defined in [21]. As a result, most of the formulas that we will describe are (near-)equivalent. However, we will not specify the full semantics as they do. These are already defined for P4 in, *e.g.*, [20] as indicated before. Instead, we will state the essential formulas and notations required to understand the eventual naive bisimulation algorithm and its implementation.

Every packet parser works on an input stream of bits. More formally, when we consider a packet parser as a state machine, its alphabet can be regarded as the set of symbols for which it needs to determine whether to accept or reject, so $\Sigma = \{0, 1\}$ for all P4 packet parsers.

A P4 packet parser always has a finite number of states per the language definition, which we will consider Q, and a finite set of associated types, which we will consider as a set of *header names* H for simplicity's sake.

Each *header* $h \in H$ has an associated size $sz(h) \in \mathbb{N}^+$. When we write $|bv|$ for the length of $bv \in \{0, 1\}^*$, we can define S as the set of finite functions: $s : H \longrightarrow \{0, 1\}^*$, where $|s(h)| = sz(h)$.

In P4, each state $q \in Q$ can be considered to consist of two components: an operation block $(op(q))$ and a transition block $(tz(q))$. The *operation block* is responsible for reading in new bits and updating the *store* S, the structured representation of what has been read. The *transition block* is responsible for determining to which state to transition next. An operation block will *buffer*, ensuring that it will only execute once the total number of bits it will read is available on the input stream/buffer. To know how many bits it will read, we need to consider the summation of all $sz(h)$ for which a `extract`(h) method call exists in $op(q)$. Note that $sz(h)$ is known for all h because we have parsed the associated types with each parser block in a `ParserProgram` object.

A P4 packet parser, represented as a state machine, can be seen in Figure 4. A state named "start" always represents the initial state. It transitions to other states based on what has been read and what is currently in the buffer. A packet parser terminates once the buffer is empty and a terminal state has been reached. In P4, each packet parser has two such states, namely "reject" and "accept".

As in [21], we can now consider a *configuration*, which represents the state of the DFA representation of a `ParserProgram`. A *configuration* is a triple:

$$\langle q, s, w \rangle \in (Q \cup \{\text{accept}, \text{reject}\}) \times S \times \{0, 1\}^*.$$

It can be interpreted as being in a state in Q, with the contents of a store defined by S, and with w in the buffer. We define C as the finite set of configurations, and F as the set of accepting configurations. Building up the buffer can be defined as a bit-by-bit operation, representing our transition function

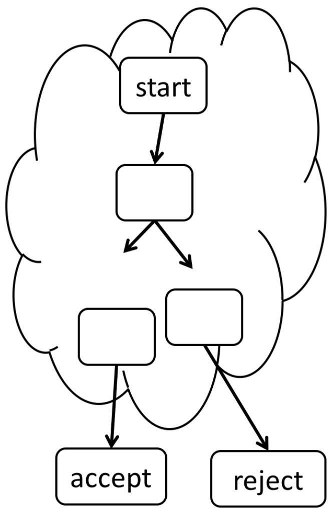


Figure 4: An illustration [19] of a parser defined in P4. Its initial state is always called "start" and is part of the definition of the parser block. It always has precisely two terminal states, "accept", an accepting state, and "reject", a rejecting state. These are implicit, namely, they are not part of the specification as supplied by the programmer.

δ, where we append the buffer by bit b if $|wb| < sz(op(q))$, and else we execute the operation block $op(q)$ and transition according to the transition block $tz(q)$.

As in P4, any further input on the buffer when the parser is in a terminal configuration will always result in a transition to $\langle \text{reject}, s, \varepsilon \rangle$.

Finally, as DFA representation, we can consider the triple: $\langle C, \delta, F \rangle$. Note how the alphabet and the initial state are implicit. That is okay, as these are equivalent for every P4 packet parser.

7.1. Bisimulation

To determine whether two packet parsers are behaviourally equivalent, we construct a bisimulation. Given its central role in this thesis, we illustrate the concept using an accessible analogy inspired by [22].

Consider two vending machines. Each has a payment terminal, a button to request hot chocolate, a button to request coffee, and a dispensing slot. By interacting with the machine, *e.g.*, by pressing buttons and observing the output, we can assess its behaviour. Most importantly, we are not concerned with the machine's internal mechanisms, physical design, or implementation details. What matters is how it behaves from an external perspective: what inputs it accepts and what outputs it produces in response to those inputs.

This mirrors our treatment of packet parsers. Their internal structure is irrelevant for our purposes. Instead, we focus on their observable behaviour, specifically, whether they accept or reject the same input. If two parsers respond identically to all possible inputs, we consider them behaviourally equivalent.

Formally, a bisimulation [22] is defined as being a relation $R \subseteq Q_0 \times Q_1$ for which holds:

1. $q_0 \in F_0 \iff q_1 \in F_1$
2. $\forall_{b \in \{0,1\}}\ \delta_0(q_0, b)\ R\ \delta_1(q_1, b)$

We have seen that we can model each parser as a DFA defined by the tuple $\langle C, \delta, F \rangle$. Here, the set of states C corresponds to the set Q as used in the formal definition of bisimulation. Thus, we can check for bisimularity of `ParserProgram` objects by representing them as DFAs. In practice, each `ParserProgram` is passed to a `DFA` class, which defines DFA operations like those specified above over the semantics of P4. The resulting automata can then be checked for bisimilarity. An overview of this architecture is provided in Figure 5.

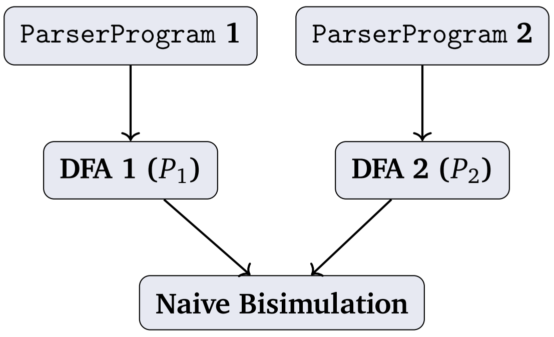


Figure 5: An overview of how naive bisimulation is implemented.

Using pseudocode, we will now describe the operation of the transition block, a component we have so far only briefly touched upon. See Listing 1.

The transition block begins by evaluating the expression used in the select statement. It then iterates over each specified keyset in the order they appear. For each keyset, its value is evaluated and compared against the evaluated key. If the key matches the current keyset, we transition to the respective state. If no match is found, a runtime error is triggered. In our implementation, we model this behaviour by treating it as an implicit transition to the reject state. If a transition block is not explicitly defined for a state, the behaviour defaults to an implicit transition reject.

Listing 1: Pseudocode [19] for the transition block of an arbitrary state in a P4 parser block.

```
key = eval(e);
for (int i=0; i < n; i++) {
    keyset = eval(ks[i]);
    if (keyset.contains(key)) return s[i];
}
verify(false, error.NoMatch);
```

Due to the state space explosion associated with naive bisimulation, we will not be investigating this approach further. However, it was very helpful to verify the correctness of our parsing logic, as its straightforward translation allowed for easy debugging. To get an idea of the performance that can be expected when using naive bisimulation, it was made available as an option for OCTOPUS. Additionally, on the author's computer, 16 bits took around 7 seconds to run, where each bit scales this number by a factor of two. It is clear to see why naive bisimulation is practically impossible for parsers with say more than 24 bits.

8. Symbolic Bisimulation

Just as with naive bisimulation, we also perform a translation for symbolic bisimulation. Instead of using DFAs on top of the parsers, we work with formulas. See Figure 6 for an illustration.

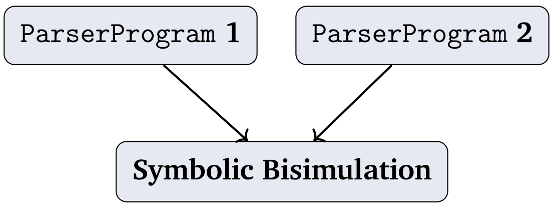


Figure 6: An overview of how symbolic bisimulation is implemented.

The formulas we consider are referred to as template-guarded. They are five-tuples: $(q^<, q^>, n^<, n^>, \varphi)$. The first two elements indicate in which state the left and right parsers are. The third and fourth elements indicate the length of the buffers in both the left and right parsers. Finally, φ represents the current knowledge. It can be considered a statement of what the group of states we symbolically consider has in common.

Key to symbolic bisimulation are the functions for the strongest postcondition (SP) for operations, and symbolic transition (ST) for symbolically representing the transition block of a state.

To understand the strongest postcondition, consider: $\{p\}\ S\ \{q\}$. Here, p represents a precondition, which are assertions that we assume to hold before the execution of program S. q represents the postcondition, which are assertions that hold after S assuming p. It is referred to as a Hoare-triple. This q can vary in 'specificity', where the official term for this is strength. The more 'specific' q is, the stronger it is considered. Consider that p stands for $Cow(x)$ and that S is a no-operation (NOP); in other words, it does not modify the precondition. A valid postcondition q could be $Animal(x)$, as this is implied by $Cow(x)$. However, the strongest postcondition would be $Cow(x)$. Another way to look at this is from the perspective of the Liskov Substitution Principle in object-oriented programming. Cow can be placed at positions where $Animal$ is used, but not the other way around ($Cow \subset animal$, but $Animal \not\subset Cow$).

Similarly to the above illustration for programs S, we can define the strongest postcondition for operations that can occur within the operation block of the packer parser. See Figure 7.

$$\mathrm{SP}(\varphi, \mathrm{prog})$$

$$\mathrm{SP}^{\lessgtr}\{\varphi, \mathrm{extract}(hdr)\} := \exists y_i \in \{0,1\}^{\mathrm{sz}(hdr.field_i)} \quad \varphi\left[\frac{y_i}{\mathrm{st}^{\lessgtr}.hdr.field_i}, \frac{\mathrm{st}^{\lessgtr}.hdr.field_i +\!\!+ \mathrm{buf}^{\lessgtr}}{\mathrm{buf}^{\lessgtr}}\right]$$

$$\mathrm{SP}^{\lessgtr}(\varphi, \mathrm{hdr}.x = e) := \exists y \in \{0,1\}^{\mathrm{sz}(x)} \quad \varphi\left[\frac{y}{\mathrm{st}^{\lessgtr}.x}\right] \wedge \mathrm{st}^{\lessgtr}.x = e$$

$$\mathrm{SP}^{\lessgtr}(\varphi, \mathrm{prog}_1; \mathrm{prog}_2) := \mathrm{SP}^{\lessgtr}(\mathrm{SP}^{\lessgtr}(\varphi, \mathrm{prog}_1), \mathrm{prog}_2)$$

Figure 7: The strongest postcondition of components of the operation block, mathematically defined.

Remember from Section 7 that a pair was in the bisimulation if it agreed on acceptance, and all the reachable pairs from it were in the bisimulation as well. In symbolic bisimulation, we do not explicitly represent pairs of states. Instead, we represent groups of equivalent states using template-guarded formulas. In order to answer questions about these formulas, such as two transitions, left and right, are satisfiable, *i.e.* doable based on our current knowledge φ, we need to ask questions to SMT solvers. In our implementation, we work with a library that abstracts away the individual APIs of SMT solvers. Instead, it defines a general interface through which we can interact with supported solvers. The package is called PySMT. For an overview, see Figure 8.

The symbolic transition function will translate transition blocks into their symbolic equivalent. See Algorithm 1 for pseudocode.

9. Evaluation

Having defined and implemented a tool for checking the equivalence of packet parsers written in P4, we now evaluate the impact of our approach. Specifically, we compare its runtime performance against that of Leapfrog and demonstrate the generation of a certificate in the presence of a counterexample. Together, these evaluations allow us to assess the practical applicability of Octopus.

To enable a direct comparison, we adopt the same benchmark set used in Leapfrog's evaluation. As Leapfrog does not accept P4 syntax, we converted their benchmarks to P4. Each converted benchmark can be found in Appendix C. An overview of the benchmarks and their associated properties is presented in Table 1. Following the methodology outlined in [21], we distinguish between two classes of case

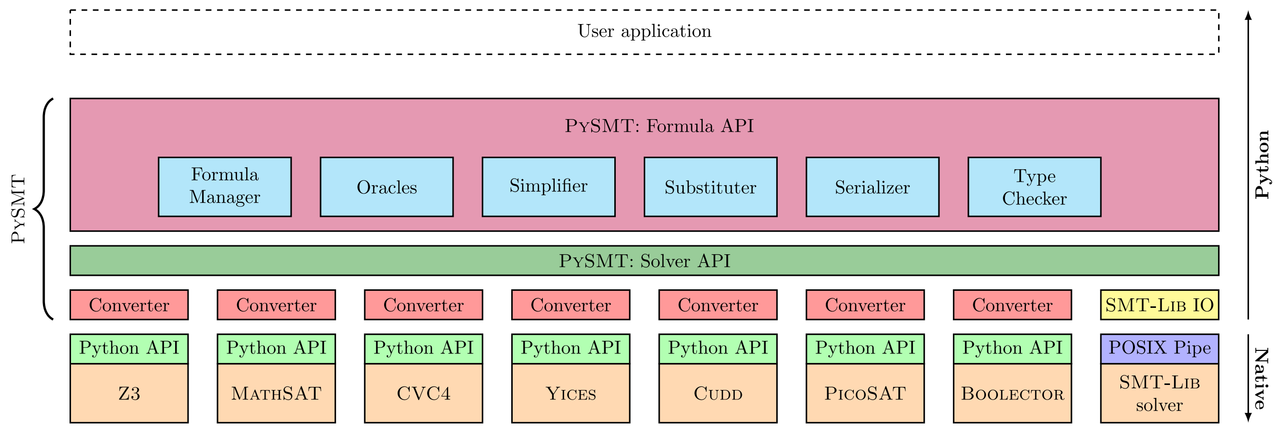


Figure 8: A graphical representation [26] of how PySMT is structured.

studies: utility and applicability. Because detailed definitions for all benchmarks are available in [21], we provide only summaries in the following sections for completeness.

As described in Section 4, we reran the Leapfrog benchmarks on a high-memory machine at Leiden University. With assistance from Dr. Tobias Kappé, we resolved several dependency issues and successfully executed the benchmarks. All series of experiments were conducted on Latinum, a server within the REL Compute lab at the LIACS Research and Education Laboratory (REL). This machine was selected based on the findings reported in [21], which demonstrated the necessity of a high-memory environment, an essential requirement met by Latinum. Namely, it is equipped with $\sim$ 1.5 TB of system memory. All experiments were executed using the respective Docker containers provided by each tool. these were run on an Intel Xeon E5-2630v3. The results can be found in Table 2. All the Octopus benchmark results displayed are averages over four runs, where the first run is ignored to avoid cold start skewing the results.

9.1. Utility

The *utility* class of case studies consists of problems that are designed to evaluate the usefulness of equivalence checking in the networking domain.

State Rearrangement Compilers may merge or split parser states to optimise for hardware constraints. In this case study, we compare two versions of a parser. In one version, fields are extracted in separate states. In the alternate version, we extract both a field and a common prefix simultaneously, thereby merging two states into one. After which, the first field is checked to determine whether additional parsing is needed. This represents a typical example of state merging. Octopus is used to show that the variants accept the same packets, *i.e.* they are behaviourally equivalent.

Variable-Length Formats Many network protocols contain fields whose lengths are not fixed but are instead encoded within the packet itself. In this case study, we examine two scenarios: one in which up to two such fields may appear, and another with up to three. The permissible lengths for each field range from 0 to 6 bytes. We use Octopus to check for equivalence in both cases. Notably, this case study goes beyond what was demonstrated in Leapfrog. Due to performance limitations, Leapfrog was unable to verify equivalence when three variable-length fields were involved. In contrast, the improved runtime performance of Octopus on the two-field case encouraged us to attempt the three-field variant as well, which we eventually successfully verified.

Header Initialisation A common mistake when implementing packet parsers is failing to initialise a header field along all execution paths. This may lead to accessing uninitialised data in certain branches.

Algorithm 1: Conversion of a transition block to symbolic transitions.

```
Input  : cases ((expr, ...) → state), selectors ((expr, ...))
Output : symbolic_transitions {(φ → state), ...}
if #selectors = 0 then
    return {(⊤, cases[DontCare()])}
fi
symbolic_transitions ← ∅
seen ← ∅
foreach (for_exprs, state) ∈ cases do
    formula ← ⊤
    for i ← 0 to #for_exprs do
        if for_exprs[i] is not an instance of DontCare then
            expression ← deepcopy(for_exprs[i])
            expression.to_formula()
            selector ← deepcopy(selectors[i])
            selector.to_formula()
            formula ← formula ∧ (expression = selector)
        fi
    od
    appended_formula ← formula
    foreach φ ∈ seen do
        appended_formula ← appended_formula ∧¬φ
    end
    seen ← seen ∪{formula}
    symbolic_transitions ← symbolic_transitions ∪{(appended_formula, state)}
end
return symbolic_transitions
```

To evaluate whether Octopus can detect such issues, we constructed two versions of a parser. In the first version, a header field is initialised in one state, but not for all possible paths. Using equivalence checking, Octopus can confirm that the program's behaviour does not depend on the uninitialised version of the field. This result is reported in Table 2. In the second version, the initialisation is removed entirely, so even for the execution flow that depends on the field. In this case, Octopus raises an exception, correctly identifying the access to an uninitialised header. This demonstrates the tool's applicability to both scenarios.

Speculative Extraction Another performance optimisation that compilers tend to make is to speculatively extract bits from the buffer in an attempt to effectively vectorise this operation. We have implemented two parsers: one employing speculative extraction and one that does not. We again use Octopus to show their behavioural equivalence.

External Filtering and Relational Verification A parser may be strict or lenient. A *strict* parser explicitly checks for values, whereas a *lenient* parser may test for one value x and, if the test fails, silently assume the alternative y. This is useful for tolerating malformed packets. We consider two parsers, one "sloppy" (lenient) and one strict.

Leapfrog can express *external filters*. These initial relations constrain the final stores and are capable of verifying richer relational properties (*e.g.*, if both parsers accept, their stores must agree on a certain field). Octopus, by contrast, currently only supports behavioural equivalence checking of the parser

Table 1: The benchmark set [21] as used for evaluation. In OCTOPUS, the benchmarks are recognisable by their **Name**, whereas in LEAPFROG they can be recognised by their **File**. **States** represents the total number of states in both parser programs. **B**ranched indicates the considered number of bits in all `transition select` statements. **T**otal reflects the number of bits over all variables. An explicit state space would contain $2^{\mathbf{T}}$ states. An optimal verification algorithm would need to represent $2^{\mathbf{B}}$ states [21].

	Name	**File**[a]	**States**	**B**ranched (b)	**T**otal (b)
Utility	State rearrangement	`IPFilter`[b]	5	8	136
	Variable-length form. 2	`IPOptions2`	30	64	632
	Variable-length form. 3	`IPOptions3`	45	80	672
	Header initialisation	`SelfComparison`	10	10	320
	Speculative extraction	`MPLSVectorized`	5	2	160
	Relational verification	`SloppyStrictStores`	6	64	1056
	External filtering	`SloppyStrictFilter`	6	64	1056
Applicability	Edge	`EdgeSelf`	28	52	3184
	Service provider	`ServiceproviderSelf`	22	50	2536
	Datacenter	`DataCenterSelf`	30	242	2944
	Enterprise	`EnterpriseSelf`	22	176	2144
	Translation validation	`EdgeTrans`	30	56	3148

[a] The full filenames are: `<name-in-table>Proof.v`, except `SloppyStrictStores/Filter`.
[b] The GITHUB repository of LEAPFROG denotes `EthernetProof.v` as the benchmark file of "state rearrangement", this is incorrect.

blocks themselves; it does not support external filters. We therefore use the sloppy–strict pair solely as a sanity check. As is clear from their definition, these parsers are not equivalent. Comparing them using OCTOPUS indeed results in a counterexample trace; see Listing 2 for a truncated example. It clearly shows how, from both start states, a contradiction can be achieved. These traces have also been proven very helpful during the debugging of OCTOPUS.

Listing 2: A truncated counterexample trace obtained by checking the sloppy-strict parser pair from the external filtering and relational verification case studies.

```
--- Counterexample ---
Step 0:
  Left state: start, Right state: start
  Buffer lengths: left=0, right=0

Step 1:
  Left state: parse_ipv4, Right state: reject
  Buffer lengths: left=0, right=0

Step 2:
  Left state: accept, Right state: reject
  Buffer lengths: left=0, right=128
```

9.2. Applicability

Under the applicability class, we consider real-world parsers. This helps us estimate the performance and thus practical applicability of OCTOPUS. To this end, we use the PARSER-GEN framework [27]. As in LEAPFROG, we examine four representative parsers:

- *edge*: modelling an edge router,
- *service provider*: modelling a core router,
- *datacenter*: modelling a top-of-rack switch, and
- *enterprise*: modelling a router for a typical campus or mid-sized enterprise.

As in LEAPFROG, we omit the *big-union* parser, as it serves primarily to estimate hardware resource requirements and is not representative of practical, real-world parser configurations [21].

We based our implementations of these parsers on the representations written by the authors of LEAPFROG. We did have to modify the enterprise parser with respect to the one found in LEAPFROG's GITHUB repository[4]. Namely, an indexing error occurred because the size of the IPv4 header was incorrectly specified. We were able to adjust this to the correct value by looking at the original code at the PARSER-GEN repository[5], where the size was correctly defined (as $4 \times 4 \times 8 = 128$).

We used OCTOPUS to perform a self-equivalence check for each parser, verifying that each is equivalent to itself. This serves both as a sanity check and as a means of assessing the tool's real-world performance. Additionally, the PARSER-GEN framework includes a compiler that produces a hardware-optimised version of a parser. In [21], this was used to compile the edge benchmark. We replicate this experiment (translation validation) by using OCTOPUS to check the plain and compiled versions of the edge parser in P4 for behavioural equivalence.

	LEAPFROG		OCTOPUS (Z3, CVC5)	
Name	**Runtime (m)**	**Memory (GiB)**	**Runtime (m)**	**Memory (GiB)**
State rearrangement	1.13	1.07	0.08	0.04
Variable-length form. 2	3825.70	391.91	1.91	0.33
Variable-length form. 3	–	–	46.22	4.36
Header initialisation	21.56	13.48	0.01	0.04
Speculative extraction	5.45	3.26	0.01	0.04
Relational verification	1.59	1.65	–	–
External filtering	2.37	2.03	–	–
Edge	730.12	243.35	0.19	0.08
Service provider	7167.03	823.70	0.08	0.05
Datacenter	1975.92	382.17	4.66	0.76
Enterprise	817.12	66.35	2.77	0.39
Translation validation	1035.38	349.15	0.38	0.12

Table 2: Runtime and memory usage for each case study under both LEAPFROG and OCTOPUS. Runtime, given in minutes, refers to the total wall-clock time required for the complete execution of the equivalence checking process. Memory usage, reported in GiB, indicates the peak resident memory consumed during this process.

9.3. Extendability

As discussed in Section 6, we extended the syntax as supported by OCTOPUS to include bit-vector AND and bit-vector shift-right operations. This demonstrated the ease with which the accepted language subset can be expanded. To also show how to validate the correctness of these additions easily, we introduced a small benchmark titled *extended syntax*. It builds on the plain parser from the speculative

[4]https://github.com/verified-network-toolchain/leapfrog/blob/main/lib/Benchmarks/Enterprise.v
[5]https://github.com/grg/parser-gen/blob/master/examples/headers-enterprise.txt

extraction case study, but redefines the alternate version. Rather than performing speculative extraction, it modifies the `select` statement.

Specifically, the original version extracts the 9th bit using a slice: `field[8:8]`, while the alternate version uses the extended syntax: `field » 8 & 0x1`. Using Octopus, we confirmed that both parsers are behaviourally equivalent.

9.4. Octopus Variants

In addition to comparing Octopus against Leapfrog, we also conducted benchmarking across different configurations of Octopus itself. Specifically, we evaluated two different solver configurations: one using only Z3, and the other using only cvc5. The results are summarised in Table 3.

As expected, memory usage remained largely consistent across both configurations. This is reasonable, as both solvers should result in Octopus examining the same formulas in its work queue, and, ultimately, in generating a certificate with equivalent content, albeit possibly in a different order. The more notable difference lies in runtime: across all benchmarks, the configuration using cvc5 consistently achieved faster execution. A Wilcoxon signed-rank test, applied to the paired average runtimes with a significance threshold of $p = 0.05$, confirmed that this performance difference is statistically significant. In other words, we can state with high confidence that cvc5 outperforms Z3 for our benchmark suite. It is also noteworthy that both configurations outperform the parallel solver portfolio, probably due to some overhead in parser creation and calling. Nonetheless, given the relatively small number of benchmarks, we retain a parallel solver configuration as the default Octopus portfolio to fence against instances that do not follow this trend.

We also evaluated the impact of disabling the leaps optimisation in Octopus. As shown in Table 3, disabling leaps results in a significant increase in both runtime and memory consumption, even for the smallest benchmarks. This increase can be attributed to the larger state space that must be explored without making leaps, resulting in higher computational overhead and memory requirements. These results underline the importance of leaps in ensuring the practical scalability of Octopus as we have observed.

	Octopus (Z3)		**Octopus** (cvc5)		**Octopus** (no leaps)	
Name	**Run (m)**	**Mem (GiB)**	**Run (m)**	**Mem** (GiB)	**Run (m)**	**Mem** (GiB)
State rearrangement	0.01	0.05	0.01	0.04	0.35	0.24
Variable-length form. 2	2.11	0.33	1.84	0.31	–	–
Variable-length form. 3	42.45	4.36	40.58	4.34	–	–
Header initialisation	0.02	0.05	0.01	0.04	5.64	3.13
Speculative extraction	0.02	0.05	0.01	0.04	0.48	0.30
Relational verification	–	–	–	–	–	–
External filtering	–	–	–	–	–	–
Edge	0.31	0.09	0.17	0.07	–	–
Service provider	0.14	0.07	0.08	0.05	149.18	49.68
Datacenter	5.68	0.77	4.16	0.74	–	–
Enterprise	3.29	0.39	2.54	0.37	–	–
Translation validation	0.54	0.13	0.35	0.11	–	–

Table 3: A tabular overview of the performance of different Octopus configurations. Note that “runtime” and “memory” have been abbreviated to “run” and “mem” respectively. The first column group details the performance of Octopus using only the Z3 SMT solver in its solver portfolio. Likewise, the configuration depicted in the second column group employs just the cvc5 solver. The final configuration in the third, right-most column group uses the default solver portfolio, but without leaps enabled.

10. Conclusion

It has been demonstrated that we were able to implement forward bisimulation and symbolic execution with the goal of equivalence checking of P4 packet parsers, inspired by LEAPFROG. We have done so in a manner that accepts a subset of the P4 syntax and accurately follows its semantics. Furthermore, due to the modular structure of the codebase, it has been shown to be easily extendable to accept a larger subset of $P4_{16}$.

We successfully replicated all findings reported by LEAPFROG, with the exception of those that rely on external filtering, which is not supported by OCTOPUS. Conversely, OCTOPUS was able to verify the variable-length format benchmark with up to three fields, whereas LEAPFROG was limited to two.

By not constructing a proof object on the fly during computation, we have achieved a more practical runtime performance. Each of the equivalence checks over parser blocks that we have considered could have been executed on a modern PC. We therefore conclude that OCTOPUS exhibits practical runtime performance, thereby answering **RQ1**.

Additionally, we have shown that our method of generating certificates is helpful and sufficient for practical usage in both positive and negative cases of behavioural equivalence. Especially the counterexample trace has already proven helpful, both during debugging, as mentioned before, but also during the conversion of LEAPFROG benchmarks to P4, as it allowed us to pinpoint issues that would have been hard to spot otherwise. We therefore consider OCTOPUS sufficiently capable of producing certificates suitable for practical and trusted use, thereby addressing **RQ2**.

In conclusion, obtaining favourable outcomes for both **RQ1** and **RQ2**, we have described and implemented a tool that can be used for the practical equivalence checking of P4 packet parsers, answering **PRQ**.

11. Future Work

While OCTOPUS demonstrates practical runtime performance, there remains potential for further optimisation. Currently, the implementation is fully synchronous and does not leverage multiprogramming. Although we utilise some asynchronous features of PYSMT, such as portfolio solving with solvers running in parallel, the validity and satisfiability queries are invoked synchronously from within our codebase. A promising direction for future improvement would be to parallelise the processing of the work queue maintained during symbolic bisimulation. Given that all pure formula objects should be unique copies, this could be a straightforward modification.

While we have experimented with the Z3 and CVC5 SMT solvers, PYSMT supports a wider range of solvers. Built-in support exists, for example, for BOOLECTOR 3.0 [28], MATHSAT 5 [29], and YICES 2.2 [30], all of which support the bit-vector SMT theory. PYSMT also supports any SMT solver that is SMT-LIB 2 compliant. It could be interesting to explore a different or more extensive combination of SMT solvers for the portfolio that symbolic bisimulation uses. This is especially interesting for solvers for which we do not have directly comparable results available, such as CVC5, which performed well in SMT-COMP 2024. Another strong contender in SMT-COMP 2024 was BITWUZLA [31], which is SMT-LIB 2 compliant and won most subcategories in the QF_BITVEC (Single Query Track) of SMT-COMP 2024.

We chose to support a subset of the $P4_{16}$ language specification, as shown in Listing 3, which matches the supported syntax by LEAPFROG. As we have shown, this subset can be easily expanded due to the structure of the OCTOPUS codebase. In further work, one could expand this subset to support other language constructs. For example, one could allow the use of slices in `lvalues`, which currently have to be emulated by concatenating slices on the right-hand side of the assignment operator. Another example could be implementing more operators, allowing for more expressive expressions to be made.

Finally, as mentioned, we were unable to perform the external filtering and relational verification benchmarks as required for the proper comparison of the sloppy and strict P4 programs. The codebase of OCTOPUS could be expanded to support this behaviour, rather than, for example, at the moment having to parse the certificate and possibly performing external filtering there, before returning true or false for equivalence.

References

[1] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*, 8th (Global Edition). Harlow, United Kingdom: Pearson Education Limited, 2022, ISBN: 978-1-292-40546-9.

[2] T. Barnett, S. Jain, U. Andra, and T. Khurana, *Cisco Annual Internet Report (2018–2023)*, en, 2018. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html (visited on 05/27/2025).

[3] *CVE: Common Vulnerabilities and Exposures*, May 2025. [Online]. Available: https://www.cve.org/ (visited on 06/01/2025).

[4] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, "Security applications of formal language theory," *IEEE Systems Journal*, vol. 7, no. 3, pp. 489–500, 2013, Publisher: IEEE. DOI: 10.1109/JSYST.2012.2222000.

[5] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014, Publisher: Ieee. DOI: 10.1109/JPROC.2014.2371999.

[6] T. Koponen, M. Casado, N. Gude, *et al.*, "Onix: A distributed control platform for large-scale production networks," in *9th USENIX symposium on operating systems design and implementation (OSDI 10)*, 2010. DOI: 10.5555/1924943.1924968.

[7] S. Jain, A. Kumar, S. Mandal, *et al.*, "B4: Experience with a globally-deployed software defined wan," en, *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, Sep. 2013, ISSN: 0146-4833. DOI: 10.1145/2534169.2486019.

[8] N. McKeown, T. Anderson, H. Balakrishnan, *et al.*, "OpenFlow: Enabling innovation in campus networks," en, *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Mar. 2008, ISSN: 0146-4833. DOI: 10.1145/1355734.1355746.

[9] P. Bosshart, D. Daly, G. Gibb, *et al.*, "P4: Programming protocol-independent packet processors," en, *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014, ISSN: 0146-4833. DOI: 10.1145/2656877.2656890.

[10] J. Liu, W. Hallahan, C. Schlesinger, *et al.*, "P4v: Practical verification for programmable data planes," en, in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, Budapest Hungary: ACM, Aug. 2018, pp. 490–503, ISBN: 978-1-4503-5567-4. DOI: 10.1145/3230543.3230582.

[11] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos, "Verification of P4 programs in feasible time using assertions," en, in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, Heraklion Greece: ACM, Dec. 2018, pp. 73–85, ISBN: 978-1-4503-6080-7. DOI: 10.1145/3281411.3281421.

[12] B. Tian, J. Gao, M. Liu, *et al.*, "Aquila: A practically usable verification system for production-scale programmable data planes," en, in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, Virtual Event USA: ACM, Aug. 2021, pp. 17–32, ISBN: 978-1-4503-8383-7. DOI: 10.1145/3452296.3472937.

[13] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," en, *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/360933.360975.

[14] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging P4 programs with vera," en, in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, Budapest Hungary: ACM, Aug. 2018, pp. 518–532, ISBN: 978-1-4503-5567-4. DOI: 10.1145/3230543.3230548.

[15] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "P4pktgen: Automated Test Case Generation for P4 Programs," en, in *Proceedings of the Symposium on SDN Research*, Los Angeles CA USA: ACM, Mar. 2018, pp. 1–7, ISBN: 978-1-4503-5664-0. DOI: 10.1145/3185467.3185497.

[16] H. Stubbe, "P4 compiler & interpreter: A survey," *Future Internet (FI) and Innovative Internet Technologies and Mobile Communication (IITM)*, vol. 47, 2017.

[17] A. Kheradmand and G. Rosu, *P4K: A Formal Semantics of P4 and Applications*, arXiv:1804.01468 [cs], Apr. 2018. DOI: 10.48550/arXiv.1804.01468.

[18] G. Roșu and T. F. Șerbănută, "An overview of the K semantic framework," *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010, Publisher: Elsevier. DOI: 10.1016/j.jlap.2010.03.012.

[19] The P4 Language Consortium, *P4_16 Language Specification - version 1.2.5*, Oct. 2024. [Online]. Available: https://p4.org/wp-content/uploads/2024/10/P4-16-spec-v1.2.5.html (visited on 04/24/2025).

[20] R. Doenges, M. T. Arashloo, S. Bautista, *et al.*, "Petr4: Formal foundations for p4 data planes," en, *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–32, Jan. 2021, ISSN: 2475-1421. DOI: 10.1145/3434322.

[21] R. Doenges, T. Kappé, J. Sarracino, N. Foster, and G. Morrisett, "Leapfrog: Certified equivalence for protocol parsers," en, in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, San Diego CA USA: ACM, Jun. 2022, pp. 950–965, ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523715.

[22] D. Sangiorgi, *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011, ISBN: 978-0-511-77711-0. DOI: 10.1017/CBO9780511777110.

[23] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Hutchison, T. Kanade, J. Kittler, *et al.*, Eds., vol. 4963, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78799-0. DOI: 10.1007/978-3-540-78800-3_24.

[24] H. Barbosa, C. Barrett, M. Brain, *et al.*, "Cvc5: A Versatile and Industrial-Strength SMT Solver," en, in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds., vol. 13243, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2022, pp. 415–442, ISBN: 978-3-030-99524-9. DOI: 10.1007/978-3-030-99524-9_24.

[25] M. Sipser, *Introduction to the Theory of Computation*, en. Cengage Learning, Jun. 2012, ISBN: 978-1-133-18779-0.

[26] M. Gario and A. Micheli, "PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms," in *SMT workshop*, vol. 2015, 2015.

[27] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design principles for packet parsers," in *Architectures for Networking and Communications Systems*, IEEE, 2013, pp. 13–24. DOI: 10.1109/ANCS.2013.6665172.

[28] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , BtorMC and Boolector 3.0," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds., vol. 10981, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2018, pp. 587–595, ISBN: 978-3-319-96144-6. DOI: 10.1007/978-3-319-96145-3_32.

[29] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT5 SMT Solver," en, in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Hutchison, T. Kanade, J. Kittler, *et al.*, Eds., vol. 7795, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 93–107, ISBN: 978-3-642-36741-0. DOI: 10.1007/978-3-642-36742-7_7.

[30] B. Dutertre, “Yices 2.2,” in *Computer Aided Verification*, D. Hutchison, T. Kanade, J. Kittler, *et al.*, Eds., vol. 8559, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2014, pp. 737–744, ISBN: 978-3-319-08866-2. DOI: 10.1007/978-3-319-08867-9_49.

[31] A. Niemetz and M. Preiner, “Bitwuzla,” en, in *Computer Aided Verification*, C. Enea and A. Lal, Eds., vol. 13965, Series Title: Lecture Notes in Computer Science, Cham: Springer Nature Switzerland, 2023, pp. 3–17, ISBN: 978-3-031-37702-0. DOI: 10.1007/978-3-031-37703-7_1.

A. Supported P4 Grammar

Listing 3: The P4 grammar as accepted by OCTOPUS. It represents the subset of the P4 language that is both valid under P4C, as well as parseable by OCTOPUS for the purpose of equivalence checking. Grammar rules required by P4C for full program compilation but irrelevant to parser block definitions are omitted. For further details, see Section 6. The grammar is deliberately written to closely follow the YACC/Bison style used in the official $P4_{16}$ language specification [19], which may lead to some verbose rules. For simplicity, lexical tokens with a fixed string representation, such as `HEADER`, are written as string literals (in violet). Lexical constructs without a fixed string representation appear in full capitals (in blue). Both should be understood as lexer-level constructs handling related syntactical details.

```
/** PROGRAM **/

p4program
    : typeDeclarations parserDeclaration
    ;

/** TYPES **/

typeDeclarations
    : typeDeclaration
    | typeDeclarations typeDeclaration
    ;

typeDeclaration
    : headerTypeDeclaration
    | structTypeDeclaration
    ;

headerTypeDeclaration
    : "header" TYPE "{" typeFieldList "}"
    ;

structTypeDeclaration
    : "struct" TYPE "{" typeFieldList "}"
    ;

typeFieldList
    : typeField
    | typeFieldList typeField
    ;

typeField
    : typeRef IDENTIFIER ";"
    ;

typeRef
    : baseType
    | typeName
    ;

baseType
    : "bit" "<" INTEGER ">"
    ;

typeName
    : prefixedType
    ;

prefixedType
```

```
    : TYPE
    | "." TYPE
    ;

/** PARSER **/

parserDeclaration
    : "parser" IDENTIFIER "(" parameterList ")" "{" parserStates "}"
    ;

parameterList
    : parameter "," parameter
    ;

parameter
    : direction typeName IDENTIFIER
    ;

direction
    : /* empty */
    | "in"
    | "out"
    | "inout"
    ;

parserStates
    : parserState
    | parserStates parserState
    ;

parserState
    : "state" STATE "{" parserStatements transitionStatement "}"
    ;

/** PARSER STATEMENTS **/

parserStatements
    : parserStatement
    | parserStatements parserStatement
    ;

parserStatement
    : assignmentOrMethodCallStatement
    ;

assignmentOrMethodCallStatement
    : lvalue "(" expression ")" ";"
    | lvalue "="  expression ";"
    ;

lvalue
    : prefixedNonTypeName
    | lvalue "." member

prefixedNonTypeName
    : nonTypeName
    | "." nonTypeName
    ;

nonTypeName
    : IDENTIFIER
    ;

```

```
member
    : IDENTIFIER
    ;

/** TRANSITION STATEMENTS **/

transitionStatement
    : "transition" stateExpression
    ;

stateExpression
    : STATE ";"
    | selectExpression
    ;

selectExpression
    : "select" "(" expressionList ")" "{" selectCaseList "}"
    ;

expressionList
    : expression
    | expressionList "," expression
    ;

selectCaseList
    : selectCase
    | selectCaseList selectCase
    ;

selectCase
    : keysetExpression ":" STATE ";"
    ;

keysetExpression
    : simpleKeysetExpression
    | "(" simpleKeysetExpression "," simpleExpressionList ")"
    ;

simpleExpressionList
    : simpleKeysetExpression
    | simpleExpressionList "," simpleKeysetExpression
    ;

simpleKeysetExpression
    : expression
    | "default"
    | "_"
    ;

/** EXPRESSIONS **/

expression
    : INTEGER
    | lvalue
    | expression "[" INTEGER ":" INTEGER "]"
    | expression "++" expression
    | expression ">>" expression
    | expression "&" expression
    | "(" expression ")"
    ;
```

B. OCTOPUS P4 Template

Listing 4: This template defines the minimal structure required for a P4 program to be parseable by OCTOPUS. It imports the standard library, declares a parser block, and instantiates a basic architecture. While all these components are necessary for the program to be compilable, only the parser block and its associated type definitions are relevant to the equivalence checking process. Consequently, all other elements have been abstracted into this template to isolate the parts of interest.

```
#include <core.p4>

// header and struct definitions (TO FILL IN)

parser Parser(packet_in pkt, out headers_t hdr) {

    // parser states (TO FILL IN)

}

parser Parser_t(packet_in pkt, out headers_t hdr);
package Package(Parser_t p);

Package(Parser()) main;
```

C. Benchmarks in P4

This section lists the P4 programs on which OCTOPUS was evaluated. They are directly based on the benchmarks of [21], but have been converted to P4 for use with OCTOPUS. All programs use the template as defined in Listing 4; however, the headers and footers have been omitted for increased readability.

C.1. State Rearrangement

Listing 5: Parser block with combined states

```
header ip_t    { bit<64> data; }
header data_t  { bit<32> data; }

struct headers_t {
    ip_t    ip;
    data_t  pref;
    data_t  suff;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.ip);
        pkt.extract(hdr.pref);
        transition select(hdr.ip.data[23:20]) {
            0: parse_suff;
            1: accept;
        }
    }

    state parse_suff {
        pkt.extract(hdr.suff);
        transition accept;
    }
}
```

Listing 6: Parser block with separate states

```
header ip_t   { bit<64> data; }
header udp_t  { bit<32> data; }
header tcp_t  { bit<64> data; }

struct headers_t {
    ip_t   ip;
    udp_t  udp;
    tcp_t  tcp;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.ip);
        transition select(hdr.ip.data[23:20]) {
            0: parse_tcp;
            1: parse_udp;
        }
    }

    state parse_udp {
        pkt.extract(hdr.udp);
        transition accept;
    }

    state parse_tcp {
        pkt.extract(hdr.tcp);
        transition accept;
    }
}
```

C.2. Variable-Length Formatting

Listing 7: The `IPOptions` parser block. For variable-length formatting 3, we consider the parser block as given. For variable-length formatting 2, we consider state `parse_1` as the start state instead. Both these versions have been shown in a single listing due to their equivalence otherwise.

```
header scratch8_t  { bit<8> data; }
header scratch16_t { bit<16> data; }
header scratch24_t { bit<24> data; }
header scratch32_t { bit<32> data; }
header scratch40_t { bit<40> data; }

header t0_t { bit<8> data; }
header l0_t { bit<8> data; }
header v0_t { bit<48> data; }

header t1_t { bit<8> data; }
header l1_t { bit<8> data; }
header v1_t { bit<48> data; }

header t2_t { bit<8> data; }
header l2_t { bit<8> data; }
header v2_t { bit<48> data; }

struct headers_t {
    t0_t t0;
    l0_t l0;
    v0_t v0;
    t1_t t1;
    l1_t l1;
    v1_t v1;
    t2_t t2;
    l2_t l2;
    v2_t v2;

    scratch8_t scratch8;
    scratch16_t scratch16;
    scratch24_t scratch24;
    scratch32_t scratch32;
    scratch40_t scratch40;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.t0);
        pkt.extract(hdr.l0);
        transition select(hdr.t0.data, hdr.l0.data) {
            (0, 0): accept;
            (1, 0): accept;
            (_, 1): parse_01;
            (_, 2): parse_02;
            (_, 3): parse_03;
            (_, 4): parse_04;
            (_, 5): parse_05;
            (_, 6): parse_06;
            default: reject;
        }
    }

    state parse_01 {
        pkt.extract(hdr.scratch8);
        hdr.v0.data = hdr.scratch8.data ++ hdr.v0.data[47:8];
        transition parse_1;
    }

    state parse_02 {
        pkt.extract(hdr.scratch16);
        hdr.v0.data = hdr.scratch16.data ++ hdr.v0.data[47:16];
        transition parse_1;
    }

    state parse_03 {
        pkt.extract(hdr.scratch24);
        hdr.v0.data = hdr.scratch24.data ++ hdr.v0.data[47:24];
        transition parse_1;
    }

    state parse_04 {
        pkt.extract(hdr.scratch32);
        hdr.v0.data = hdr.scratch32.data ++ hdr.v0.data[47:32];
        transition parse_1;
    }

    state parse_05 {
        pkt.extract(hdr.scratch40);
        hdr.v0.data = hdr.scratch40.data ++ hdr.v0.data[47:40];
        transition parse_1;
    }

    state parse_06 {
        pkt.extract(hdr.v0);
        transition parse_1;
    }

    state parse_1 {
        pkt.extract(hdr.t1);
        pkt.extract(hdr.l1);
        transition select(hdr.t1.data, hdr.l1.data) {
            (0, 0): accept;
            (1, 0): accept;
            (_, 1): parse_11;
            (_, 2): parse_12;
            (_, 3): parse_13;
            (_, 4): parse_14;
            (_, 5): parse_15;
            (_, 6): parse_16;
            default: reject;
        }
    }

    state parse_11 {
        pkt.extract(hdr.scratch8);
        hdr.v1.data = hdr.scratch8.data ++ hdr.v1.data[47:8];
        transition parse_2;
    }

    state parse_12 {
        pkt.extract(hdr.scratch16);
        hdr.v1.data = hdr.scratch16.data ++ hdr.v1.data[47:16];
        transition parse_2;
    }

    state parse_13 {
        pkt.extract(hdr.scratch24);
        hdr.v1.data = hdr.scratch24.data ++ hdr.v1.data[47:24];
        transition parse_2;
    }

    state parse_14 {
        pkt.extract(hdr.scratch32);
        hdr.v1.data = hdr.scratch32.data ++ hdr.v1.data[47:32];
        transition parse_2;
    }

    state parse_15 {
        pkt.extract(hdr.scratch40);
        hdr.v1.data = hdr.scratch40.data ++ hdr.v1.data[47:40];
        transition parse_2;
    }

    state parse_16 {
        pkt.extract(hdr.v1);
        transition parse_2;
    }

    state parse_2 {
        pkt.extract(hdr.t2);
        pkt.extract(hdr.l2);
        transition select(hdr.t2.data, hdr.l2.data) {
            (0, 0): accept;
            (1, 0): accept;
            (_, 1): parse_21;
            (_, 2): parse_22;
            (_, 3): parse_23;
            (_, 4): parse_24;
            (_, 5): parse_25;
            (_, 6): parse_26;
            default: reject;
        }
    }

    state parse_21 {
        pkt.extract(hdr.scratch8);
        hdr.v2.data = hdr.scratch8.data ++ hdr.v2.data[47:8];
        transition accept;
    }

    state parse_22 {
        pkt.extract(hdr.scratch16);
        hdr.v2.data = hdr.scratch16.data ++ hdr.v2.data[47:16];
        transition accept;
    }

    state parse_23 {
        pkt.extract(hdr.scratch24);
        hdr.v2.data = hdr.scratch24.data ++ hdr.v2.data[47:24];
        transition accept;
    }

    state parse_24 {
        pkt.extract(hdr.scratch32);
        hdr.v2.data = hdr.scratch32.data ++ hdr.v2.data[47:32];
        transition accept;
    }

    state parse_25 {
        pkt.extract(hdr.scratch40);
        hdr.v2.data = hdr.scratch40.data ++ hdr.v2.data[47:40];
        transition accept;
    }

    state parse_26 {
        pkt.extract(hdr.v2);
        transition accept;
    }
}
```

Listing 8: The `Timestamp` parser block. For variable-length formatting 3, we consider the parser block as given. For variable-length formatting 2, we consider state `parse_1` as the start state instead. Both these versions have been shown in a single listing due to their equivalence otherwise.

```
header scratch8_t  { bit<8> data; }
header scratch16_t { bit<16> data; }
header scratch24_t { bit<24> data; }
header scratch32_t { bit<32> data; }
header scratch40_t { bit<40> data; }

header t0_t { bit<8> data; }
header l0_t { bit<8> data; }
header v0_t { bit<48> data; }

header t1_t { bit<8> data; }
header l1_t { bit<8> data; }
header v1_t { bit<48> data; }

header t2_t { bit<8> data; }
header l2_t { bit<8> data; }
header v2_t { bit<48> data; }

header pointer_t   { bit<8> data; }
header overflow_t  { bit<4> data; }
header flag_t      { bit<4> data; }
header timestamp_t { bit<32> data; }

struct headers_t {
    scratch8_t scratch8;
    scratch16_t scratch16;
    scratch24_t scratch24;
    scratch32_t scratch32;
    scratch40_t scratch40;

    t0_t t0;
    l0_t l0;
    v0_t v0;
    t1_t t1;
    l1_t l1;
    v1_t v1;
    t2_t t2;
    l2_t l2;
    v2_t v2;

    pointer_t pointer;
    overflow_t overflow;
    flag_t flag;
    timestamp_t timestamp;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.t0);
        pkt.extract(hdr.l0);
        transition select(hdr.t0.data, hdr.l0.data) {
            (0x44, 6): parse_0s;
            (0, 0): accept;
            (1, 0): accept;
            (_, 1): parse_01;
            (_, 2): parse_02;
            (_, 3): parse_03;
            (_, 4): parse_04;
            (_, 5): parse_05;
            (_, 6): parse_06;
            default: reject;
        }
    }

    state parse_0s {
        pkt.extract(hdr.pointer);
        pkt.extract(hdr.overflow);
        pkt.extract(hdr.flag);
        pkt.extract(hdr.timestamp);
        transition parse_1;
    }

    state parse_01 {
        pkt.extract(hdr.scratch8);
        hdr.v0.data = hdr.scratch8.data ++ hdr.v0.data[47:8];
        transition parse_1;
    }

    state parse_02 {
        pkt.extract(hdr.scratch16);
        hdr.v0.data = hdr.scratch16.data ++ hdr.v0.data[47:16];
        transition parse_1;
    }

    state parse_03 {
        pkt.extract(hdr.scratch24);
        hdr.v0.data = hdr.scratch24.data ++ hdr.v0.data[47:24];
        transition parse_1;
    }

    state parse_04 {
        pkt.extract(hdr.scratch32);
        hdr.v0.data = hdr.scratch32.data ++ hdr.v0.data[47:32];
        transition parse_1;
    }

    state parse_05 {
        pkt.extract(hdr.scratch40);
        hdr.v0.data = hdr.scratch40.data ++ hdr.v0.data[47:40];
        transition parse_1;
    }

    state parse_06 {
        pkt.extract(hdr.v0);
        transition parse_1;
    }

    state parse_1 {
        pkt.extract(hdr.t1);
        pkt.extract(hdr.l1);
        transition select(hdr.t1.data, hdr.l1.data) {
            (0x44, 6): parse_1s;
            (0, 0): accept;
            (1, 0): accept;
            (_, 1): parse_11;
            (_, 2): parse_12;
            (_, 3): parse_13;
            (_, 4): parse_14;
            (_, 5): parse_15;
            (_, 6): parse_16;
            default: reject;
        }
    }

    state parse_1s {
        pkt.extract(hdr.pointer);
        pkt.extract(hdr.overflow);
        pkt.extract(hdr.flag);
        pkt.extract(hdr.timestamp);
        transition parse_2;
    }

    state parse_11 {
        pkt.extract(hdr.scratch8);
        hdr.v1.data = hdr.scratch8.data ++ hdr.v1.data[47:8];
        transition parse_2;
    }

    state parse_12 {
        pkt.extract(hdr.scratch16);
        hdr.v1.data = hdr.scratch16.data ++ hdr.v1.data[47:16];
        transition parse_2;
    }

    state parse_13 {
        pkt.extract(hdr.scratch24);
        hdr.v1.data = hdr.scratch24.data ++ hdr.v1.data[47:24];
        transition parse_2;
    }

    state parse_14 {
        pkt.extract(hdr.scratch32);
        hdr.v1.data = hdr.scratch32.data ++ hdr.v1.data[47:32];
        transition parse_2;
    }

    state parse_15 {
        pkt.extract(hdr.scratch40);
        hdr.v1.data = hdr.scratch40.data ++ hdr.v1.data[47:40];
        transition parse_2;
    }

    state parse_16 {
        pkt.extract(hdr.v1);
        transition parse_2;
    }

    state parse_2 {
        pkt.extract(hdr.t2);
        pkt.extract(hdr.l2);
        transition select(hdr.t2.data, hdr.l2.data) {
            (0x44, 6): parse_2s;
            (0, 0): accept;
            (1, 0): accept;
            (_, 1): parse_21;
            (_, 2): parse_22;
            (_, 3): parse_23;
            (_, 4): parse_24;
            (_, 5): parse_25;
            (_, 6): parse_26;
            default: reject;
        }
    }

    state parse_2s {
        pkt.extract(hdr.pointer);
        pkt.extract(hdr.overflow);
        pkt.extract(hdr.flag);
        pkt.extract(hdr.timestamp);
        transition accept;
    }

    state parse_21 {
        pkt.extract(hdr.scratch8);
        hdr.v2.data = hdr.scratch8.data ++ hdr.v2.data[47:8];
        transition accept;
    }

    state parse_22 {
        pkt.extract(hdr.scratch16);
        hdr.v2.data = hdr.scratch16.data ++ hdr.v2.data[47:16];
        transition accept;
    }

    state parse_23 {
        pkt.extract(hdr.scratch24);
        hdr.v2.data = hdr.scratch24.data ++ hdr.v2.data[47:24];
        transition accept;
    }

    state parse_24 {
        pkt.extract(hdr.scratch32);
        hdr.v2.data = hdr.scratch32.data ++ hdr.v2.data[47:32];
        transition accept;
    }

    state parse_25 {
        pkt.extract(hdr.scratch40);
        hdr.v2.data = hdr.scratch40.data ++ hdr.v2.data[47:40];
        transition accept;
    }

    state parse_26 {
        pkt.extract(hdr.v2);
        transition accept;
    }
}
```

C.3. Header Initialisation

Listing 9: The parser block where header initialisation occurs in a single state (which is correct).

```
header eth_t  { bit<112> data; }
header vlan_t { bit<32> data; }
header ip_t   { bit<160> data; }
header udp_t  { bit<64> data; }

struct headers_t {
    eth_t   eth;
    vlan_t  vlan;
    ip_t    ip;
    udp_t   udp;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.eth);
        transition select(hdr.eth.data[111:111]) {
            0: default_vlan;
            1: parse_vlan;
        }
    }

    state default_vlan {
        hdr.vlan.data = 0;
        pkt.extract(hdr.ip);
        transition parse_udp;
    }

    state parse_vlan {
        pkt.extract(hdr.vlan);
        transition parse_ip;
    }

    state parse_ip {
        pkt.extract(hdr.ip);
        transition parse_udp;
    }

    state parse_udp {
        pkt.extract(hdr.udp);
        transition select(hdr.vlan.data[31:28]) {
            0b1111: reject;
            default: accept;
        }
    }
}
```

Listing 10: The parser block where header initialisation occurs nowhere (which is incorrect).

```
header eth_t  { bit<112> data; }
header vlan_t { bit<32> data; }
header ip_t   { bit<160> data; }
header udp_t  { bit<64> data; }

struct headers_t {
    eth_t  eth;
    vlan_t vlan;
    ip_t   ip;
    udp_t  udp;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.eth);
        transition select(hdr.eth.data[111:111]) {
            0: default_vlan;
            1: parse_vlan;
        }
    }

    state default_vlan {
        // BUG: vlan not initialised
        pkt.extract(hdr.ip);
        transition parse_udp;
    }

    state parse_vlan {
        pkt.extract(hdr.vlan);
        transition parse_ip;
    }

    state parse_ip {
        pkt.extract(hdr.ip);
        transition parse_udp;
    }

    state parse_udp {
        pkt.extract(hdr.udp);
        transition select(hdr.vlan.data[31:28]) {
            0b1111: reject;
            default: accept;
        }
    }
}
```

C.4. Speculative Extraction

Listing 11: The plain version of the parser block

```
header mpls_t  { bit<32> label; }
header udp_t   { bit<64> data; }

struct headers_t {
    mpls_t  mpls;
    udp_t   udp;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.mpls);
        transition select(hdr.mpls.label[8:8]) {
            0: start;
            1: parse_udp;
        }
    }

    state parse_udp {
        pkt.extract(hdr.udp);
        transition accept;
    }
}
```

Listing 12: The vectorised version of the parser block

```
header udp_t  { bit<64> data; }
header tmp_t  { bit<32> field; }

struct headers_t {
    udp_t  udp;
    tmp_t  tmp;
    tmp_t  old;
    tmp_t  new;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.old);
        pkt.extract(hdr.new);
        transition select(hdr.old.field[8:8], hdr.new.field[8:8]) {
            (0, 0): start;
            (0, 1): parse_udp;
            (1, _): cleanup;
        }
    }

    state parse_udp {
        pkt.extract(hdr.udp);
        transition accept;
    }

    state cleanup {
        pkt.extract(hdr.tmp);
        hdr.udp.data = hdr.new.field ++ hdr.tmp.field;
        transition accept;
    }
}
```

C.5. Relational Verification and External Filtering

Listing 13: The sloppy version of the parser block

```
header eth_t   { bit<112> data; }
header ipv4_t  { bit<128> data; }
header ipv6_t  { bit<288> data; }

struct headers_t {
    eth_t   eth;
    ipv4_t  ipv4;
    ipv6_t  ipv6;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.eth);
        transition select(hdr.eth.data[15:0]) {
            0x86dd: parse_ipv6;
            default: parse_ipv4;
        }
    }

    state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        transition accept;
    }

    state parse_ipv6 {
        pkt.extract(hdr.ipv6);
        transition accept;
    }
}
```

Listing 14: The strict version of the parser block

```
header eth_t  { bit<112> data; }
header ipv4_t { bit<128> data; }
header ipv6_t { bit<288> data; }

struct headers_t {
    eth_t   eth;
    ipv4_t  ipv4;
    ipv6_t  ipv6;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.eth);
        transition select(hdr.eth.data[15:0]) {
            0x86dd: parse_ipv6;
            0x8600: parse_ipv4;
            default: reject;
        }
    }

    state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        transition accept;
    }

    state parse_ipv6 {
        pkt.extract(hdr.ipv6);
        transition accept;
    }
}
```

C.6. Edge and Translation Validation

Listing 15: The plain version of the parser block

```
header eth_t     { bit<112> data; }
header mpls_t    { bit<32> data; }
header eompls_t  { bit<28> data; }
header ipver_t   { bit<4> data; }
header ipv4_5_t  { bit<152> data; }
header ipv4_6_t  { bit<184> data; }
header ipv4_7_t  { bit<216> data; }
header ipv4_8_t  { bit<248> data; }
header ipv6_t    { bit<316> data; }

struct headers_t {
    eth_t      eth0;
    eth_t      eth1;
    mpls_t     mpls0;
    mpls_t     mpls1;
    eompls_t   eompls;
    ipver_t    ipver;
    ipv4_5_t   ipv4_5;
    ipv4_6_t   ipv4_6;
    ipv4_7_t   ipv4_7;
    ipv4_8_t   ipv4_8;
    ipv6_t     ipv6;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.eth0);
        transition select(hdr.eth0.data[15:0]) {
            0x8847: parse_mpls0;
            0x8848: parse_mpls0;
            0x0800: ignore_ipver4;
            0x86dd: ignore_ipver6;
            default: accept;
        }
    }

    state parse_mpls0 {
        pkt.extract(hdr.mpls0);
        transition select(hdr.mpls0.data[8:8]) {
            0: parse_mpls1;
            1: parse_ipver;
            default: reject;
        }
    }

    state parse_mpls1 {
        pkt.extract(hdr.mpls1);
        transition select(hdr.mpls1.data[8:8]) {
            1: parse_ipver;
            default: reject;
        }
    }

    state parse_ipver {
        pkt.extract(hdr.ipver);
        transition select(hdr.ipver.data) {
            0: parse_eompls;
            4: parse_ipv4;
            6: parse_ipv6;
            default: reject;
        }
    }

    state ignore_ipver4 {
        pkt.extract(hdr.ipver);
        transition parse_ipv4;
    }

    state ignore_ipver6 {
        pkt.extract(hdr.ipver);
        transition parse_ipv6;
    }

    state parse_eompls {
        pkt.extract(hdr.eompls);
        transition parse_eth1;
    }

    state parse_eth1 {
        pkt.extract(hdr.eth1);
        transition accept;
    }

    state parse_ipv4 {
        pkt.extract(hdr.ipver);
        transition select(hdr.ipver.data) {
            5: parse_ipv4_5;
            6: parse_ipv4_6;
            7: parse_ipv4_7;
            8: parse_ipv4_8;
            default: reject;
        }
    }

    state parse_ipv4_5 {
        pkt.extract(hdr.ipv4_5);
        transition accept;
    }

    state parse_ipv4_6 {
        pkt.extract(hdr.ipv4_6);
        transition accept;
    }

    state parse_ipv4_7 {
        pkt.extract(hdr.ipv4_7);
        transition accept;
    }

    state parse_ipv4_8 {
        pkt.extract(hdr.ipv4_8);
        transition accept;
    }

    state parse_ipv6 {
        pkt.extract(hdr.ipv6);
        transition accept;
    }
}
```

Listing 16: This is the optimised version of the parser block. Together with its plain version, it is used in the translation validation benchmark.

```
header buf_16_t   { bit<16> data; }
header buf_32_t   { bit<32> data; }
header buf_64_t   { bit<64> data; }
header buf_112_t  { bit<112> data; }
header buf_128_t  { bit<128> data; }
header buf_144_t  { bit<144> data; }
header buf_176_t  { bit<176> data; }
header buf_208_t  { bit<208> data; }
header buf_240_t  { bit<240> data; }
header buf_304_t  { bit<304> data; }
header buf_320_t  { bit<320> data; }

struct headers_t {
    buf_16_t   b16;
    buf_32_t   b32;
    buf_64_t   b64;
    buf_112_t  b112;
    buf_128_t  b128;
    buf_144_t  b144;
    buf_176_t  b176;
    buf_208_t  b208;
    buf_240_t  b240;
    buf_304_t  b304;
    buf_320_t  b320;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.b112);
        transition select(hdr.b112.data[15:0]) {
            0x0800: state_3;
            0x86dd: state_0_suff_1;
            0x8847: state_0_suff_2;
            0x8848: state_0_suff_3;
            default: accept;
        }
    }

    state state_0_suff_1 {
        pkt.extract(hdr.b320);
        transition accept;
    }

    state state_0_suff_2 {
        pkt.extract(hdr.b16);
        transition state_4;
    }

    state state_0_suff_3 {
        pkt.extract(hdr.b16);
        transition state_4;
    }

    state state_1 {
        pkt.extract(hdr.b16);
        transition state_1_suff_0;
    }

    state state_1_suff_0 {
        pkt.extract(hdr.b128);
        transition accept;
    }

    state state_2 {
        pkt.extract(hdr.b16);
        transition state_2_suff_0;
    }

    state state_2_suff_0 {
        pkt.extract(hdr.b304);
        transition accept;
    }

    state state_3 {
        pkt.extract(hdr.b16);
        transition select(hdr.b16.data[11:8]) {
            5: state_3_suff_0;
            6: state_3_suff_1;
            7: state_3_suff_2;
            8: state_3_suff_3;
            default: reject;
        }
    }

    state state_3_suff_0 {
        pkt.extract(hdr.b144);
        transition accept;
    }

    state state_3_suff_1 {
        pkt.extract(hdr.b176);
        transition accept;
    }

    state state_3_suff_2 {
        pkt.extract(hdr.b208);
        transition accept;
    }

    state state_3_suff_3 {
        pkt.extract(hdr.b240);
        transition accept;
    }

    state state_4 {
        pkt.extract(hdr.b16);
        transition select(hdr.b16.data[8:8]) {
            0: state_4_skip;
            1: state_4_trailer;
            default: reject;
        }
    }

    state state_4_skip {
        pkt.extract(hdr.b32);
        transition select(hdr.b32.data[8:8]) {
            1: state_4_trailer;
            default: reject;
        }
    }

    state state_4_trailer {
        pkt.extract(hdr.b16);
        transition select(hdr.b16.data[15:12], hdr.b16.data[11:8]) {
            (0, _):     state_1_suff_0;
            (6, _):     state_2_suff_0;
            (4, 5):    state_3_suff_0;
            (4, 6):    state_3_suff_1;
            (4, 7):    state_3_suff_2;
            (4, 8):    state_3_suff_3;
            default:       reject;
        }
    }
}
```

C.7. Service Provider

```
header eth_t     { bit<112> data; }
header mpls_t    { bit<32> data; }
header ip_ver_t  { bit<4> data; }
header ihl_t     { bit<4> data; }
header ipv4_5_t  { bit<152> data; }
header ipv4_6_t  { bit<184> data; }
header ipv4_7_t  { bit<216> data; }
header ipv4_8_t  { bit<248> data; }
header ipv6_t    { bit<316> data; }

struct headers_t {
    eth_t     eth;
    mpls_t    mpls;
    ip_ver_t  ip_ver;
    ihl_t     ihl;
    ipv4_5_t  ipv4_5;
    ipv4_6_t  ipv4_6;
    ipv4_7_t  ipv4_7;
    ipv4_8_t  ipv4_8;
    ipv6_t    ipv6;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.eth);
        transition select(hdr.eth.data[111:96]) {
            0x8847: parse_mpls;
            0x8848: parse_mpls;
            0x0800: parse_ethv4;
            0x86dd: parse_ethv6;
            default: reject;
        }
    }

    state parse_ethv4 {
        pkt.extract(hdr.ip_ver);
        transition parse_ipv4;
    }

    state parse_ethv6 {
        pkt.extract(hdr.ip_ver);
        transition parse_ipv6;
    }

    state parse_mpls {
        pkt.extract(hdr.mpls);
        transition select(hdr.mpls.data[24:24]) {
            0: parse_mpls;
            1: parse_ip_ver;
            default: reject;
        }
    }

    state parse_ip_ver {
        pkt.extract(hdr.ip_ver);
        transition select(hdr.ip_ver.data) {
            4: parse_ipv4;
            6: parse_ipv6;
            default: reject;
        }
    }

    state parse_ipv4 {
        pkt.extract(hdr.ihl);
        transition select(hdr.ihl.data) {
            5: parse_ipv4_5;
            6: parse_ipv4_6;
            7: parse_ipv4_7;
            8: parse_ipv4_8;
            default: reject;
        }
    }

    state parse_ipv4_5 {
        pkt.extract(hdr.ipv4_5);
        transition accept;
    }

    state parse_ipv4_6 {
        pkt.extract(hdr.ipv4_6);
        transition accept;
    }

    state parse_ipv4_7 {
        pkt.extract(hdr.ipv4_7);
        transition accept;
    }

    state parse_ipv4_8 {
        pkt.extract(hdr.ipv4_8);
        transition accept;
    }

    state parse_ipv6 {
        pkt.extract(hdr.ipv6);
        transition accept;
    }
}
```

C.8. Datacenter

```
header eth_t     { bit<112> data; }
header vlan_t    { bit<160> data; }
header ipv4_t    { bit<160> data; }
header icmp_t    { bit<32> data; }
header tcp_t     { bit<160> data; }
header udp_t     { bit<160> data; }
header gre_t     { bit<32> data; }
header nvgre_t   { bit<32> data; }
header vxlan_t   { bit<64> data; }
header arp_t     { bit<64> data; }
header arp_ip_t  { bit<160> data; }

```

```
struct headers_t {
    eth_t     eth0;
    eth_t     eth1;
    vlan_t    vlan0;
    vlan_t    vlan1;
    ipv4_t    ipv4;
    icmp_t    icmp;
    tcp_t     tcp;
    udp_t     udp;
    gre_t     gre0;
    gre_t     gre1;
    gre_t     gre2;
    nvgre_t   nvgre;
    vxlan_t   vxlan;
    arp_t     arp;
    arp_ip_t  arp_ip;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.eth0);
        transition select(hdr.eth0.data[15:0]) {
            0x8100: parse_vlan0;
            0x9100: parse_vlan0;
            0x9200: parse_vlan0;
            0x9300: parse_vlan0;
            0x0800: parse_ipv4;
            0x0806: parse_arp;
            0x8035: parse_arp;
            default: reject;
        }
    }

    state parse_vlan0 {
        pkt.extract(hdr.vlan0);
        transition select(hdr.vlan0.data[15:0]) {
            0x8100: parse_vlan1;
            0x9100: parse_vlan1;
            0x9200: parse_vlan1;
            0x9300: parse_vlan1;
            0x0800: parse_ipv4;
            0x0806: parse_arp;
            0x8035: parse_arp;
            default: reject;
        }
    }

    state parse_vlan1 {
        pkt.extract(hdr.vlan1);
        transition select(hdr.vlan1.data[15:0]) {
            0x0800: parse_ipv4;
            0x0806: parse_arp;
            0x8035: parse_arp;
            default: reject;
        }
    }

    state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        transition select(hdr.ipv4.data[87:80]) {
            6: parse_tcp;
            17: parse_udp;
            47: parse_gre0;
            default: accept;
        }
    }

    state parse_tcp {
        pkt.extract(hdr.tcp);
        transition accept;
    }

    state parse_udp {
        pkt.extract(hdr.udp);
        transition select(hdr.udp.data[143:128]) {
            0xFFFF: parse_vxlan;
            default: accept;
        }
    }

    state parse_icmp {
        pkt.extract(hdr.icmp);
        transition accept;
    }

    state parse_gre0 {
        pkt.extract(hdr.gre0);
        transition select(hdr.gre0.data[29:29], hdr.gre0.data[15:0]) {
            (1, 0x6558): parse_nvgre;
            (1, 0x6559): parse_gre1;
            default: accept;
        }
    }

    state parse_gre1 {
        pkt.extract(hdr.gre1);
        transition select(hdr.gre1.data[15:0]) {
            0x16558: parse_nvgre;
            0x16559: parse_gre2;
            default: accept;
        }
    }

    state parse_gre2 {
        pkt.extract(hdr.gre2);
        transition select(hdr.gre2.data[15:0]) {
            0x16558: parse_nvgre;
            0x16559: reject;
            default: accept;
        }
    }

    state parse_nvgre {
        pkt.extract(hdr.nvgre);
        transition parse_eth1;
    }

    state parse_vxlan {
        pkt.extract(hdr.vxlan);
        transition parse_eth1;
    }

    state parse_eth1 {
        pkt.extract(hdr.eth1);
        transition accept;
    }

    state parse_arp {
        pkt.extract(hdr.arp);
        transition select(hdr.arp.data[47:32]) {
            0x0800: parse_arp_ip;
            default: accept;
        }
    }

    state parse_arp_ip {
        pkt.extract(hdr.arp_ip);
        transition accept;
    }
}
```

C.9. Enterprise

```
header eth_t      { bit<112> data; }
header vlan_t     { bit<160> data; }
header ipv4_t     { bit<128>  data; }
header ipv6_t     { bit<64>  data; }
header tcp_t      { bit<160> data; }
header udp_t      { bit<160> data; }
header icmp_t     { bit<32>  data; }
header icmp_v6_t  { bit<32>  data; }
header arp_t      { bit<64>  data; }
header arp_ip_t   { bit<64>  data; }

struct headers_t {
    eth_t      eth;
    vlan_t     vlan0;
    vlan_t     vlan1;
    ipv4_t     ipv4;
    ipv6_t     ipv6;
    tcp_t      tcp;
    udp_t      udp;
    icmp_t     icmp;
    icmp_v6_t  icmp_v6;
    arp_t      arp;
    arp_ip_t   arp_ip;
}

parser Parser(packet_in pkt, out headers_t hdr) {
    state start {
        pkt.extract(hdr.eth);
        transition select(hdr.eth.data[111:96]) {
            0x8100: parse_vlan0;
            0x9100: parse_vlan0;
            0x9200: parse_vlan0;
            0x9300: parse_vlan0;
            0x0800: parse_ipv4;
            0x86dd: parse_ipv6;
            0x0806: parse_arp;
            0x8035: parse_arp;
            default: reject;
        }
    }

  state parse_vlan0 {
    pkt.extract(hdr.vlan0);
    transition select(hdr.vlan0.data[159:144]) {
      0x8100: parse_vlan1;
      0x9100: parse_vlan1;
      0x9200: parse_vlan1;
      0x9300: parse_vlan1;
      0x0800: parse_ipv4;
      0x86dd: parse_ipv6;
      0x0806: parse_arp;
      0x8035: parse_arp;
      default: reject;
    }
  }

  state parse_vlan1 {
    pkt.extract(hdr.vlan1);
    transition select(hdr.vlan1.data[159:144]) {
      0x0800: parse_ipv4;
      0x86dd: parse_ipv6;
      0x0806: parse_arp;
      0x8035: parse_arp;
      default: reject;
    }
  }

  state parse_ipv4 {
    pkt.extract(hdr.ipv4);
    transition select(hdr.ipv4.data[79:72]) {
      1: parse_icmp;
      6: parse_tcp;
      11: parse_udp;
      default: accept;
    }
  }

    state parse_ipv6 {
        pkt.extract(hdr.ipv6);
        transition select(hdr.ipv6.data[55:48]) {
            1: parse_icmp_v6;
            6: parse_tcp;
            11: parse_udp;
            default: accept;
        }
    }

    state parse_tcp {
        pkt.extract(hdr.tcp);
        transition accept;
    }

    state parse_udp {
        pkt.extract(hdr.udp);
        transition accept;
    }

    state parse_icmp {
        pkt.extract(hdr.icmp);
        transition accept;
    }

    state parse_icmp_v6 {
        pkt.extract(hdr.icmp_v6);
        transition accept;
    }

    state parse_arp {
        pkt.extract(hdr.arp);
        transition select(hdr.arp.data[31:16]) {
            0x0800: parse_arp_ip;
            default: accept;
        }
    }

    state parse_arp_ip {
        pkt.extract(hdr.arp_ip);
        transition accept;
    }
}
```