Leapfrog: Certified Equivalence for Protocol Parsers

Ryan Doenges
Cornell University
USA
rhd89@cornell.edu

Tobias Kappé
ILLC, University of Amsterdam
Netherlands
t.kappe@uva.nl

John Sarracino
Cornell University
USA
jsarracino@cornell.edu

Nate Foster
Cornell University
USA
jnfoster@cs.cornell.edu

Greg Morrisett
Cornell University
USA
jgm19@cornell.edu

Abstract

We present Leapfrog, a Coq-based framework for verifying equivalence of network protocol parsers. Our approach is based on an automata model of P4 parsers, and an algorithm for symbolically computing a compact representation of a bisimulation, using "leaps." Proofs are powered by a certified compilation chain from first-order entailments to low-level bitvector verification conditions, which are discharged using off-the-shelf SMT solvers. As a result, parser equivalence proofs in Leapfrog are fully automatic and push-button.

We mechanically prove the core metatheory that underpins our approach, including the key transformations and several optimizations. We evaluate Leapfrog on a range of practical case studies, all of which require minimal configuration and no manual proof. Our largest case study uses Leapfrog to perform translation validation for a third-party compiler from automata to hardware pipelines. Overall, Leapfrog represents a step towards a world where all parsers for critical network infrastructure are verified. It also suggests directions for follow-on efforts, such as verifying relational properties involving security.

CCS Concepts: • **Theory of computation → Automata extensions**; • **Software and its engineering → Software verification**.

Keywords: P4, network protocol parsers, Coq, automata, equivalence, foundational verification, certified parsers

ACM Reference Format:
Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. 2022. Leapfrog: Certified Equivalence for Protocol Parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3519939.3523715

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06.
https://doi.org/10.1145/3519939.3523715

1 Introduction

Devices like routers, firewalls and network interface cards as well as operating system kernels occupy a critical role in modern communications infrastructure. Each of these implements parsing for a cornucopia of networking protocols in its *protocol parser*. The parser is the network's first line of defense, responsible for organizing and filtering unstructured and often untrusted data as it arrives from the outside world. Due to their crucial role, bugs in parsers are a significant source of crashes, vulnerabilities, and other faults [48].

Example Router Bug. Consider the following bug, which was present in a commercial router developed by a leading equipment vendor several years ago. Internally, the router was organized around a high-throughput pipeline, which most packets traversed in a single pass. However some packets had to be *recirculated*, meaning they took additional passes through the pipeline before being sent back out on the wire. The router used an internal state variable to decide whether a packet should be recirculated. Usually this state variable was initialized by vendor-supplied code. But, as was discovered by a customer, it could also be erroneously initialized from data in non-standard, malformed packets. Hence, crafted packets could bypass the vendor-supplied initialization code, resulting in an infinite recirculation loop—a denial-of-service (DoS) attack on the router and its peers. In the presence of broadcast traffic, such a "packet storm" would monopolize the router's resources, rendering it unusable until it was rebooted.

An easy way to avoid this bug would be to modify the router's parser to filter away malformed packets, while still accepting valid packets. However, to have full confidence in the new parser, one would need to prove that it is *equivalent* to the original, modulo malformed packets. Although parsers tend to be simple, this would likely be a challenging verification task—it requires reasoning about a *relational* property across two distinct programs.

Parser Equivalence Checking. This paper studies relational verification of protocol parsers, focusing specifically on equivalence of parsers expressed in terms of state machines. Semantic equivalence [15] is a fundamental problem that underpins a wide range of practical verification tasks including translation-validation [42], superoptimization [39], and program synthesis [29]. As we will see in Section 7, the algorithm that we develop for computing equivalence can also be straightforwardly extended to other relational verification challenges, including one inspired by the router bug above (c.f. our external filtering case study in Section 7.1).

There are several technical challenges related to mechanically and formally proving protocol parser equivalence. First, we need a computational model for parsers that is expressive enough to handle practical parsers, but also sufficiently restricted to enable tractable formal verification. Second, we need efficient reasoning techniques based on symbolic representations and domain-specific insights to handle the enormous state space of real-world parsers. Third, we need effective automation and tool support so programmers can avoid manually crafting sprawling proofs of equivalence.

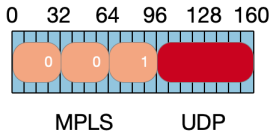
Certified Tooling. The foundational guarantees offered by proof assistants are highly desirable in error-prone domains. However, achieving these guarantees is notoriously hard, as proof assistants need an experienced engineer's guidance to prove all but the simplest goals. One way to scale verification is to break systems into smaller components that compose along shared specifications [2]. This allows individual verification tasks to be solved in isolation, without compromising top-level guarantees. Our hope is that push-button verifiers can reduce total proof burden by certifying some components automatically.

Consider the task of proving that a realistic parser meets a functional specification, in the style of VST [3] proofs which relate C programs to functional specifications. The parser might be hand-optimized for performance reasons like the vectorized parser in Figure 1, which makes it difficult to reason about directly. With an automated equivalence checker, we could justify replacing it with the simpler and easier to verify reference implementation. Other problems like translation validation [36, 53] or proof-producing synthesis [55] would similarly benefit from certified tooling.

Our Contribution: Leapfrog. We present Leapfrog,[1] a new framework that addresses these challenges. It provides an expressive automata model for parsers, with syntax inspired by P4 [10, 16], a networking DSL. The model captures common programming idioms and offers a domain-specific interface for packet parsing. We demonstrate its applicability by encoding parsers for real-world protocols like IPv4 and MPLS.

[1] https://github.com/verified-network-toolchain/leapfrog

```
q1 {
  extract(mpls, 32);
  select(mpls[23:23]) {
    0 ⇒ q1
    1 ⇒ q2
  }
}
q2 {
  extract(udp, 64);
  goto accept
}
```




```
q3 {
  extract(old, 32);
  extract(new, 32);
  select(old[23:23],
         new[23:23]) {
    (0, 0) ⇒ q3
    (0, 1) ⇒ q4
    (1, _) ⇒ q5
  }
}
q4 {
  extract(udp, 64);
  goto accept
}
q5 {
  extract(tmp, 32);
  udp := new ++ tmp;
  goto accept
}
```

Figure 1. Reference (q1, q2) and vectorized (q3, q4, q5) parsers for MPLS and UDP headers (depicted inset).

To establish the equivalence of Leapfrog parsers, we extend classical techniques based on bisimulations to work with symbolic representations of the state space. We also develop a novel up-to technique based on "leaps" that dramatically reduces the cardinality of the constructed relation.

We implement Leapfrog as a Coq library. This allows us to mechanize our metatheory and produce certificates. Our algorithm, which runs inside the Coq prover, produces reusable Coq theorems of parser equivalence. At a technical level, our Coq development combines classical techniques based on predicate transformers, domain-specific optimizations, and a plugin to interface with SMT solvers, to facilitate effective automation. We apply Leapfrog to several benchmarks and find that it is able to scale up to handle realistic protocols.

The contributions of this paper are as follows:

- We develop a parser model based on automata extended with domain-specific features (Section 3).
- We design efficient algorithms for establishing the equivalence of parsers based on symbolic and up-to techniques (Sections 4 and 5).
- We realize our approach in a Coq-based framework for automatically constructing equivalence proofs (Section 6). Crucially, our design integrates off-the-shelf SMT solvers into a verification loop within Coq.
- We explore Leapfrog's expressiveness and scalability (Section 7), finding that it can handle common protocol verification challenges and can perform translation validation for an existing parser compiler.

Overall, we believe Leapfrog represents a promising step toward the vision of certified proofs for protocol parsers.

2 Overview

We now give a high-level overview of our automata model and equivalence checking framework through an illustrative example. Suppose we would like to parse packets with MPLS [54] and UDP [46] headers. An MPLS header is a sequence of 32-bit *labels*. Rather than prefix this sequence with its length, the MPLS format marks the end of the sequence with a label whose 24th bit is 1. This is analogous to the role of the null terminator in C strings. In our example, the MPLS header is followed by an 8-byte UDP header.

We can parse this format in our *P4 automata* model of parsers, which is based on a subset of P4. A P4 automaton is a state machine that parses a packet bitstring into a collection of *headers* stored in global variables, ultimately either accepting or rejecting the packet. Each state in a P4 automaton contains a program that may assign to variables and extract some number of bits from the front of the packet into a variable. For instance, an `extract(h, 64)` operation removes 64 bits from the front of the packet and stores them into the header `h`. Next, the machine transitions to a new state by branching on the contents of its header variables. This is accomplished with either an unconditional transition of the form `goto state` or a conditional transition of the form `select(e) { pat => state }`. A `select` evaluates `e` and transitions to the state associated with the first matching pattern in `pat`.

We give two P4 automata for our format in Figure 1. The first automaton is a reference parser with a state `q1` that parses MPLS and another state `q2` that parses UDP. The second automaton has been *vectorized* to parse two MPLS labels at a time in state `q3`. When the second label is the bottom of the stack, the vectorized parser goes on to handle UDP normally (`q4`). If the first label is the bottom of the stack, however, the vectorized parser marshals the 32 bits of the ill-fated second label into UDP, along with the remaining 32 bits of UDP header data remaining in the packet (`q5`).

The two parsers in Figure 1 each use different names for the header (e.g., `mpls` vs. `old`/`new`). Also, those headers are overwritten multiple times—a real P4 parser would use an array-like data structure called a *header stack* to store the labels [16, Section 8.17]. Our language does not support header stacks directly, although they can be emulated. Here, we omit this detail for simplicity, and we focus on proving that the parsers accept the same sets of packets. Leapfrog can also be used to prove relational properties involving the header values—see Section 7 for details.

Tractable Equivalence Checking. Our equivalence algorithm is inspired by Moore's algorithm [31] applied to the domain of P4 automata. It is a worklist-style algorithm that begins with a coarse approximation of language equivalence and iteratively refines it by analyzing the joint state space of the two automata. P4 automata have finitely many states and their header variables are fixed-size bitvectors, so their configuration spaces are finite. However, because of the large bitvectors encoded in their stores, P4 automata may have an intractably large configuration space. For instance, the automata in Figure 1 have a joint configuration space on the order of $2^{128} \approx 10^{38}$ states! So, naive bisimulation-based approaches will never be tractable for realistic automata. We address this challenge by representing large relations *symbolically*, rather than keeping track of concrete sets of configuration pairs.

h	$\in$	H	header names
n	$\in$	$\mathbb{N}$	natural numbers
bv	$\in$	$\{0,1\}^*$	bitvector
e	$::=$	h	headers
	$\mid$	bv	bitvectors
	$\mid$	$e[n_1 : n_2]$	bitslices
	$\mid$	e_1 ++ e_2	concatenation
pat	$::=$	bv	exact match
	$\mid$	_	wildcard
q	$\in$	$Q \cup \{\text{accept}, \text{reject}\}$	state names
c	$::=$	$\overline{pat} \Rightarrow q$	select case
tz	$::=$	$\texttt{goto}(q)$	direct
	$\mid$	$\texttt{select}(\overline{e})\{\overline{c}\}$	select
op	$::=$	$\texttt{extract}(h)$	extract
	$\mid$	$h := e$	assign
	$\mid$	$op_1;\ op_2$	sequence
st	$::=$	$q\,\{op; tz\}$	states ($q \in Q$)
aut	$::=$	$\overline{st}$	P4 automaton

Figure 2. Internal syntax for P4 automata.

Furthermore, we prune the configuration space from the start using a simple reachability analysis. This lets us avoid spurious search steps through unreachable configurations.

In the automata-theoretic semantics of the reference MPLS parser, the `extract(udp, 64)` call performs 64 steps that read one bit of the packet into the buffer. The 64th step empties the buffer into the `udp` header variable, and transitions to the accept state. This bit-by-bit approach is needed to relate parsers that read the packet in differently-sized chunks—as is the case for states `q1` and `q3` in Figure 1. However, a naive search for a bisimulation that treats each step separately would have a huge symbolic state and too many SMT queries. To counteract this, we introduce *bisimulations with leaps*, which keeps the symbolic state compact and avoids redundant SMT queries, processing multiple consecutive steps in each iteration. Together, these optimizations make it feasible to compute bisimulations for realistic parsers.

Certifying Parser Equivalence. To make Leapfrog usable in larger developments, we need it to produce reusable proof certificates that can be checked by the Coq kernel. Obtaining full certificates of equivalence from a push-button tool is a significant engineering challenge. Rather than write a solver in Coq for our verification conditions, which sit roughly in the first-order theory of bitvectors, we chose to use external SMT solvers. This had engineering and performance benefits, but getting Coq and external solvers to

work together still posed several challenges. First, existing interfaces between Coq and SMT solvers did not meet our needs. We tried using existing plugins for proving Coq theorems with external solvers [4, 19], but found that they scaled poorly or lacked support for key logical operators. To address this, we developed a first-order theory of bitvectors in Coq, as well as a plugin that pretty-prints this logic in SMT-LIB format [7] and discharges the query using an off-the-shelf solver. We did not implement proof reconstruction, which converts the SMT solver refutation into a Coq proof term. Consequently the solver output and our pretty-printer must be trusted, although this restriction could be lifted in future work.

Second, in order to target this low-level logic and improve the scalability of our tool, we developed and verified a chain of compilation steps that go from the high-level logic used in our algorithm to the low-level logic sent to solvers. This process compiles away features like finite maps, but also performs algebraic simplifications and other rewrites to keep the size and complexity of our SMT queries under control.

Third, in order to communicate with SMT solvers, our algorithm needed to perform I/O, which Coq programs are generally not allowed to do. Coq tactics, however, are allowed to perform I/O, because the proof they produce is still checked by the Coq kernel. We rephrased our algorithm as a proof search problem (c.f. Figure 4) and developed a custom Coq tactic as an escape hatch, allowing the algorithm to consult an SMT solver while still producing a Coq certificate (modulo the soundness of the solver and our plugin).

3 Parser Model

We now describe *P4 automata* (P4As), an automata-theoretic model close to P4's parsing language [16]. A P4A is a state machine that (1) decides whether to accept a packet and (2) builds a data structure (the *store*) using the packet data. The store consists of bitvectors called *headers*,[2] and is a representation of the (partially) parsed packet used within the parser, but also in later processing phases. If a P4A consumes data in each state, it terminates on finite packets.[3]

For example, state `q1` of the reference MPLS/UDP parser in Figure 1 extracts 32 bits into the `mpls` header, before looping back to `q1` or transitioning to `q2` to parse a UDP header.

Concretely, a P4A is composed of states, each of which acts in two steps: first, it runs its internal program, which consumes some bits from the packet and updates the headers; then, it decides (based on the store) whether to accept (resp. reject) the packet by transitioning to the accept (resp. reject) state, or continue processing the remainder elsewhere. This second step defines the state's transitions.

[2]Initial header values are undefined in P4 [16, Sections 6.7 and 8.22]; our semantics considers them part of the packet.

[3]Such a restriction is allowed by P4 specification [16, Sec. 12.10].

3.1 Syntax

The syntax for P4As is best understood by example. Consider the two programs in Figure 1. Together, these contain five states, named `q1` through `q5`. Each state contains code, consisting of assignments and `extract` statements, and ending in a `select` or `goto` statement that defines the outgoing transitions. Headers function as variables whose scope and lifetime is shared between states. For instance, in `q5`, the vectorized parser extracts bits into `tmp`, and then stores the contents of `new` and `tmp` in `udp`, before accepting.

We formally define the syntax in Figure 2, parameterized over a finite set of states Q, and a finite set of *header names* H. Each header $h \in H$ has an associated *size* $\mathrm{sz}(h) \in \mathbb{N}^+$. We refrain from specifying these parameters explicitly, as they can be inferred from the program text. For instance, in the P4A on the left in Figure 1, $Q = \{\texttt{q1}, \texttt{q2}\}$, and $H = \{\texttt{mpls}, \texttt{udp}\}$, while header sizes are $\mathrm{sz}(\texttt{mpls}) = 32$ and $\mathrm{sz}(\texttt{udp}) = 64$.

P4As associate with each state $q \in Q$ an operation block $op(q)$ and transition block $tz(q)$. Crucially, we require that least one call to `extract` appears in the operations of each state. This guarantees that each state makes some progress on the packet, which ensures termination of both the parsing process and our equivalence checking algorithm.

3.2 Semantics

To provide a semantics for P4As, we first assign a semantics to the operation and transition code associated with each state. We fix a P4A *aut* with states Q, headers H and sizes sz. We write $|bv|$ for the length of $bv \in \{0,1\}^*$. We define S as the finite set of functions $s : H \to \{0,1\}^*$ where $|s(h)| = \mathrm{sz}(h)$.

We first give a semantics to expressions.

Definition 3.1 (Expression Semantics)**.** Let $w, x \in \{0,1\}^*$. We write wx for their concatenation. If $n_1, n_2 \in \mathbb{N}$, we write $w[n_1 : n_2]$ for the zero-indexed substring starting at position $\min(n_1, |w| - 1)$ and ending at $\min(n_2, |w| - 1)$, inclusive.

Given an expression e, we inductively define its semantics in the form of a function $[\![e]\!]_{\mathcal{E}} : S \to \{0,1\}^*$, as follows:

$$[\![h]\!]_{\mathcal{E}}(s) = s(h) \qquad [\![e[n_1 : n_2]]\!]_{\mathcal{E}}(s) = [\![e]\!]_{\mathcal{E}}(s)[n_1 : n_2]$$
$$[\![bv]\!]_{\mathcal{E}}(s) = bv \qquad [\![e_1 \texttt{ ++ } e_2]\!]_{\mathcal{E}}(s) = [\![e_1]\!]_{\mathcal{E}}(s)[\![e_2]\!]_{\mathcal{E}}(s)$$

There is a straightforward typing judgement $\vdash_{\mathcal{E}}$, where $\vdash_{\mathcal{E}} e : n$ implies that $|[\![e]\!]_{\mathcal{E}}(s)| = n$. We elide this definition.

Next, we give a semantics to operations and transitions, which constitute the code that can appear inside states.

Definition 3.2 (Operation Semantics)**.** When $s \in S$, $h \in H$ and $v \in \{0,1\}^{\mathrm{sz}(h)}$, we use $s[v/h]$ to denote the store where $s[v/h](h) = v$, and $s[v/h](h') = s(h')$ for all $h' \in H \setminus \{h\}$.

For operations op, define $\|op\| \in \mathbb{N}$ inductively, as follows:

$$\|h := e\| = 0 \qquad \|\texttt{extract}(h)\| = \mathrm{sz}(h)$$
$$\|op_1;\ op_2\| = \|op_1\| + \|op_2\|$$

Intuitively, $\|op\|$ is the exact number of bits necessary to execute all `extract` statements that appear in *op*.

For each block of operations *op*, we define a *partial* function $[\![op]\!]_O : S \times \{0,1\}^* \rightharpoonup S \times \{0,1\}^*$, as follows:

$$[\![h := e]\!]_O(s, w) = \langle s[v/h], w\rangle \quad (\text{if } [\![e]\!]_{\mathcal{E}}(s) = v \text{ and } |v| = \mathrm{sz}(h))$$
$$[\![\texttt{extract}(h)]\!]_O(s, xy) = \langle s[x/h], y\rangle \quad (\text{if } |x| = \mathrm{sz}(h))$$
$$[\![op_1;\ op_2]\!]_O(s, w) = [\![op_2]\!]_O([\![op_1]\!]_O(s, w))$$

There exists a type judgement $\vdash_O$ such that if $\vdash_O op$ then $[\![op]\!]_O(s, w) \in S \times \{\epsilon\}$ for all $bv \in \{0,1\}^{\|op\|}$.

Definition 3.3 (Pattern and Transition Semantics)**.** For a pattern *pat*, define $[\![pat]\!]_{\mathcal{P}} \subseteq \{0,1\}^*$ by case distinction:

$$[\![bv]\!]_{\mathcal{P}} = \{bv\} \qquad [\![\_]\!]_{\mathcal{P}} = \{0,1\}^*$$

Given a transition block *tz*, we define a partial function $[\![tz]\!]_{\mathcal{T}} : S \rightharpoonup Q \cup \{\text{accept}, \text{reject}\}$ inductively, as follows:

$$[\![\texttt{goto}(q)]\!]_{\mathcal{T}}(s) = q \qquad [\![\texttt{select}(\overline{e})\{\}]\!]_{\mathcal{T}}(s) = \text{reject}$$

$$\frac{\forall i.\ [\![e_i]\!]_{\mathcal{E}}(s) = v_i \qquad q' = [\![\texttt{select}(\overline{e})\{\overline{c}\}]\!]_{\mathcal{T}}(s)}{[\![\texttt{select}(\overline{e})\{\overline{pat} \Rightarrow q;\ \overline{c}\}]\!]_{\mathcal{T}}(s) = \begin{cases} q & \forall i.\ v_i \in [\![pat_i]\!]_{\mathcal{P}} \\ q' & \text{otherwise} \end{cases}}$$

As before, a straightforward type judgement $\vdash_{\mathcal{T}}$ can be formulated such that $\vdash_{\mathcal{T}} tz$ implies that $[\![tz]\!]_{\mathcal{T}}(s)$ is defined.

We now have the ingredients necessary to define the dynamics of a P4A in terms of a deterministic finite automaton (DFA). To facilitate the comparison of P4As that consume packets in differently-sized chunks, this DFA buffers until it has read enough bits to execute the `extract` blocks associated with the current state. We first precisely define a configuration of a P4A, as follows.

Definition 3.4 (Configurations)**.** A *configuration* is a triple

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

where $|w| < \|op(q)\|$ if $q \in Q$, and $w = \epsilon$ otherwise. We write C for the (finite) set of configurations, and F for the *accepting configurations*: $\{\langle \text{accept}, s, \epsilon\rangle \in C : s \in S\}$.

We can define a bit-by-bit step function on configurations, which implements the idea of filling up the store before actuating the transition, outlined above.

Definition 3.5 (Configuration Dynamics)**.** We define the step function $\delta : C \times \{0,1\} \rightharpoonup C$ as follows. Let $c = \langle q, s, w\rangle \in C$. If $q \in Q$, then we define $\delta(c, b)$ by setting

$$\delta(c, b) = \begin{cases} \langle q, s, wb\rangle & |wb| < \|op(q)\| \\ \langle [\![tz(q)]\!]_{\mathcal{T}}(s'), s', \epsilon\rangle & [\![op(q)]\!]_O(s, wb) = \langle s', \epsilon\rangle \end{cases}$$

Otherwise, if $q \in \{\text{accept}, \text{reject}\}$, then $\delta(c, b) = \langle \text{reject}, s, \epsilon\rangle$.

There exists a type judgement $\vdash_{\mathcal{A}}$ such that $\vdash_{\mathcal{A}} aut$ implies that δ is well-defined and total; again, we omit its definition.[4]

To match the behavior of P4 parsers, accepting states should not parse any further input. As a consequence a configuration of the form $\langle \text{accept}, s, \epsilon\rangle$ steps unconditionally to $\langle \text{reject}, s, \epsilon\rangle$.

Put together, $\langle C, \delta, F\rangle$ is a DFA. We can therefore define the language semantics of our parser *aut* as a function $[\![aut]\!]_{\mathcal{A}} : Q \times S \to 2^{\{0,1\}^*}$, where 2^X denotes the set of subsets of a set X. This semantics associates with each initial state and store the set of bit-strings that lead to an accepting configuration.

Definition 3.6 (Multi-Step Configuration Dynamics)**.** We can lift δ to $\delta^* : C \times \{0,1\}^* \rightharpoonup C$ as follows:

$$\delta^*(c, \epsilon) = c \qquad \delta^*(c, bw) = \delta^*(\delta(c, b), w)$$

Given $c \in C$, we define its *language* $L(c) \subseteq \{0,1\}^*$ as follows:

$$L(c) = \{w \in \{0,1\}^* : \delta^*(c, w) \in F\}$$

Given $q \in Q$ and $s \in S$, we define $[\![aut]\!]_{\mathcal{A}}(q, s) = L(q, s, \epsilon)$.

Our semantics embeds the initial store in the start state. Our equivalence checking procedure can help verify that packet acceptance does not depend on the initial store value.

4 Symbolic Equivalence Checking

Many verification questions about P4As can be phrased as questions about the underlying DFAs. For instance, let aut_1 and aut_2 be the P4As from Figure 1, suppose we want to verify that they accept the same packets when started from certain initial states `q1` and `q3`, regardless of their initial store. To do this, we could check whether $L(\texttt{q1}, s_1, \epsilon) = L(\texttt{q3}, s_2, \epsilon)$ for all $s_1, s_2 \in S$. This problem is decidable, because S is finite and language equivalence of DFAs is decidable [40].

Unfortunately, the DFA arising from a P4A may be extremely large: every $q \in Q$ contributes $|S| \times 2^{\|op(q)\|-1}$ configurations. Even for simple parsers, this leads to an intractably large configuration space. For instance, for the reference MPLS parser on the left in Figure 1, $|S| = 2^{96}$; a back-of-the-envelope calculation then tells us that $|C| \geq 10^{38}$.

Moreover, we anticipate that a large portion of the configuration space is reachable, and should therefore be taken into consideration. This is because parsers tend to propagate every bit of the packet into the store in order to facilitate packet reconstruction for forwarding. Off-the-shelf algorithms for DFAs are therefore unlikely to scale to this setting.

In this section, we develop an algorithm that can answer several questions about P4As. This algorithm mitigates state space explosion by representing configurations symbolically. Our presentation focuses on deciding language equivalence of configurations. As a consequence, the procedure can be

[4] Our requirement that each state extracts some bits is part of this typing judgement, and in fact necessary in order for the definition of δ to be useful. Because a transition is triggered by the final bit, if $\|op(q)\| = 0$ for some state, then there would be no way to actuate this transition.

x	$\in$	Var	variables
be	::=	bv	literal
	|	$\text{buf}^<, \text{buf}^>$	left and right buffer
	|	$h^<, h^>$	left and right header
	|	x	variable
	|	$be[n_1 : n_2]$	slice
	|	$be_1 \mathbin{+\!\!+} be_2$	concat
p	::=	$be_1 = be_2$	bitvector equality
	|	$q^<, q^>$	left and right state assertion
	|	$n^<, n^>$	left and right buffer length
ϕ	::=	$\bot$	bottom
	|	p	atomic predicate
	|	$\phi_1 \implies \phi_2$	implication

Figure 3. Syntax for relations on configurations.

thought of as a variation on Moore's algorithm [40]. We discuss more general applications in Section 7.

For the sake of simplicity, we fix a P4A *aut* with underlying DFA $\langle C, \delta, F\rangle$ for the remainder of this section. One can compare configurations in two different P4As by taking their disjoint sum, renaming states and headers as necessary.

4.1 A Symbolic Approach

A sound and complete method to show that two configurations of our DFA $\langle C, \delta, F\rangle$ accept the same language is to demonstrate that they are related by a *bisimulation* [32], i.e., a relation $R \subseteq C \times C$ such that when $c_1\ R\ c_2$, (1) $c_1 \in F$ if and only if $c_2 \in F$; and (2) $\delta(c_1, b)\ R\ \delta(c_2, b)$ for all $b \in \{0, 1\}$.

A language equivalence checking algorithm for DFAs typically tries to build some form of bisimulation. Because C may be very large, representing a bisimulation by listing its constituent pairs becomes intractable quickly. Luckily, we can write down bisimulations symbolically.

Example 4.1. Suppose *aut* is the disjoint sum of the MPLS parsers displayed in Figure 1. Let R be the smallest relation on C satisfying the following rules for all $s_1, s_2 \in S$:

$$\frac{w \in \{0,1\}^{32} \quad x \in \{0,1\}^* \quad |x| < 32}{\langle \mathsf{q2}, s_1, wx\rangle\ R\ \langle \mathsf{q5}, s_2, x\rangle} \qquad \frac{q \in \{\text{accept}, \text{reject}\}}{\langle q, s_1, \epsilon\rangle\ R\ \langle q, s_2, \epsilon\rangle}$$

R is a bisimulation, and thus all configurations related by R have the same language. Clearly, this representation is much more concise than listing the contents of R explicitly.

To systematically represent and manipulate symbolic relations on configurations, we propose the syntax in Figure 3. Its formulas are generated by equality assertions between expressions built over the buffers and stores of both configurations, as well as predicates about states and buffer lengths. We also include variables $x \in \text{Var}$ for later use. We omit conjunction ($\vee$) and disjunction ($\wedge$) from the syntax to keep our definitions brief. They are derivable from $\implies$ and $\bot$, so we will still use them in the sequel as abbreviations.

Example 4.2. We can describe the pairs matching the second rule from Example 4.1 using the following formula

$$\phi = (\text{accept}^< \wedge \text{accept}^>) \vee (\text{reject}^< \wedge \text{reject}^>)$$

Given $n < 32$, we can choose the formula ϕ_n to symbolize the first rule from Example 4.1 where $|x| = n$:

$$\phi_n = (n + 32)^< \wedge q2^< \wedge n^> \wedge q5^> \wedge \text{buf}^<[0:31] = \text{buf}^>$$

In total, R is represented by the formula $\phi \vee \phi_0 \vee \cdots \vee \phi_{31}$.

Definition 4.3. A *valuation* is a function $\sigma : \text{Var} \to \{0, 1\}$. For every bitvector expression be and valuation σ, we define $[\![be]\!]^\sigma_{\mathcal{B}} : C \times C \to \{0, 1\}^*$ inductively as follows, where $c^<, c^> \in C$ are such that $c^\lessgtr = \langle q^\lessgtr, s^\lessgtr, w^\lessgtr\rangle$ for $\lessgtr \in \{<, >\}$:

$$[\![bv]\!]^\sigma_{\mathcal{B}}(c^<, c^>) = bv \qquad [\![\text{buf}^\lessgtr]\!]^\sigma_{\mathcal{B}}(c^<, c^>) = w^\lessgtr$$

$$[\![x]\!]^\sigma_{\mathcal{B}}(c^<, c^>) = \sigma(x) \qquad [\![h^\lessgtr]\!]^\sigma_{\mathcal{B}}(c^<, c^>) = s^\lessgtr(h)$$

The cases for slices and concatenation are as in Definition 3.1.

For a formula ϕ and valuation σ, define $[\![\phi]\!]^\sigma_{\mathcal{L}}$ as the least relation on C satisfying the following rules for all $c^<, c^> \in C$, where $c^\lessgtr = \langle q^\lessgtr, s^\lessgtr, w^\lessgtr\rangle$, and $n^\lessgtr = |w^\lessgtr|$ for $\lessgtr \in \{<, >\}$:

$$c^<\ [\![q^<]\!]^\sigma_{\mathcal{L}}\ c^> \qquad \frac{[\![be_2]\!]_{\mathcal{B}}(c_1, c_2, \sigma) = [\![be_2]\!]_{\mathcal{B}}(c_1, c_2, \sigma)}{c_1\ [\![be_1 = be_2]\!]^\sigma_{\mathcal{L}}\ c_2}$$

$$c^<\ [\![q^>]\!]^\sigma_{\mathcal{L}}\ c^>$$

$$c^<\ [\![n^<]\!]^\sigma_{\mathcal{L}}\ c^> \qquad \frac{c_1\ [\![\phi_1]\!]^\sigma_{\mathcal{L}}\ c_2 \implies c_1\ [\![\phi_2]\!]^\sigma_{\mathcal{L}}\ c_2}{c_1\ [\![\phi_1 \implies \phi_2]\!]^\sigma_{\mathcal{L}}\ c_2}$$

$$c^<\ [\![n^>]\!]^\sigma_{\mathcal{L}}\ c^>$$

Let ϕ and ψ be formulas. We write $[\![\phi]\!]_{\mathcal{L}}$ for the relation on C where $c_1\ [\![\phi]\!]_{\mathcal{L}}\ c_2$ if and only if $c_1\ [\![\phi]\!]^\sigma_{\mathcal{L}}\ c_2$ for all valuations σ. Finally, we write $\phi \vDash \psi$ when $[\![\phi]\!]_{\mathcal{L}} \subseteq [\![\psi]\!]_{\mathcal{L}}$.

Note that because there are finitely many configurations and valuations, entailments are decidable. We will revisit this particular decision problem in Sections 5 and 6.

We can now define symbolic bisimulations, as follows.

Definition 4.4 (Symbolic Bisimulation)**.** A *symbolic bisimulation* is a formula ϕ such that $[\![\phi]\!]_{\mathcal{L}}$ is a bisimulation.

Finding a (symbolic) bisimulation is a sound and complete method to establish language equivalence of states.

Lemma 4.5. *For formulas ϕ, the following are equivalent:*

1. *There exists a symbolic bisimulation ψ such that $\phi \vDash \psi$.*
2. *There exists a bisimulation R such that $[\![\phi]\!]_{\mathcal{L}} \subseteq R$.*
3. *If $c_1\ [\![\phi]\!]_{\mathcal{L}}\ c_2$, then $L(c_1) = L(c_2)$.*

4.2 The Weakest Symbolic Bisimulation

To search for a symbolic bisimulation, we turn to Moore's algorithm [40]. In its concrete formulation, this algorithm approximates the largest (coarsest) bisimulation from above, by iteratively removing non-bisimilar pairs. Eventually, the process stops, at which point the remaining pairs must be bisimilar; hence, the computed relation is the largest bisimulation. Two configurations are related by some bisimulation

Algorithm 1: Symbolic equivalence checking.

Input: A formula ϕ representing initial states.
Input: A set of formulas I s.t. for all $c_1, c_2 \in C$,

$$[\forall \psi \in I.\ c_1 \ [\![\psi]\!]\ c_2] \Leftrightarrow [c_1 \in F \Leftrightarrow c_2 \in F]$$

Input: A function WP s.t. for all ψ, and $c_1, c_2 \in C$,

$$[\forall b \in \{0,1\}.\ \delta(c_1, b)\ [\![\psi]\!]\ \delta(c_2, b)] \Leftrightarrow c_1 \left[\!\!\left[\bigwedge \mathsf{WP}(\psi)\right]\!\!\right] c_2$$

Output: **true** if and only if for all $c_1, c_2 \in C$ with $c_1 \ [\![\phi]\!]_{\mathcal{L}}\ c_2$, it holds that $L(c_1) = L(c_2)$

```
1 R ← ∅; T ← I
2 while T ≠ ∅ do
3 |   pop ψ from T
4 |   if not ⋀R ⊨ ψ then
5 |   |   R ← R ∪ {ψ}
6 |   |   T ← T ∪ WP(ψ)
7 return true if φ ⊨ ⋀R, otherwise false
```

if and only if they are related by the largest bisimulation, so the algorithm concludes with a simple containment check.

Moore's algorithm can be made symbolic, by representing the current overapproximation as a formula and successively strengthening it, thus converging to the weakest symbolic bisimulation. We present an abstract formulation of this process in Algorithm 1. The algorithm has two parameters.

- The formulas in I constitute the initial overapproximation of the weakest symbolic bisimulation. In Section 7, we consider instantiations of I that can be used to verify different but related properties.
- The function WP takes a formula ϕ, and outputs a set of formulas whose conjunction represents a *weakest precondition* of ϕ, in the sense that two configurations are related by *all* formulas in $\mathsf{WP}(\phi)$ if and only if they step into configurations related by ϕ.

Algorithm 1 builds the weakest symbolic bisimulation as a set of conjuncts R, maintaining a frontier T of formulas to be considered. The frontier is initially I. In each iteration, we pop a conjunct ψ from T and check if it is entailed by $\bigwedge R$. If $\bigwedge R \not\models \phi$, then ψ constitutes a novel restriction, and we add it to R. Because bisimulations are closed under steps, we add the weakest preconditions of ψ to T, to be checked later. If $\bigwedge R \models \psi$, including ψ in R would not change $\bigwedge R$, so we move on. The loop terminates when T is empty; at this point, $\bigwedge R$ will be the weakest symbolic bisimulation, and the algorithm checks $\phi \models \bigwedge R$. We instantiate the parameters momentarily; first, we show that the algorithm is correct.

Theorem 4.6. *Algorithm 1 is correct.*

Proof Sketch. For termination, note that in each iteration either $[\![\bigwedge R]\!]$ or T shrinks; hence, the algorithm must terminate.

For partial correctness, one can show the following invariants: (1) if ϕ is a symbolic bisimulation, then $\phi \models \bigwedge R \wedge \bigwedge T$; and (2) configurations related by $\bigwedge R \wedge \bigwedge T$ are equally accepting, and (3) configurations related by $\bigwedge R \wedge \bigwedge T$ step into configurations related by $\bigwedge R$. Thus, when Algorithm 1 terminates, $\bigwedge R$ must be the largest symbolic bisimulation, and we can conclude by applying Lemma 4.5. □

4.3 Instantiating the Parameters

We now sketch how we instantiate the parameters of Algorithm 1; the details are worked out in our Coq development.

For WP, the main idea is to focus on a particular subclass of formulas. First, we isolate assertions about the current state; this lets us calculate weakest preconditions on a state-by-state basis, by means of a traditional substitution-based procedure on the formula using the associated program text. Second, we isolate statements about buffer lengths; this means that when a formula in our algorithm makes a claim about the buffer contents, it does so in a context where the buffer length is known. This simplifies the analysis and generation of formulas, because we do not have to cover cases where slices go beyond the end of a bitvector.

Concretely, this format takes the following form.

Definition 4.7 (Templates)**.** A *template* is a pair $\langle q, n\rangle \in (Q \cup \{\text{accept}, \text{reject}\}) \times \mathbb{N}$ where $n < \|op(q)\|$ if $q \in Q$, and $n = 0$ otherwise. The set of all templates is T. When $t = \langle q, n\rangle$ and $\lessgtr \in \{<, >\}$, we write $t^{\lessgtr}$ as shorthand for $q^{\lessgtr} \wedge n^{\lessgtr}$.

A formula ϕ is *pure* when it does not contain state or buffer length assertions; ϕ is *template-guarded* if it is of the form $t_1^{<} \wedge t_2^{>} \implies \psi$ where $t_1, t_2 \in T$ and ψ is pure.

Let ϕ be template-guarded. We compute $\mathsf{WP}(\phi)$ by operating on the left- and right hand side, giving rise to functions $\mathsf{WP}^{<}$ and $\mathsf{WP}^{>}$, whose definitions we omit. Each takes a formula and a pair of state templates, as well as a fresh variable $x \in \mathsf{Var}$ to represent the bit to be read, and returns a formula. The relevant correctness statement is as follows.

Lemma 4.8. *Let ϕ be pure, let $x \in \mathsf{Var}$ be fresh for ϕ, and let $c^{<}, c^{>} \in C$ as well as $t \in T$. The following are equivalent:*

1. *For all $b \in \{0,1\}$, we have $\delta(c^{<}, b)\ [\![t^{<} \implies \phi]\!]_{\mathcal{L}}\ c^{>}$.*
2. *For all $t' \in T$, $c^{<}\ [\![t'^{<} \implies \mathsf{WP}^{<}(\phi, t', t, x)]\!]_{\mathcal{L}}\ c^{>}$.*

A similar equivalence holds for $\mathsf{WP}^{>}$. Furthermore, if ϕ is pure, then so are $\mathsf{WP}^{<}(\phi, x, t', t)$ and $\mathsf{WP}^{>}(\phi, x, t', t)$.

Using $\mathsf{WP}^{<}$ and $\mathsf{WP}^{>}$, we can then provide a version of WP that acts on and returns template-guarded formulas. Its definition and correctness statement is as follows.

Lemma 4.9. *Let $t_1^{<} \wedge t_2^{>} \implies \phi$ be template-guarded, and let x be fresh in ϕ. Define $\mathsf{WP}(t_1^{<} \wedge t_2^{>} \implies \phi)$ as the smallest set satisfying the following rule, for all $t_1', t_2' \in T$:*

$$\frac{\phi' = \mathsf{WP}^{<}(\mathsf{WP}^{>}(\phi', x, t_2', t_2), x, t_1', t_1)}{[t_1'^{<} \wedge t_2'^{>} \implies \phi'] \in \mathsf{WP}(t_1^{<} \wedge t_2^{>} \implies \phi)}$$

Now WP fits the requirement from Algorithm 1, when restricted to template-guarded formulas. Moreover, each of the formulas in $\mathsf{WP}(t_1^{<} \wedge t_2^{>} \implies \phi)$ is template-guarded.

By the latter property, if all formulas in I are template-guarded, then the formulas in R and T remain template-guarded. We thus instantiate I as a set of template-guarded formulas that rule out pairs containing both accepting and non-accepting configurations, as follows.

Lemma 4.10. *Let $t_{\text{accept}} = \langle \text{accept}, 0 \rangle$. Define I as the smallest set of formulas satisfying the following rule:*

$$\frac{t_1, t_2 \in T \qquad t_1 = t_{\text{accept}} \iff t_2 \neq t_{\text{accept}}}{[t_1^< \wedge t_2^> \implies \bot] \in I}$$

Now I fits the requirement from Algorithm 1.

5 Optimizing the Algorithm

We now discuss two optimizations of Algorithm 1. The first optimization refines WP and I such that fewer entailments between formulas need to be checked (line 4). The second optimization generalizes WP to compute multi-step weakest preconditions, thereby strengthening the approximation of the weakest symbolic bisimulation more quickly.

5.1 Abstract Interpretation

Algorithm 1 computes the weakest symbolic bisimulation, which relates *all* language equivalent configurations, but it cares only about the configurations related by ϕ. We can compute a symbolic bisimulation more loosely, by disregarding unreachable (and hence, irrelevant) configuration pairs.

Example 5.1. Recall the symbolic bisimulation in Example 4.2, which was sufficient to conclude language equivalence of related configurations. There was no need to compute the largest symbolic bisimulation, which involves many configuration pairs unreachable from the pairs of interest.

Of course, computing the set of reachable pairs—even symbolically—is tantamount to checking equivalence. Instead, we approximate it by analyzing the P4A to capture the pairs of reachable configurations based on their templates.

To this end, let $\rho(tz)$ denote the set of states appearing in a transaction block tz. We define $\sigma : T \to 2^T$ as follows:

$$\sigma(q, n) = \begin{cases} \{\langle q, n+1 \rangle\} & q \in Q \wedge n + 1 < \text{sz}(q) \\ \rho(tz(q)) \times \{0\} & q \in Q \wedge n + 1 = \text{sz}(q) \\ \{\langle \text{reject}, 0 \rangle\} & q \in \{\text{accept}, \text{reject}\} \end{cases}$$

When $c = \langle q, s, w \rangle \in C$, write $\lfloor c \rfloor$ for $\langle q, |w| \rangle \in T$, i.e., the unique template describing c. One can show that for all $c \in C$ and $b \in \{0, 1\}$, we have $\lfloor \delta(c, b) \rfloor \in \sigma(\lfloor c \rfloor)$. In a sense, this makes σ an abstract interpretation of δ.

Given a formula ϕ, we define reach_ϕ as the smallest relation on T satisfying the following rules:

$$\frac{c_1 \; [\![\phi]\!]_{\mathcal{L}} \; c_2}{\lfloor c_1 \rfloor \; \text{reach}_\phi \; \lfloor c_2 \rfloor} \qquad \frac{t_1 \; \text{reach}_\phi \; t_2}{\sigma(t_1) \times \sigma(t_2) \subseteq \text{reach}_\phi}$$

Usually, the pairs generated by the first rule can be inferred from ϕ. For instance, if we want to compare the languages of two initial states q_1 and q_2, then $\phi = q_1^< \wedge 0^< \wedge q_2^> \wedge 0^>$, and so the sole instantiation of the first rule yields $\langle q_1, 0 \rangle \; \text{reach}_\phi \; \langle q_2, 0 \rangle$. Computing the full contents of reach_ϕ is then a matter of applying the second rule until a fixpoint is reached.

Theorem 5.2. *Let ϕ be a formula. Algorithm 1 remains correct for this ϕ if we set I to the smallest set satisfying the rule*

$$\frac{t_1 \; \text{reach}_\phi \; t_2 \qquad t_1 = t_{\text{accept}} \iff t_2 \neq t_{\text{accept}}}{[t_1^< \wedge t_2^> \implies \bot] \in I}$$

and for each template-guarded formula $t_1^< \wedge t_2^> \implies \psi$ we set $\text{WP}(t_1^< \wedge t_2^> \implies \psi)$ to the smallest set satisfying the rule

$$\frac{t_1' \; \text{reach}_\phi \; t_2' \qquad \psi' = \text{WP}^<(\text{WP}^>(\psi', x, t_2', t_2), x, t_1', t_1)}{[t_1'^< \wedge t_2'^> \implies \phi'] \in \text{WP}(t_1^< \wedge t_2^> \implies \phi)}$$

where $x \in \text{Var}$ is some variable that is fresh for ψ.

5.2 Leaps and Bounds

Algorithm 1 operates on a bit-by-bit basis. However, most steps just fill up the buffer, and do not affect the state or store. We exploit this to compute a different form of weakest precondition, which takes as many steps as necessary to execute a "real" state-to-state transition in the P4A.

The following auxiliary notion allows us to compute the number of steps until the next transition.

Definition 5.3 (Leap Size)**.** Let $c_1, c_2 \in C$ and $c_i = \langle q_i, s_i, w_i \rangle$; we define the *leap size* $\sharp(c_1, c_2) \in \mathbb{N}$ as follows:

$$\sharp(c_1, c_2) = \begin{cases} 1 & q_1, q_2 \notin Q \\ \|tz(q_1)\| - |w_1| & q_1 \in Q, q_2 \notin Q \\ \|tz(q_2)\| - |w_2| & q_1 \notin Q, q_2 \in Q \\ \min(\|tz(q_1)\| - |w_1|, \\ \qquad \|tz(q_2)\| - |w_2|) & q_1, q_2 \in Q \end{cases}$$

We can use leap size to define a notion of (symbolic) bisimilarity that can take larger steps; this will help us to formally justify the soundness of multi-step weakest preconditions.

Definition 5.4 (Bisimulation with Leaps)**.** A *bisimulation with leaps* is a relation $R \subseteq C \times C$, such that for all $c_1 \; R \; c_2$, (1) $c_1 \in F$ if and only if $c_2 \in F$, and (2) $\delta^*(c_1, w) \; R \; \delta^*(c_2, w)$ for all $w \in \{0, 1\}^{\sharp(c_1, c_2)}$. A *symbolic bisimulation with leaps* is a formula ϕ such that $[\![\phi]\!]_{\mathcal{L}}$ is a bisimulation with leaps.

Bisimulations with leaps can be more concise because they do not need to constrain configurations where both P4A are just buffering input, waiting for the next transition.

Example 5.5. Recall the bisimulation from Example 4.1. This relation contains the bisimulation with leaps R', which is the smallest relation satisfying the rules

$$\frac{w \in \{0, 1\}^{32}}{\langle \text{q2}, s_1, w \rangle \; R' \; \langle \text{q5}, s_2, \epsilon \rangle} \qquad \frac{q \in \{\text{accept}, \text{reject}\}}{\langle q, s_1, \epsilon \rangle \; R' \; \langle q, s_2, \epsilon \rangle}$$

Bisimilarity with leaps is a sound and complete proof principle for language equivalence, which we record as follows.

Table 1. Concepts from earlier in this paper and their realizations in the implementation.

Paper name	Coq name	Implemented as
aut (Figure 2)	`Syntax.t`	Dependent record
e (Figure 2), $\vdash_{\mathcal{E}}$	`expr`	Type-indexed ind.
WP	`wp`	Gallina function
$\bigwedge R \vDash \psi$	`interp_entailment`	Gallina function
$\phi \vDash \bigwedge R$	`interp_entailment'`	Gallina function
Bisimilarity	`bisimilar`	Coinductive relation
Algorithm 1	`pre_bisimulation`	Inductive relation
if $\bigwedge R \vDash \psi \ldots$	`decide_entailment`	L_{tac}

Lemma 5.6. *Let ϕ be a formula. The following are equivalent:*

1. *There exists a symbolic bisim. with leaps ψ s.t. $\phi \vDash \psi$.*
2. *If $c_1 \ [\![\phi]\!]_{\mathcal{L}} \ c_2$, then $L(c_1) = L(c_2)$.*

We can adapt Algorithm 1 to calculate the weakest symbolic bisimulation with leaps instead, if we adapt the axiomatization of the weakest precondition operator, as follows.

Theorem 5.7. *Algorithm 1 remains correct if we change the condition on* WP *to require that, for all formulas ϕ and all $c_1, c_2 \in C$, the following equivalence holds:*

$$\forall w \in \{0,1\}^{\sharp(c_1,c_2)}.\ \delta^*(c_1, w) \ [\![\phi]\!]_{\mathcal{L}} \ \delta^*(c_2, w)$$
$$\iff \forall \psi \in \mathsf{WP}(\phi).\ c_1 \ [\![\psi]\!]_{\mathcal{L}} \ c_2$$

We can adapt the existing definition of WP to conform to this specification: simply repeat $\mathsf{WP}^<$ and $\mathsf{WP}^>$ as many times as is indicated by the source templates t_1' and t_2'.

5.3 Combining Optimizations

The optimizations discussed are largely orthogonal. However, their combination naturally gives rise to a third optimization, where reach_ϕ is computed using leaps as well. This results in an algorithm that computes a symbolic bisimulation with leaps that does not constrain intermediate (buffering) configurations. We refer to the Coq development for full details.

6 Implementation

We implement Leapfrog in Coq [8, 17] using the Equations plugin [49, 50]. See Table 1 for a summary of how concepts from the formal development in Sections 3 to 5 map to Coq notions. Although an implementation in a different language might be more efficient, our use of Coq produces rich semantic automata definitions and reusable proofs of equivalence. These artifacts are defined in Coq's expressive higher-order logic, so they can be reused and composed with other mechanized logics hosted in Coq like the Mathematical Components library [38] or verification tools like the Verified Software Toolchain [1].

```
Inductive pre_bisimulation
  : conf_rel → list conf_rel → list conf_rel →
    Prop :=
| Skip: forall phi R psi T,
    pre_bisimulation phi R T →
    interp_entailment R psi →
    pre_bisimulation phi R (psi :: T)
| Extend: forall phi R psi T,
    pre_bisimulation (psi :: R) (T ++ wp psi) →
      interp_entailment R psi →
    pre_bisimulation R (psi :: T)
| Done: forall phi R,
    interp_entailment' phi R →
    pre_bisimulation phi R [].
```

Figure 4. Algorithm 1 as an inductive relation in Coq.

6.1 Automated Proof Search

The most direct way to implement algorithms in Coq is by writing them as functions in Gallina, Coq's functional programming language, but unfortunately Gallina does not have I/O. As a consequence a Gallina implementation of Algorithm 1 would have to include a hand-written decision procedure for entailments $\bigwedge R \vDash \psi$. We instead realize Algorithm 1 in Coq as an inductive relation (Figure 4), so we can rely on external SMT solvers to handle entailments. This has the added benefit of sidestepping Coq's termination checker.[5] The algorithm is run by performing proof search within the inductive relation, and each step of the search proceeds by checking an entailment in the high-level automata logic. While the logic of entailments is close to SMT's theory of bitvectors, it also has richer terms that need to be desugared (for example a finite-map encoding of the program store, constraints on the input packet length, constraints on the automata states, etc.).

6.2 Reduction to SMT

To reach a low-level logic amenable to off-the-shelf solvers, we simplify formulas before checking them, through a chain of verified simplifications and translations (Figure 6).

This compilation turns formulas from the high-level logic ConfRel into low-level first-order formulas over bitvectors, FOL(BV). In order, the implementation performs (1) algebraic simplifications, (2) template filtering, (3) FOL compilation, and (4) store elimination. We now elaborate on each step.

First, we use smart constructors to apply local algebraic simplifications. Each application of the weakest precondition operator increases the size of a formula, so these simplifications help prevent the formulas from growing too quickly.

Second, we perform template filtering to discard unused premises from entailments. Entailments have the form

$$\bigwedge \phi_i \implies \psi_i \vDash \phi \implies \psi,$$

[5] In particular, our pen-and-paper termination proof of Algorithm 1 does not directly translate to Coq's guarded primitive recursion [28].

```
Lemma small_filter_equiv:
  lang_equiv_state
    (P4A.interp IncrementalBits.aut)
    (P4A.interp BigBits.aut)
    IncrementalBits.Start
    BigBits.Parse.
Proof.
  solve_lang_equiv_state_axiom
    IncrementalBits.state_eqdec
    BigBits.state_eqdec
    false.
Time Qed.
```

Figure 5. A Coq proof that the states `Start` and `Parse` of two automata named `IncrementalBits` and `BigBits` accept the same language (`lang_equiv_state`). The tactic `solve_lang_equiv_state_axiom` takes decision procedures for equality on the state sets of each automaton and a flag controlling a tactic optimization for large problems (here `false`).

where ϕ and all ϕ_i are templates. We discard any conjunct with $\phi_i \neq \phi$ and emit a simplified entailment $\phi \vDash \bigwedge \psi_i \implies \psi$. This puts our goal in the logic ConfRelSimp.

Third, we embed ConfRelSimp into the more general FOL(Conf) syntax, removing references to states. This fragment is the first-order theory of bitvectors and finite maps.

Finally, the store elimination pass fits formulas into the theory of bitvectors FOL(BV), by turning finite maps into first-order variables. This is necessary because some SMT solvers we targeted do not support the theory of finite maps.

6.3 Querying Solvers

The final FOL(BV) formula is serialized to SMT-LIB by a custom Coq plugin and passed to an off-the-shelf SMT solver that can be selected using a custom vernacular command. Currently, we support Z3 [21], CVC4 [6], and Boolector [44].

Before implementing our own plugin, we tried existing SMT integrations for Coq, including CoqHammer [19] and SMTCoq [4]. Neither solved our problem: CoqHammer scaled poorly due to its flexible SMT encoding and proof search procedure, while SMTCoq performed better but lacked support for quantifiers. Note however that, in contrast with our plugin, both of these tools perform *proof reconstruction* to produce a Coq proof term from solver output. Consequently, our proof search must *trust* the output of the SMT solver and our plugin. A straightforward technique for doing this is to directly `admit` the low-level goals once the plugin has given the thumbs up. This is rather error-prone because it means `admit` is used within automation, and moreover, it forces the final proof to be `Admitted` by the Coq kernel. An alternative is to use a pair of *axioms* for positive and negative validity of formulas in the low-level logic and use the output of the SMT solver to conditionally apply the axioms. While this approach is less performant because the Coq kernel checks the resulting term, it allows for closed proof terms and avoids accidental misuse of `admit` in automation.

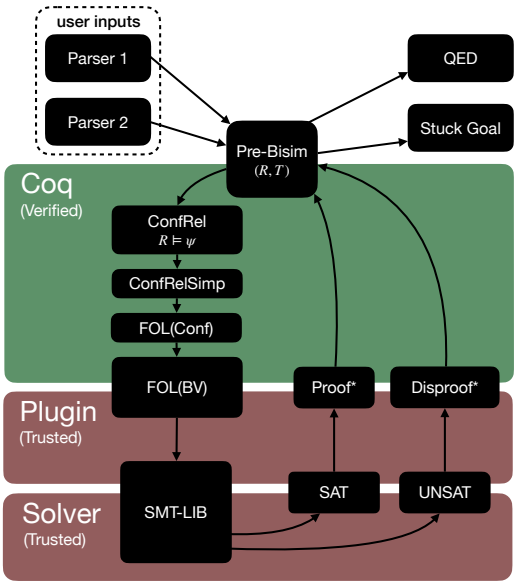


Figure 6. The Leapfrog implementation architecture. In each iteration, a ConfRel formula is checked by reduction to SMT via a chain of intermediate logics (at left). A Coq plugin pretty-prints FOL(BV) syntax to SMT-LIB syntax and invokes the SMT solver. The asterisk (*) on Proof and Disproof (at right) indicates that the plugin does not produce proofs. When the procedure halts, either the Coq goal is provable (QED), or the goal is stuck and no certificate is produced.

6.4 Soundness and Trusted Computing Base

The most important metatheoretic goal is to ensure that our algorithm produces a certificate of equivalence only when the input parsers are indeed equivalent. Towards this goal, our certificate-producing equivalence checker has a compact TCB, with soundness relying on the Coq definitions of automata and automata equivalence, the correctness of the SMT solver, the faithfulness of the pretty-printing plugin, and the soundness of the Coq typechecker extended with Streicher's axiom K [51]. The SMT solver and plugin (and the corresponding use of `admit`/axioms) are used only in the proof search algorithm and could be removed from the TCB by implementing proof reconstruction.

Our Coq development proves the soundness theorems stated in the paper, but omits completeness and termination arguments. Our proof search is really a semi-decision procedure: either the tactic finds a proof and produces a Coq proof term, or it does not find a proof and no certificate is produced. Trustworthy certificates, our main metatheoretic goal, only require a mechanization of soundness. In fact, the only termination or completeness bug we encountered arose from incorrectly interpreting failed SMT queries as UNSAT, which was a bug in the plugin and not in the algorithm itself.

Table 2. Parsers in our evaluation: **States** gives the total number of states in both parsers, **B**ranched gives the number of bits in automata transition select statements, **T**otal gives the number of bits across all variables, and **Runtime** and **Memory** give the aggregate runtime and maximum resident size. An optimal verification algorithm would need to represent $2^{\mathbf{B}}$ states, while an explicit state space would contain $2^{\mathbf{T}}$ states. An asterisk on the memory use indicates an out-of-memory exception.

	Name	States	Branched (bits)	Total (bits)	Runtime (minutes)	Memory (GB)
Utility	State Rearrangement	5	8	136	0.12	0.66
	Variable-length parsing	30	64	632	953.42	405.64
	Header initialization	10	10	320	15.95	13.71
	Speculative loop	5	2	160	4.12	3.16
	Relational verification	6	64	1056	1.68	2.07
	External filtering	6	64	1056	1.18	1.71
Applicability	Edge	28	52	3184	528.38	251.26
	Service Provider	22	50	2536	1244.5	499.80*
	Datacenter	30	242	2944	1387.95	404.50
	Enterprise	22	176	2144	217.93	66.13
	Translation Validation	30	56	3148	746.2	350.48

7 Evaluation

We evaluate Leapfrog through case studies (listed in Table 2) along two dimensions: (1) the *utility* of bisimulations for solving problems of interest in networking (and, by extension, the *expressiveness* of Leapfrog), and (2) the *applicability* of Leapfrog to real-world parsers (and, by extension, its *scalability* to non-trivial inputs).

7.1 Utility

To evaluate whether equivalence checks are useful in the networking domain, we identified *six* distinct verification tasks, and showed how they can be solved with Leapfrog.

State Rearrangement. Because parser states translate to hardware resources, it is common for compilers to merge and split parser states, to optimize the write and branch behavior for the particular hardware. We implemented a reference parser for a stylized IP and UDP/TCP protocol in which the prefix is 64 bits of IP and the suffix is either 32 bits of UDP or 64 bits of TCP (Figure 7). Note that the TCP and UDP headers share a common prefix of 32 bits. We then implemented an optimized parser that extracts the IP and common prefix, and then branches to determine how to parse the remaining bits. We used Leapfrog to show that the parsers accept the same packets, even though they do so in different ways.

Variable-Length Formats. Handling formats with variable lengths, such as type-length-value (TLV) encodings, is a common challenge in protocol parsing, because the amount of data parsed in each state depends on a previously-parsed values. We implemented a parser for Internet Protocol options [5], a common variable-length networking format. Our parser handles up to two generic options, with data-dependent lengths that range from 0 bytes to 6 bytes. We also implemented a custom parser for the Timestamp option, in which a specialized parser extracts the fields specific to its format. Again, we used Leapfrog to show that the parsers accept the same packets, even though the header formats are variable and they do so in different ways.

Header Initialization. A common error in P4 programs is reading from uninitialized headers. In parsers, this can happen when several paths converge on a common state, and the programmer has forgotten to write to a given header on one or more of the paths. For example, VLAN tags [34] are an optional 4-byte format that can appear at the end of an Ethernet frame. If the VLAN tag is present, its value can be used to influence routing behavior. However, a common bug is to accidentally branch on an uninitialized VLAN tag when it was not present in the packet. To fix this bug, one can assign a default value to missing VLAN tags. We implemented a parser for Ethernet, optional VLAN, IP, and UDP, that either parses a VLAN tag or fills it with a default value if it is missing (Figure 9 of the appendix). We used Leapfrog to check that the set of accepted packets is independent of the initial store. This check succeeds, so we conclude that the parser *not* depend on uninitialized headers.

Speculative Extraction. Many high-performance protocol parsers *speculatively extract* packet data and then make control-flow decisions based off the contents of that data. We implemented the example from Figure 1 with MPLS followed by UDP, in which the body of the optimized MPLS loop speculatively extracts two MPLS headers. If the first of these indicates the end of the header, then the parser has overshot the MPLS header, and the remaining data must be reinterpreted as a UDP packet. We used Leapfrog to verify that these parsers accept the same packets.

External Filtering. Another common idiom is to implement a lenient parser that accepts well-formed and malformed packets, and then compensate with an external filter

```
parse_ip {
  extract(ip, 64);
  select(ip[40:43]) {
    (0001) ⇒
      parse_udp
    (0000) ⇒
      parse_tcp
  }
}
parse_udp {
  extract(udp, 32);
  goto accept
}
parse_tcp {
  extract(tcp, 64);
  goto accept
}
```

```
parse_combined {
  extract(ip, 64);
  extract(pref, 32)
  select(ip[40:43]) {
    (0001) ⇒
      accept
    (0000) ⇒
      parse_suff
  }
}
parse_suff {
  extract(suff, 32);
  goto accept
}
```

Figure 7. Reference and combined parsers for a stylized IP and TCP/UDP protocol.

(e.g., by dropping packets later). Recall that the last two bytes of an Ethernet header are a type field that determines the header that follows—e.g., IPv4, IPv6, or something else. We implemented two parsers: a lenient parser that assumes the input packet is IPv6 if it is not IPv4, and a strict parser that explicitly checks the Ethernet type field and rejects other types of packets. We modeled an external filter for the lenient parser by picking an initial relation that not only requires the states to be equally accepting, but to also terminate with a store where the Ethernet type is IPv4 or IPv6. We then computed a bisimulation modulo this initial relation and prove that the two parsers are equivalent. This case study shows that Leapfrog can do more than just relate the sets of accepted packets: it can also relate the values in the stores.

Relational Verification. Leapfrog can also verify other useful relational properties of parsers. For instance, consider two parsers that extract data into differently named headers (e.g., the example from Section 2), or even with different fragments of the input packet scattered across the store. Leapfrog can be used to phrase and verify relations between parser stores. To demonstrate this, we verified that when the lenient and strict parsers from the previous case both accept some packet, there is a correspondence between the values in their stores. We picked an initial relation that requires the values for headers on the left to correspond to the values for headers on the right, provided both configurations are accepting and used Leapfrog to establish a pre-bisimulation. Compared to language equivalence, these applications do not have as much metatheory developed in Coq, where there is a lemma connecting an appropriately configured pre-bisimulation to language equivalence. However, we believe our technique is sound and could be justified in Coq.

Separately from these six tasks, we used Leapfrog to compare parsers that did *not* accept the same packets, such as the two parsers of the *external filtering* task. This was done as a sanity check to see if (1) the proof script still terminated and (2) it did not erroneously claim to prove equivalence. A failure would indicate a bug in our pen-and-paper analysis of the algorithm, or our trusted codebase. Fortunately, Leapfrog acts as expected, by reaching the end of the main loop, then failing when trying to apply the `Close` step.

7.2 Applicability

To evaluate Leapfrog's applicability to real-world parsers, we encoded the benchmarks used by the developers of the `parser-gen` tool [27]. It provides parsers for *four* different scenarios: (1) Edge, for a gateway router, (2) Service Provider, for a core router, (3) Datacenter, for a top-of-rack switch in a cloud, (4) and Enterprise, for a router in a campus or company network. Each of these parsers supports a different set of protocols depending on its intended use.[6] We translated each of these parsers into a corresponding P4A parser and used Leapfrog to perform a *self-comparison* check—i.e., we verified that each parser is equivalent to itself.

Next, we used Leapfrog to perform translation validation. The `parser-gen` framework also comes with a compiler that takes a parse graph (analogous to a P4A) and compiles it to an efficient hardware representation. The compiler models constraints at the hardware level (e.g., limiting the number of bits that can be extracted or branched on in each state) and incorporates sophisticated optimizations to make the best use of limited resources (e.g., splitting and merging states).

We ran the `parser-gen` compiler on the parser for the Edge router, which generated a hardware-level representation with states, instructions, and transitions encoded in a table—see Figure 8. We then wrote a script to translate the table representation back into a P4 automaton.[7] Finally, we used Leapfrog to check the equivalence of the two parsers.

We were able to prove that the `parser-gen` compiler preserves the semantics of the original Edge P4A automata. Hence, Leapfrog was able to validate a third-party compiler's output on its own benchmark program. Note that we designed Leapfrog before we had experience using `parser-gen`.

[6] We did not consider one of the parsers discussed in the `parser-gen` paper, Big-Union, which models the combined features from all four scenarios. Unlike the others, Big-Union does not model a typical scenario but is primarily intended for bounding hardware requirements.

[7] While the two languages are similar, the parser-gen hardware representation is different enough from P4A (mainly due to unproductive states and speculative lookahead transitions) to make the reverse translation fuzzy. Of all of the parser-gen benchmarks, we found that Edge's hardware table was the closest to P4A and required the least amount of manual repair. This technique could in principle be adapted to other parser-gen benchmarks; while they are a bit larger and could stress Leapfrog's scaling, the main challenge is a robust translation from hardware tables to P4A.

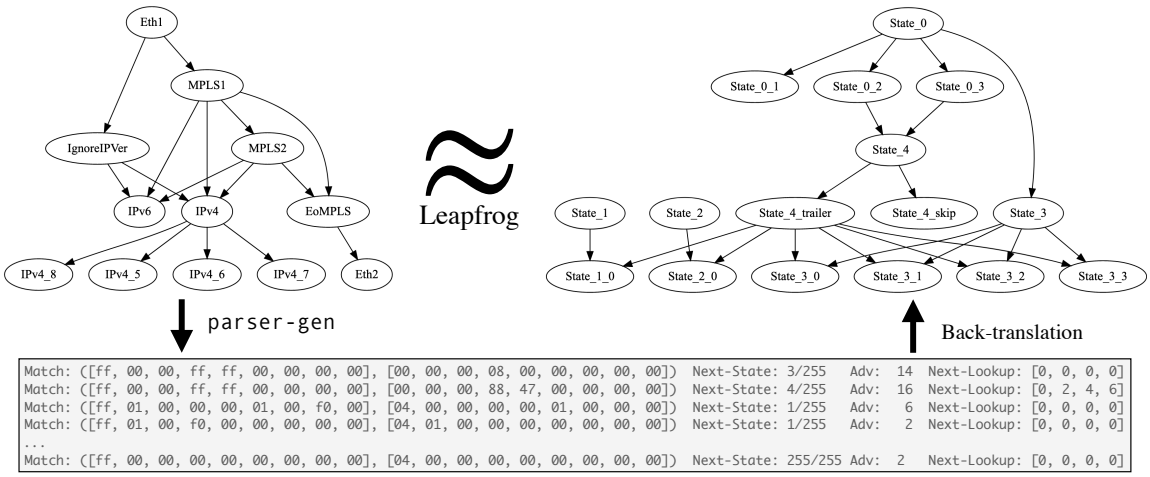


Figure 8. The Edge stack case study. The original parser (left) is compiled to a table (below, most entries elided), which we translate back into a parser (right) and prove equivalent to the original.

7.3 Discussion

Overall we find that Leapfrog can be applied to a diverse set of practical scenarios. In the rest of this section, we discuss some of our experiences using the tool, including its limitations and directions for future work.

Automation. In the early stages of this work, we derived and validated the relevant bisimulations without automated tactics. This turned out to be a significant proof burden—e.g., our manual equivalence proof for the State Rearrangement case study took two weeks of work. In contrast, the push-button Leapfrog proof takes only six seconds on a laptop. Although Leapfrog could be adapted to be more interactive, letting the user apply `Skip` or `Extend` and prove the required entailment, we believe that its power lies in the convenience offered by delegating goals to an SMT solver.

Leapfrog is particularly useful in situations where it is difficult to see whether two parsers are equivalent, such as in the translation validation experiment. While we spent a few days trying to prove the translation validation parsers equivalent on pen-and-paper, we were unsure that they were actually equivalent until the Leapfrog proof succeeded.

SMT Solver Performance. SMT solvers have unpredictable performance. We used Z3 for the queries in most of our benchmarks, but sometimes needed to switch to CVC4. Overall we found that all of the queries were solved in at most 10 seconds, with 99% taking at most 5 seconds. It was easy to switch between SMT solvers because we targeted a well-supported subset of the SMT-LIB query format (namely the theory of bitvectors).

Overall Performance. Like any verification tool, Leapfrog has limitations. Scaling to large parsers is challenging due to the combinatorial explosion of configurations. All of the smaller experiments (up to around 10 states) were interactive on stock hardware, finishing in $\leq$ 5 minutes and $\leq$ 16 GB of memory. Most took several minutes. For larger experiments, the larger state space lead to significantly higher memory demands. Coq needed 400 GB of RAM to verify the largest Applicability study (Datacenter) and ran out of memory on the Service Provider study. Although this is a lot, it's unsurprising because the concrete state space for the Applicability study would have around 2^{242} elements.

The optimizations discussed in Section 5 had a significant impact. Specifically, our smallest State Rearrangement benchmark went from 30 seconds and 1.7 GB of memory to 42 minutes and 36 GB of memory when leaps were disabled; it did not finish without reachable state pruning.

Future Work. One way to improve the scalability of Leapfrog in the future could be to investigate compositional reasoning techniques. Such techniques could facilitate divide-and-conquer strategies, allowing Leapfrog to be applied to larger parsers than our current implementation supports.

Another possibility is to vary the underlying algorithm. One could imagine a symbolic treatment of Hopcroft and Karp's algorithm [32], which approximates a suitable bisimulation from below, or Paige and Tarjan's *partition refinement* algorithm [45], which represents the current approximation of the largest bisimulation in terms of its equivalence classes. For the latter, one would need to estimate the number of configurations in a symbolically represented equivalence class to choose the next block to split.

The bulk of Leapfrog's memory usage is occupied by the proof object generated by our L_{tac} script. Alternatively, one could implement the same algorithm in Gallina, axiomatizing the decision procedure, and extract it to OCaml. While such an approach is likely to be more efficient, it would also undermine our goal of producing a proof object that is reusable in a larger verification effort.

P4As are an abstraction of P4 parsers. For one thing, they do not incorporate externs, which are architecture-specific

extensions that support, for instance, checksum algorithms or persistent state. In addition, P4 parsers support arrays (in the form of header stacks), subparser calls, and parser lookahead, all of which are not part of our definition of P4 automata. More work is necessary to see whether P4As can be extended to support or simulate these features.

In the future, we would like to use Leapfrog's equivalence checks to systematically perform translation validation on other networking stacks. For example, one could imagine writing a library of reference implementations for protocols defined in RFCs, and checking that real-world implementations conform to those standards.

8 Related work

Automata Equivalence Checking. Our algorithm is a variation on Moore's classical algorithm to decide all-pairs language equivalence in a DFA [40]. Moore's approach was later improved upon by *partition refinement* [31, 35, 45]. We deviate from these classical procedures in two key aspects.

First, instead of using concrete data structures we use symbolic ones. This idea goes back to Coudert et al. [18], and has since been widely applied [11–14, 24]. These algorithms use Binary Decision Diagrams (BDDs) as their symbolic representation. Other authors favored a logical representation, combined with decision procedures for the logic [26, 30]. Dehnert et al. [22] makes use of an SMT solver to decide questions about the logical representations.

Second, instead of maintaining a list of equivalence classes, we maintain a representation of an equivalence relation. The earliest instance of this we have been able to track down is due to Bouali and De Simone [13]. Mumme and Ciardo observed that such an approach is particularly beneficial when there tend to be a large number of equivalence classes [41].

Algorithms based on *bisimulation up to congruence* [9, 20] are similar in the sense that they mitigate state space explosion—in this case, as a result of determinization. They exploit the internal structure of the expanded state space to terminate early, something that inspired us propose the notion of a bisimulation with leaps.

Network Verification. The `p4v` verifier [37], the verifier of Neves et al. [43], and Aquila [52] are push-button verifiers for functional properties of P4 programs, including P4 parsers. They work by translating to a verification IR (either guarded command language [25] or simple C) and then analyzing the IR. None of these tools produce proofs, and their translations are not proved sound with respect to a reusable semantics of P4. Moreover, these tools verify *functional* specifications about a single P4 program. Our work is complementary because by contrast, our tool produces *relational* proofs grounded in a reusable Coq semantics for two P4 automata. Aquila includes a self-validation system for finding semantic bugs in the verifier. Defining our semantics in Coq allowed us to foundationally *prove* the absence of semantic bugs, so while Leapfrog does not need self-validation, it could be an oracle for validating other tools.

The Gauntlet translation validator checks program equivalence for P4 programs without parsers or externs. We see this work as complementary to Leapfrog, which focuses on parser equivalence. Outside the parser, P4 programs have loop-free control flow, complex data structures, and rich semantic actions. Inside the parser, P4 programs have loops, simpler data structures, and simpler semantic actions. Consequently, parser verification is concerned with control flow more than anything else, making it a different kind of verification problem than verification for the rest of a P4 program.

Automatic Foundational Verification. SpaceSearch [56] exposes a high-level solver interface to search large state spaces. In contrast, our solver interface is lower level, and our tool avoids extraction to produce a Coq certificate.

CreLLVM [36] instruments LLVM to produce translation validation proofs in a relational Hoare logic, resulting in a compact TCB and reusable Coq proof certificate. Leapfrog has a similar TCB, but completeness (Theorem 4.6) means it does not require proof hints.

The Narcissus [23] and EverParse [47] tools synthesize correct parsers and serializers from high level descriptions of packet formats using verified parser combinator libraries. Synthesis and equivalence are related but distinct problems, and our tool is complementary to synthesis tools. For instance, a P4 parser generated by a parser synthesizer like EverParse might be further optimized by a P4 compiler to run on hardware. Leapfrog could validate the results of compilation, preserving the guarantee offered by the synthesizer.

GPaco [33, 57] is a framework for modular coinductive reasoning in Coq, which supports "up-to" bisimilarity techniques. It is designed for interactive use and focuses on automating low-level proof steps. GPaco may be useful for generalizing our mechanized metatheory for leaps.

Acknowledgments

We thank Glen Gibb for help understanding and using his `parser-gen` framework. We received helpful feedback on the writing from Glen Gibb, James R. Wilcox, Bill Harris, and members of the Cornell programming languages group; we thank them for their feedback.

R. Doenges and N. Foster were supported in part by the National Science Foundation under grant FMiTF-1918396, DARPA under contract HR0011-20-C-0107, and gifts from Fujitsu, Google, InfoSys, and Keysight.

T. Kappé was partially supported by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101027412 (VERLAN). J. Sarracino and G. Morrisett were supported by DARPA contract HR0011-19-C-0073.

References

[1] Andrew W. Appel. 2011. Verified Software Toolchain. In *Proc. of European Symposium on Programming (ESOP)*. 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

[2] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. 2017. Position paper: the Science of Deep Specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017), 20160331. https://doi.org/10.1098/rsta.2016.0331

[3] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press.

[4] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 2011. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *Proc. of the International Conference on Certified Programs and Proofs (CPP)*. 135–150. https://doi.org/10.1007/978-3-642-25379-9_12

[5] Internet Assigned Numbers Authority. 2018. *Internet Protocol Version 4 (IPv4) Parameters*. https://www.iana.org/assignments/ip-parameters/ip-parameters.xhtml

[6] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proc. of Computer Aided Verification (CAV)*. 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

[7] Clark W. Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proc. of the 8th International Workshop on Satisfiability Modulo Theories (SMT, Vol. 13)*. 14–14.

[8] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. https://doi.org/10.1007/978-3-662-07964-5

[9] Filippo Bonchi and Damien Pous. 2013. Checking NFA Equivalence with Bisimulations up to Congruence. In *Proc. of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 457–468. https://doi.org/10.1145/2429069.2429124

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[11] Ahmed Bouajjani, Jean-Claude Fernandez, and Nicolas Halbwachs. 1991. Minimal Model Generation. In *Proc. of the 2nd International Workshop on Computer Aided Verification (CAV)*. 197–203. https://doi.org/10.1007/BFb0023733

[12] Ahmed Bouajjani, Jean-Claude Fernandez, Nicolas Halbwachs, and Pascal Raymond. 1992. Minimal State Graph Generation. *Science of Computer Programming* 18, 3 (1992), 247–269. https://doi.org/10.1016/0167-6423(92)90018-7

[13] Amar Bouali and Robert de Simone. 1992. Symbolic Bisimulation Minimisation. In *Proc. of the 4th International Workshop on Computer Aided Verification (CAV)*. 96–108. https://doi.org/10.1007/3-540-56496-9_9

[14] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and Lucius J. Hwang. 1992. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation* 98, 2 (1992), 142–170. https://doi.org/10.1016/0890-5401(92)90017-A

[15] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proc. of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 1027–1040. https://doi.org/10.1145/3314221.3314596

[16] The P4 Language Consortium. 2021. *P4 Language Specification, Version 1.2.2*. Available at https://p4.org/p4-spec/docs/P4-16-v1.2.2.html.

[17] The Coq Development Team. 2021. *The Coq Reference Manual, version 8.14*. Available electronically at http://coq.inria.fr/doc.

[18] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. 1989. Verification of Synchronous Sequential Machines Based on Symbolic Execution. In *Proc. of Automatic Verification Methods for Finite State Systems*. 365–373. https://doi.org/10.1007/3-540-52148-8_30

[19] Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *Journal of Automated Reasoning* 61, 1–4 (jun 2018), 423–453. https://doi.org/10.1007/s10817-018-9458-4

[20] Loris D'Antoni, Zachary Kincaid, and Fang Wang. 2018. A Symbolic Decision Procedure for Symbolic Alternating Finite Automata. In *Proc. of the 33rd International Conference on the Mathematical Foundations of Programming Semantics (MFPS)*. 79–99. https://doi.org/10.1016/j.entcs.2018.03.017

[21] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[22] Christian Dehnert, Joost-Pieter Katoen, and David Parker. 2013. SMT-Based Bisimulation Minimisation of Markov Models. In *Proc. of the 14th International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 28–47. https://doi.org/10.1007/978-3-642-35873-9_5

[23] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: Correct-by-Construction Derivation of Decoders and Encoders from Binary Formats. *Proceedings of the ACM on Programming Languages* 3, ICFP, Article 82 (July 2019), 29 pages. https://doi.org/10.1145/3341686

[24] Salem Derisavi. 2007. A Symbolic Algorithm for Optimal Markov Chain Lumping. In *Proc. of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 139–154. https://doi.org/10.1007/978-3-540-71209-1_13

[25] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (Aug. 1975), 453–457. https://doi.org/10.1145/360933.360975

[26] Yuan Feng, Yuxin Deng, and Mingsheng Ying. 2014. Symbolic Bisimulation for Quantum Processes. *ACM Transactions on Computational Logic* 15, 2, Article 14 (May 2014), 32 pages. https://doi.org/10.1145/2579818

[27] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. 2013. Design Principles for Packet Parsers. In *Proc. of the 9th ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*. 13–24. https://doi.org/10.1109/ANCS.2013.6665172

[28] Eduardo Giménez. 1994. Codifying Guarded Definitions with Recursive Schemes. In *Proc. of the International Workshop on Types for Proofs and Programs (TYPES)*. 39–59. https://doi.org/10.1007/3-540-60579-7_3

[29] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[30] Matthew Hennessy and Huimin Lin. 1995. Symbolic Bisimulations. *Theoretical Computer Science* 138, 2 (1995), 353–389. https://doi.org/10.1016/0304-3975(94)00172-F

[31] John Hopcroft. 1971. An $n \log n$ Algorithm for Minimizing States in a Finite Automaton. In *Proceedings of an International Symposium on the Theory of Machines and Computations*. Academic Press, 189–196. https://doi.org/10.1016/B978-0-12-417750-5.50022-1

[32] John Hopcroft and Richard M. Karp. 1971. *A Linear Algorithm for Testing Equivalence of Finite Automata*. Technical Report TR 71-114. Cornell University, Ithaca, NY. https://hdl.handle.net/1813/5958

[33] Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2013. The Power of Parameterization in Coinductive Proof. In *Proc. of the 40th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 193–206. https://doi.org/10.1145/2480359.2429093

[34] IEEE Computer Society. 2018. IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks. *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)* (2018), 1–1993. https://doi.org/10.1109/IEEESTD.2018.8403927

[35] Paris C. Kanellakis and Scott A. Smolka. 1983. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. In *Proc. of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*. 228–240. https://doi.org/10.1145/800221.806724

[36] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 2018. Crellvm: Verified Credible Compilation for LLVM. In *Proc. of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 631–645. https://doi.org/10.1145/3192366.3192377

[37] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4v: Practical Verification for Programmable Data Planes. In *Proc. of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 490–503. https://doi.org/10.1145/3230543.3230582

[38] Assia Mahboubi and Enrico Tassi. 2021. *Mathematical Components*. Zenodo. https://doi.org/10.5281/zenodo.4457887

[39] Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proc. of the 2nd International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS)*. 122–126. https://doi.org/10.1145/36177.36194

[40] Edward F. Moore. 2016. *Gedanken-Experiments on Sequential Machines*. Princeton University Press, 129–154. https://doi.org/doi:10.1515/9781400882618-006

[41] Malcolm Mumme and Gianfranco Ciardo. 2011. A Fully Symbolic Bisimulation Algorithm. In *Proc. of the 5th International Workshop on Reachability Problems (RP)*. 218–230. https://doi.org/10.1007/978-3-642-24288-5_19

[42] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proc. of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*. 83–94. https://doi.org/10.1145/349299.349314

[43] Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Verification of P4 Programs in Feasible Time Using Assertions. In *Proc. of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. 73–85. https://doi.org/10.1145/3281411.3281421

[44] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation* 9, 1 (2014), 53–58. https://doi.org/10.3233/sat190101

[45] Robert Paige and Robert Endre Tarjan. 1987. Three Partition Refinement Algorithms. *SIAM J. Comput.* 16, 6 (1987), 973–989. https://doi.org/10.1137/0216062

[46] Jon Postel. 1980. User Datagram Protocol. RFC 768. https://doi.org/10.17487/RFC0768

[47] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *28th USENIX Security Symposium (USENIX Security)*. 1465–1482. https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud

[48] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Michael E. Locasto. 2013. Security Applications of Formal Language Theory. *IEEE Systems Journal* 7, 3 (2013), 489–500. https://doi.org/10.1109/JSYST.2012.2222000

[49] Matthieu Sozeau. 2010. Equations: A Dependent Pattern-Matching Compiler. In *Interactive Theorem Proving (ITP)*, Matt Kaufmann and Lawrence C. Paulson (Eds.). 419–434. https://doi.org/10.1007/978-3-642-14052-5_29

[50] Matthieu Sozeau and Cyprien Mangin. 2019. Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq. *Proceedings of the ACM on Programming Languages* 3, ICFP, Article 86 (Jul 2019), 29 pages. https://doi.org/10.1145/3341690

[51] Thomas Streicher. 1993. Investigations into Intensional Type Theory. *Habilitiation Thesis, Ludwig Maximilian Universität* (1993). https://www2.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf

[52] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. 2021. Aquila: A Practically Usable Verification System for Production-Scale Programmable Data Planes. In *Proc. of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM)*. 17–32. https://doi.org/10.1145/3452296.3472937

[53] Jean-Baptiste Tristan and Xavier Leroy. 2009. Verified Validation of Lazy Code Motion. In *Proc. of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA, 316–326. https://doi.org/10.1145/1542476.1542512

[54] Arun Viswanathan, Eric C. Rosen, and Ross Callon. 2001. Multiprotocol Label Switching Architecture. RFC 3031. https://doi.org/10.17487/RFC3031

[55] Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. 2021. Certifying the Synthesis of Heap-Manipulating Programs. *Proceedings of the ACM on Programming Languages* 5, ICFP, Article 84 (Aug. 2021), 29 pages. https://doi.org/10.1145/3473589

[56] Konstantin Weitz, Steven Lyubomirsky, Stefan Heule, Emina Torlak, Michael D. Ernst, and Zachary Tatlock. 2017. SpaceSearch: A Library for Building and Verifying Solver-Aided Tools. *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 25 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110269

[57] Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. 2020. An Equational Theory for Weak Bisimulation via Generalized Parameterized Coinduction. In *Proc. of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. 71–84. https://doi.org/10.1145/3372885.3373813