Leapfrog: Certified Equivalence for Protocol Parsers

Tobias Kappé

ILLC, University of Amsterdam

June 28, 2022
Joint work with folks at Cornell

Ryan Doenges

John Sarracino

Nate Foster

Greg Morrisett
Packet parsing

01000111011011110010
00000110001001101001
01100111001000000111
00100110010101100100

(and metadata)
Packet parsing

01000111011011110010
00000110001001101001
01100111001000000111
00100110010101100100

(and metadata)

Success

Error
Packet parsing

header baby_ip {
    bit<8> src;
    bit<8> dst;
    bit<4> proto;
} (and metadata)
Packet parsing

header baby_ip {
    bit<8> src;
    bit<8> dst;
    bit<4> proto;
}  (and metadata)

Success

Error

01000111011011110010
00000110001001101001
01100111001000000111
00100110010101100100
(and metadata)
A horror story
A horror story
A horror story

parsed packet + metadata → control logic

output packet + metadata → flag set?

forward
no
recirculate yes
A horror story

parsed packet + metadata

control logic

output packet + metadata

flag set?

no
forward

no recirculate
A horror story

Control logic

Parsed packet + metadata

Forward

Flag set?

Yes
Recirculate

No
Output packet + metadata

Forward

Recirculate

Yes

No
State of the art

Verification frameworks for parsers exist:

- p4v (Liu et al. 2018)
- Aquila (Tian et al. 2021)
- Neves et al. 2018
State of the art

Verification frameworks for parsers exist:

- p4v (Liu et al. 2018)
- Aquila (Tian et al. 2021)
- Neves et al. 2018

Great works... but room for improvement:

- Only functional properties are verified.
State of the art

Verification frameworks for parsers exist:

- p4v (Liu et al. 2018)
- Aquila (Tian et al. 2021)
- Neves et al. 2018

Great works... but room for improvement:

- Only functional properties are verified.
- No reusable certificate is produced.
State of the art

Verification frameworks for parsers exist:

- p4v (Liu et al. 2018)
- Aquila (Tian et al. 2021)
- Neves et al. 2018

Great works... but room for improvement:

- Only functional properties are verified.
- No reusable certificate is produced.
- Rely on (trusted) verification to IR.
Comparing parsers
Comparing parsers
Comparing parsers
Contribution

- P4 automata: a syntax and semantics for protocol parsers.
Contribution

- P4 automata: a syntax and semantics for protocol parsers.
- Algorithm to check (hyperproperties like) language equivalence.
- Implementation of algorithm in Coq + SMT solver.
- Proof of soundness (in Coq) and completeness (on paper).
Contribution

- P4 automata: a syntax and semantics for protocol parsers.
- Algorithm to check (hyperproperties like) language equivalence.
- Implementation of algorithm in Coq + SMT solver.
Contribution

- P4 automata: a syntax and semantics for protocol parsers.
- Algorithm to check (hyperproperties like) language equivalence.
- Implementation of algorithm in Coq + SMT solver.
- Proof of soundness (in Coq) and completeness (on paper).
Parameters: states $Q$, headers $H$, header sizes $sz : H \rightarrow \mathbb{N}$. 
Semantics

\[ c = \langle q_1, s, \epsilon \rangle \]
Semantics

\[ c = \langle q_1, s, 0 \rangle \]
Semantics

\[ \langle q_1, s, 01 \rangle \]
Semantics

\[ c = \langle q_1, s, 01 \cdots \rangle \]
Semantics

\[ c = \langle q_1, s, 01 \cdots 0 \rangle \]
Semantics

\[ c = \langle q_1, s[01 \cdots 0/\text{mpls}], 01 \cdots 0 \rangle \]
$c = \langle q_2, s[01 \cdots 0/mpls], 01 \cdots 0 \rangle$
Semantics

$$c = \langle q_2, s[01\cdots 0/mpls], \epsilon \rangle$$
Formalization

Every P4 automaton gives rise to a DFA $\langle C, \delta, F \rangle$. 
Every P4 automaton gives rise to a DFA $\langle C, \delta, F \rangle$.

**Definition (Bisimulation)**

A binary relation $R$ is a *bisimulation* if for all $c_1 R c_2$,

1. $c_1 \in F$ if and only if $c_2 \in F$
2. $\delta(c_1, b) R \delta(c_2, b)$ for all $b$
Every P4 automaton gives rise to a DFA $\langle C, \delta, F \rangle$.

**Definition (Bisimulation)**
A binary relation $R$ is a *bisimulation* if for all $c_1 R c_2$,

1. $c_1 \in F$ if and only if $c_2 \in F$
2. $\delta(c_1, b) R \delta(c_2, b)$ for all $b$

**Definition (Equivalence)**
$P_1$ and $P_2$ are *equivalent* if there exists a bisimulation that relates their start states.
Problem: $|C| \geq 10^{37}$ for reference MPLS parser.

Two-pronged solution:

- Symbolic representation + SMT solving.
- Up-to techniques to skip buffering.
Symbolic representation

First-order logic with semantics $\llbracket \phi \rrbracket \subseteq C \times C$.

Examples

- $\phi = q_1^<$ means “the left state is $q_1$”
Symbolic representation

First-order logic with semantics $[[\phi]] \subseteq C \times C$.

Examples

- $\phi = q_1^<$ means “the left state is $q_1$”
- $\phi = 10^>$ means “the right buffer has 10 bits”
Symbolic representation

First-order logic with semantics $\llbracket \phi \rrbracket \subseteq C \times C$.

Examples

- $\phi = q_1^<$ means “the left state is $q_1$”
- $\phi = 10^>$ means “the right buffer has 10 bits”
- $\text{mpls}^<[24:24] = 1$ means “the 24th bit of the mpls header in the left store is 1”
Symbolic representation

First-order logic with semantics $\llbracket \phi \rrbracket \subseteq C \times C$.

Examples

- $\phi = q_1^<$ means “the left state is $q_1$”
- $\phi = 10^>$ means “the right buffer has 10 bits”
- $mpls^<[24 : 24] = 1$ means “the 24th bit of the $mpls$ header in the left store is 1”

Definition (Symbolic bisimulation)

If $\llbracket \phi \rrbracket$ is a bisimulation, then $\phi$ is a *symbolic bisimulation*. 
Equivalence checking — intuition

\[ \phi_0 = \text{accept}^< \iff \text{accept}^> \]
Equivalence checking — intuition

\[ \phi_0 = \text{accept} \leftarrow \leftrightarrow \rightarrow \text{accept} \]

\[ \phi_1 = \text{WP}(\phi_0) \]

\[ \phi_0 \land \phi_1 \]
Equivalence checking — intuition

\[ \phi_0 = \text{accept} \leftarrow \leftrightarrow \text{accept} \rightarrow \]

\[ \phi_1 = \text{WP}(\phi_0) \]

\[ \phi_2 = \text{WP}(\phi_1) \]

\[ \phi_0 \land \phi_1 \land \phi_2 \]
Equivalence checking — algorithm

\[
R \leftarrow \emptyset \\
T \leftarrow \{\text{accept} \iff \text{accept}\} \\
\text{while } T \neq \emptyset \text{ do} \\
\quad \text{pop } \psi \text{ from } T \\
\quad \text{if not } \land R \models \psi \text{ then} \\
\quad \quad R \leftarrow R \cup \{\psi\} \\
\quad \quad T \leftarrow T \cup \text{WP}(\psi) \\
\text{if } \phi \models \land R \text{ then} \\
\quad \text{return true} \\
\text{else} \\
\quad \text{return false}
\]
Equivalence checking — algorithm

\[ R \leftarrow \emptyset \]
\[ T \leftarrow \{ \text{accept}^< \iff \text{accept}^> \} \]

while \( T \neq \emptyset \) do

pop \( \psi \) from \( T \)

if not \( \forall R \models \psi \) then

\[ R \leftarrow R \cup \{ \psi \} \]
\[ T \leftarrow T \cup \text{WP}(\psi) \]

if \( \phi \models \forall R \) then

return true

else

return false

Loop termination: either

- \( \forall R \) shrinks; or
- \( \forall R \) stays the same, \( T \) shrinks.
Equivalence checking — algorithm

\[ R \leftarrow \emptyset \]
\[ T \leftarrow \{ \text{accept} < \iff \text{accept} > \} \]

while \( T \neq \emptyset \) do
  pop \( \psi \) from \( T \)
  if not \( \wedge R \models \psi \) then
    \[ R \leftarrow R \cup \{ \psi \} \]
    \[ T \leftarrow T \cup \text{WP}(\psi) \]

if \( \phi \models \wedge R \) then
  return true
else
  return false

Loop invariants:

▶ If \( c_1 \vdash V(R \cup T) \vdash c_2 \), then \( c_1 \in F \iff c_2 \in F \).

▶ If \( c_1 \vdash V(R \cup T) \vdash c_2 \), then \( \delta(c_1, b) \vdash V R \vdash \delta(c_2, b) \).

▶ If \( \phi \) is a symbolic bisimulation, then \( \phi \models V(R \cup T) \).

After the loop, \( V R \) is the weakest symbolic bisimulation.
Equivalence checking — algorithm

\[
R \leftarrow \emptyset \\
T \leftarrow \{\text{accept} \leftarrow \iff \text{accept} \}
\]

\textbf{while} \ T \neq \emptyset \ \textbf{do}

\hspace{1em} \text{pop } \psi \text{ from } T

\hspace{1em} \textbf{if} \ \text{not } \bigwedge R \models \psi \ \textbf{then}

\hspace{2em} R \leftarrow R \cup \{\psi\}

\hspace{2em} T \leftarrow T \cup \text{WP}(\psi)

\textbf{if } \phi \models \bigwedge R \ \textbf{then}

\hspace{1em} \text{return true}

\textbf{else}

\hspace{1em} \text{return false}

\textbf{Loop invariants:}

\hspace{1em} \text{If } c_1 \ \llbracket \bigwedge (R \cup T) \rrbracket c_2, \text{ then}

\hspace{2em} c_1 \in F \iff c_2 \in F.
Equivalence checking — algorithm

\[ R \leftarrow \emptyset \]
\[ T \leftarrow \{ \text{accept} \leftarrow \iff \text{accept} \rightarrow \} \]

while \( T \neq \emptyset \) do

<table>
<thead>
<tr>
<th>pop ( \psi ) from ( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>if not ( \bigwedge R \models \psi ) then</td>
</tr>
<tr>
<td>[ R \leftarrow R \cup { \psi } ]</td>
</tr>
<tr>
<td>[ T \leftarrow T \cup \mathit{WP}(\psi) ]</td>
</tr>
</tbody>
</table>

if \( \phi \vdash \bigwedge R \) then

| return true |

else

| return false |

Loop invariants:

- If \( c_1 \models \left[ \bigwedge (R \cup T) \right] c_2 \), then \( c_1 \in F \iff c_2 \in F \).
- If \( c_1 \models \left[ \bigwedge (R \cup T) \right] c_2 \),
Equivalence checking — algorithm

\[ R \leftarrow \emptyset \]
\[ T \leftarrow \{ \text{accept}^< \iff \text{accept}^> \} \]

while \( T \neq \emptyset \) do

\begin{align*}
\text{pop } \psi \text{ from } T \\
\text{if not } \land R \models \psi \text{ then} \\
\quad R \leftarrow R \cup \{ \psi \} \\
\quad T \leftarrow T \cup \text{WP}(\psi)
\end{align*}

if \( \phi \models \land R \) then

\[
\text{return true}
\]
else

\[
\text{return false}
\]

Loop invariants:

\begin{itemize}
\item If \( c_1 \left[ \land (R \cup T) \right] c_2 \), then \( c_1 \in F \iff c_2 \in F \).
\item If \( c_1 \left[ \land (R \cup T) \right] c_2 \),
\end{itemize}
Equivalence checking — algorithm

\[ R \leftarrow \emptyset \]
\[ T \leftarrow \{ \text{accept}< \iff \text{accept}> \} \]

**while** \( T \neq \emptyset \) **do**

\[ \text{pop} \ \psi \ \text{from} \ T \]

**if not** \( \exists R \models \psi \) **then**

\[ R \leftarrow R \cup \{ \psi \} \]
\[ T \leftarrow T \cup \text{WP}(\psi) \]

**if** \( \phi \models \exists R \) **then**

**return true**

**else**

**return false**

**Loop invariants:**

\[ \checkmark \text{If} \ c_1 \ \models \exists (R \cup T) \ c_2, \]
then
\[ c_1 \in F \iff c_2 \in F. \]

\[ \checkmark \text{If} \ c_1 \ \models \exists (R \cup T) \ c_2, \]
Optimizations — Pruning the bisimulation

$q_1$: \[ \text{extract(mpls, 32)} \]

$q_2$: \[ \text{mpls[23] == 1} \rightarrow \text{extract(udp, 64)} \]

$q_3$: \[ \text{extract(old, 32)} ; \text{extract(new, 32)} \]

$q_4$: \[ \text{old[23] == 0 \&\& new[23] == 1} \rightarrow \text{extract(udp, 64)} \]

$q_5$: \[ \text{old[23] == 1} \rightarrow \text{extract(tmp, 32)} ; \text{udp := new ++ tmp} \]
Optimizations — Pruning the bisimulation

Example (Unreachable pairs)
Left buffer 0, right buffer 13.
Optimizations — Pruning the bisimulation

Example (Buffering pairs)
Left buffer 7, right buffer 7.
Optimizations — Pruning the bisimulation

$q_1$: extract(mpls, 32)
$mpls[23]==0$

$q_2$: extract(udp, 64)
$mpls[23]==1$

$q_3$: extract(old, 32);
extract(new, 32)
old[23]==0 &&
new[23]==0

$q_4$: extract(udp, 64)
old[23]==0 &&
new[23]==1

$q_5$: extract(tmp, 32);
udp := new ++ tmp
old[23]==1
Optimizations — Pruning the bisimulation

$q_1$: extract(mpls, 32)

$q_2$: extract(udp, 64)

$mpls[23]==0$

$q_3$: extract(old, 32);
extract(new, 32)

$q_4$: extract(udp, 64)

$q_5$: extract(tmp, 32);
udp := new ++ tmp

$\langle 0, 0 \rangle$
Optimizations — Pruning the bisimulation

$q_1$: \text{extract(mpls, 32)}

$q_2$: \text{extract(udp, 64)}

$q_3$: \text{extract(old, 32); extract(new, 32)}

$q_4$: \text{old[23]==0 \\ new[23]==0}

$q_5$: \text{old[23]==1}

\text{extract(old, 32); extract(new, 32)}

\text{extract(tmp, 32); udp := new ++ tmp}
Optimizations — Pruning the bisimulation

$q_1$: extract(mpls, 32)  
$mpls[23]==0$

$q_2$: extract(udp, 64)  
$mpls[23]==1$

$q_3$: extract(old, 32); extract(new, 32)  
$old[23]==0 && new[23]==0$

$q_4$: extract(udp, 64)  
$old[23]==0 && new[23]==1$

$q_5$: extract(tmp, 32); udp := new ++ tmp  
$old[23]==1$
Optimizations — Correctness

Idea: compute *bisimulation with leaps* instead.

\[\#(c_1, c_2) = \text{“no. of bits until next state change”}\]

\(R\) is a bisimulation with leaps if for all \(c_1 \; R \; c_2\),

1. \(c_1 \in F\) if and only if \(c_2 \in F\)
2. \(\delta^*(c_1, w) \; R \; \delta^*(c_2, w)\) for all \(w \in \{0, 1\}^{\#(c_1, c_2)}\)

This is an up-to technique in disguise!

Note: requires adjusting implementation of WP.
Implementation — Side-stepping the termination checker
Implementation — Side-stepping the termination checker
Algorithm state as proof rules:

\[
\begin{align*}
\phi \models \bigwedge R & \quad \text{pre_bisim } \phi \ R \ \square \\
\bigwedge R \models \psi & \quad \text{pre_bisim } \phi \ R \ T \\
\bigwedge R \not\models \psi & \quad \text{pre_bisim } \phi \ (\psi :: T) \\
\bigwedge R \not\models \psi & \quad \text{pre_bisim } \phi \ (\psi :: T; WP(\psi))
\end{align*}
\]

**Lemma (Soundness)**

*If* \( \text{pre_bisim } \phi \ \square \ I \), *then all pairs in* \([\phi]\) *are bisimilar.*

Workflow: proof search for \( \text{pre_bisim} \), applying exactly one of these three rules.
Implementation — Talk to SMT solver
Implementation — Talk to SMT solver

In theory:

- If $T$ is empty, apply Done.
- If $\bigwedge R \models \psi$, apply Skip.
- If $\bigwedge R \not\models \psi$, apply Extend.

In practice:

- Massage entailment into fully quantified boolean formula.
- Custom plugin pretty-prints to SMT-LIB 2.0, asks solver.
  - If SAT, admit $\bigwedge R \models \psi$ and apply Skip.
  - If UNSAT, admit $\bigwedge R \not\models \psi$ and apply Extend.
Implementation — Talk to SMT solver

Existing tools (Armand et al. 2011; Czajka and Kaliszyk 2018):
▶ Encode goal in SMT, translate result to Coq proof.
▶ No support for fully quantified boolean formulas.
▶ Very little control over eventual SMT query.
Existing tools (Armand et al. 2011; Czajka and Kaliszyk 2018):

▶ Encode goal in SMT, translate result to Coq proof.
▶ No support for fully quantified boolean formulas.
▶ Very little control over eventual SMT query.

interp (R |= phi)
Implementation — Talk to SMT solver

Existing tools (Armand et al. 2011; Czajka and Kaliszyk 2018):

- Encode goal in SMT, translate result to Coq proof.
- No support for fully quantified boolean formulas.
- Very little control over eventual SMT query.

```coq
interp (R |= phi)
< vm_compute.
```
Implementation — Talk to SMT solver

Existing tools (Armand et al. 2011; Czajka and Kaliszyk 2018):

- Encode goal in SMT, translate result to Coq proof.
- No support for fully quantified boolean formulas.
- Very little control over eventual SMT query.

```
interp (R |= phi)
< vm_compute.
forall (x: bitvec n) (y: bitvec m), ...
```
Implementation — Talk to SMT solver

Existing tools (Armand et al. 2011; Czajka and Kaliszyk 2018):

▶ Encode goal in SMT, translate result to Coq proof.
▶ No support for fully quantified boolean formulas.
▶ Very little control over eventual SMT query.

interp (R \models \phi)
< vm_compute.
forall (x: bitvec n) (y: bitvec m), ...
< hammer.
Implementation — Talk to SMT solver

Existing tools (Armand et al. 2011; Czajka and Kaliszyk 2018):

▶ Encode goal in SMT, translate result to Coq proof.
▶ No support for fully quantified boolean formulas.
▶ Very little control over eventual SMT query.

interp (R |= phi)
< vm_compute.
forall (x: bitvec n) (y: bitvec m), ...
< hammer.
Tactic failure: cannot solve this goal.
Implementation — Talk to SMT solver

Our approach:

- Series of verified simplifications in Gallina.
- Eventual goal is translated almost literally into SMT query.
- No back-translation — have to trust solver (for now).
Our approach:

- Series of verified simplifications in Gallina.
- Eventual goal is translated almost literally into SMT query.
- No back-translation — have to trust solver (for now).

\[ \text{interp} \ (R \models \phi) \]
Implementation — Talk to SMT solver

Our approach:

- Series of verified simplifications in Gallina.
- Eventual goal is translated almost literally into SMT query.
- No back-translation — have to trust solver (for now).

```
interp (R |= phi)
< apply compile_formula.
```
Implementation — Talk to SMT solver

Our approach:
- Series of verified simplifications in Gallina.
- Eventual goal is translated almost literally into SMT query.
- No back-translation — have to trust solver (for now).

\[
\text{interp} \left( R \models \phi \right) \\
< \text{apply} \ \text{compile} \_\text{formula}. \\
\text{interp'} \left( \text{compile} \left( R \models \phi \right) \right)
\]
Implementation — Talk to SMT solver

Our approach:
- Series of verified simplifications in Gallina.
- Eventual goal is translated almost literally into SMT query.
- No back-translation — have to trust solver (for now).

```plaintext
interp (R |= phi) < apply compile_formula.
interp' (compile (R |= phi)) < cbn compile.
```
Our approach:

- Series of verified simplifications in Gallina.
- Eventual goal is translated almost literally into SMT query.
- No back-translation — have to trust solver (for now).

```
interp (R |= phi)
< apply compile_formula.
interp' (compile (R |= phi))
< cbn compile.
interp' (FForall (FExists (...)))
```
Our approach:

- Series of verified simplifications in Gallina.
- Eventual goal is translated almost literally into SMT query.
- No back-translation — have to trust solver (for now).

```
interp (R |= phi)
< apply compile_formula.
interp' (compile (R |= phi))
< cbn compile.
interp' (FForall (FExists (...)))
< verify_interp; admit.
```
Ceci n’est pas une diapo vide.
Implementation — Trusted computing base
Evaluation — Benchmarks

Automatically verifies common transformations:
- Speculative extraction / vectorization.
- Common prefix factorization
- General versus specialized TLV parsing.
- Early versus late filtering.

Extends to certain hyperproperties:
- Independence of initial header store.
- Correspondence between final stores.
Leapfrog verifies many interesting properties of protocol parsers.

<table>
<thead>
<tr>
<th>Name</th>
<th>States</th>
<th>Branched (bits)</th>
<th>Total (bits)</th>
<th>Time (min)</th>
<th>Mem. (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>St. rearrangement</td>
<td>5</td>
<td>8</td>
<td>136</td>
<td>0.12</td>
<td>0.66</td>
</tr>
<tr>
<td>Variable-length</td>
<td>30</td>
<td>64</td>
<td>632</td>
<td>953.42</td>
<td>405.64</td>
</tr>
<tr>
<td>Initialization</td>
<td>10</td>
<td>10</td>
<td>320</td>
<td>15.95</td>
<td>13.71</td>
</tr>
<tr>
<td>Speculation</td>
<td>5</td>
<td>2</td>
<td>160</td>
<td>4.12</td>
<td>3.16</td>
</tr>
<tr>
<td>Relational</td>
<td>6</td>
<td>64</td>
<td>1056</td>
<td>1.68</td>
<td></td>
</tr>
<tr>
<td>Filtering</td>
<td>6</td>
<td>64</td>
<td>1056</td>
<td>1.18</td>
<td></td>
</tr>
<tr>
<td>Edge</td>
<td>28</td>
<td>52</td>
<td>3184</td>
<td>528.38</td>
<td>251.26</td>
</tr>
<tr>
<td>Service Provider</td>
<td>22</td>
<td>50</td>
<td>2536</td>
<td>1244.5</td>
<td></td>
</tr>
<tr>
<td>Datacenter</td>
<td>30</td>
<td>242</td>
<td>2944</td>
<td>1387.95</td>
<td></td>
</tr>
<tr>
<td>Enterprise</td>
<td>22</td>
<td>176</td>
<td>2144</td>
<td>217.93</td>
<td></td>
</tr>
<tr>
<td>Tr. Validation</td>
<td>30</td>
<td>56</td>
<td>3148</td>
<td>746.2</td>
<td></td>
</tr>
</tbody>
</table>
Evaluation — Applicability study

parser-gen (Gibb et al. 2013) compiles parser to optimized implementation.

- Benchmarks: about 30 states each, huge store datastructure.
- Leapfrog can validate equivalence of input to output.
Lessons learned

- Finite automata can go the distance.
- Up-to techniques can be specialized.
- Programming in Coq is fun.

http://langsec.org/occupy/
Thank you!

Questions?

For your convenience:
► https://kap.pe/papers
► https://kap.pe/slides
References


M. C. Neves et al. (2018). “Verification of P4 programs in feasible time using assertions”. In: CoNEXT, pp. 73–85. DOI: 10.1145/3281411.3281421.