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

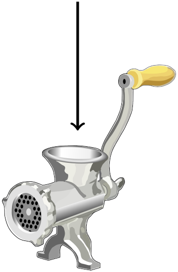
01000111011011110010

00000110001001101001

01100111001000000111

00100110010101100100

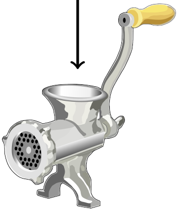
(and metadata)



Packet parsing

010001110110111110010
000001100010011010 01
011001110010000000111
00100110010101100100

(and metadata)



Success

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

Packet parsing

01000111011011110010

00000110001001101001

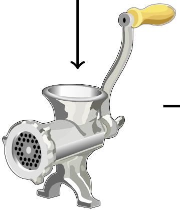
01100111001000000111

00100110010101100100

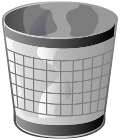
(and metadata)

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

Success

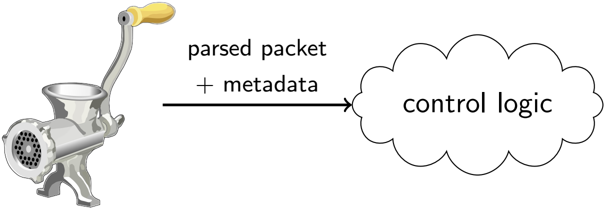


Error

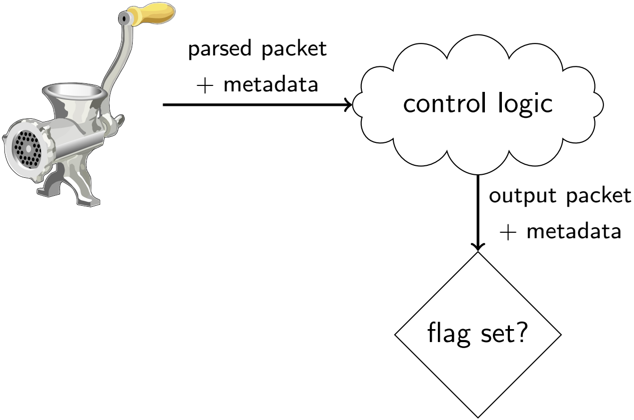


A horror story

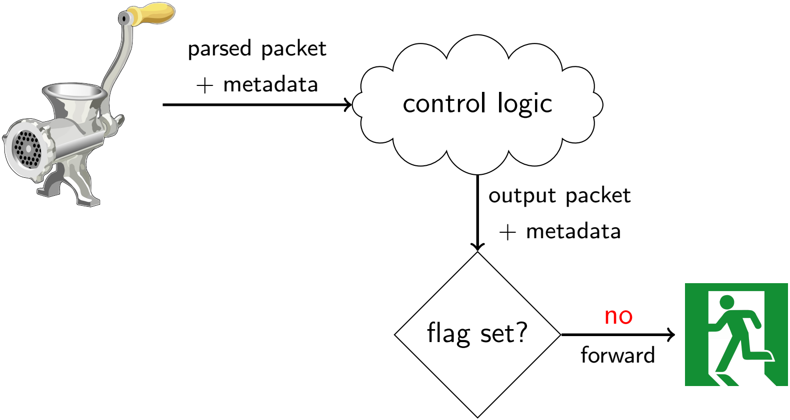
A horror story



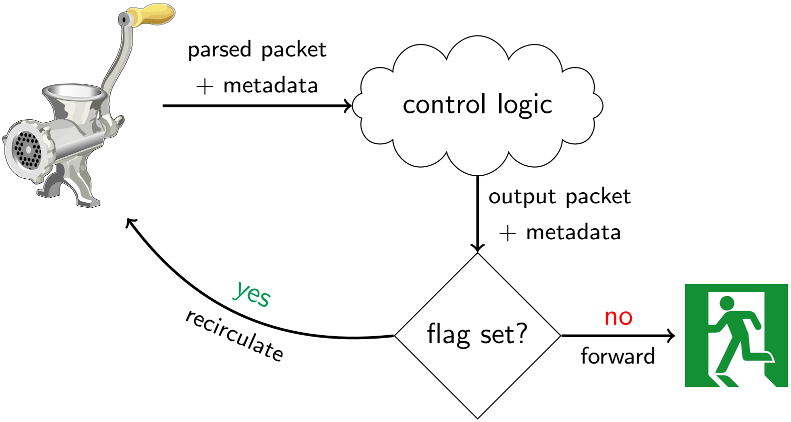
A horror story



A horror story



A horror story



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

$$\stackrel{?}{\approx}$$

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.

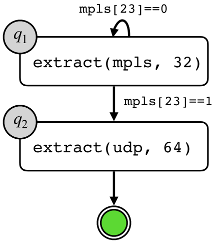
Contribution

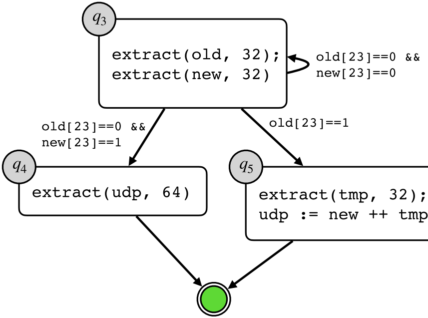
- P4 automata: a syntax and semantics for protocol parsers.
- Algorithm to check (hyperproperties like) language equivalence.
- Implementation of algorithm in Coq + SMT solver.

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).

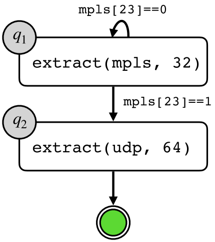
Running Example





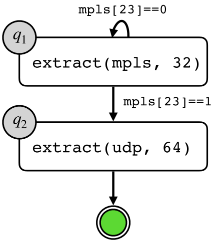
Parameters: states Q, headers H, header sizes $\text{sz} : H \to \mathbb{N}$.

Semantics



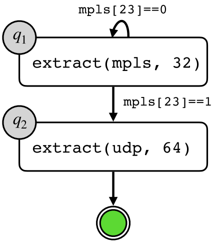
$$c = \langle q_1, s, \epsilon \rangle$$

Semantics



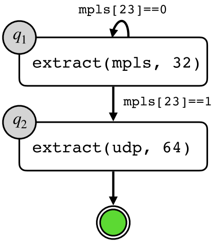
$$c = \langle q_1, s, 0 \rangle$$

Semantics



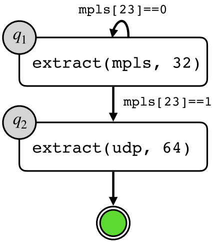
$$c = \langle q_1, s, 01 \rangle$$

Semantics



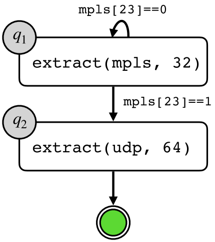
$$c = \langle q_1, s, 01 \cdots \rangle$$

Semantics



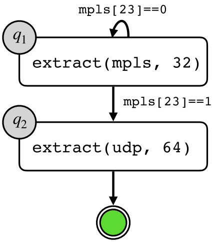
$$c = \langle q_1, s, 01 \cdots 0 \rangle$$

Semantics



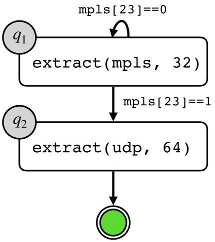
$$c = \langle q_1, s[01 \cdots 0/\texttt{mpls}], 01 \cdots 0 \rangle$$

Semantics



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

Semantics



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

Formalization

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

Formalization

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

Formalization

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.

Challenge

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 $[\![\phi]\!] \subseteq C \times C$.

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 $[\![\phi]\!] \subseteq C \times C$.

Examples

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

Symbolic representation

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

Examples

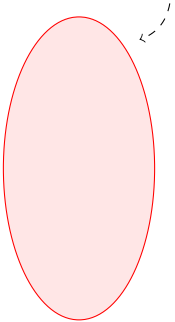
- $\phi = q_1^<$ means "the left state is q_1"
- $\phi = 10^>$ means "the right buffer has 10 bits"
- $\texttt{mpls}^<[24:24] = 1$ means "the 24th bit of the `mpls` header in the left store is 1"

Definition (Symbolic bisimulation)

If $[\![\phi]\!]$ is a bisimulation, then ϕ is a *symbolic bisimulation*.

Equivalence checking — intuition

$$\phi_0 = \mathsf{accept}^< \iff \mathsf{accept}^>$$

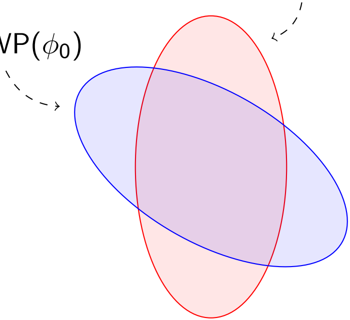


ϕ_0

Equivalence checking — intuition

$\phi_0 = \text{accept}^< \iff \text{accept}^>$

$\phi_1 = \text{WP}(\phi_0)$



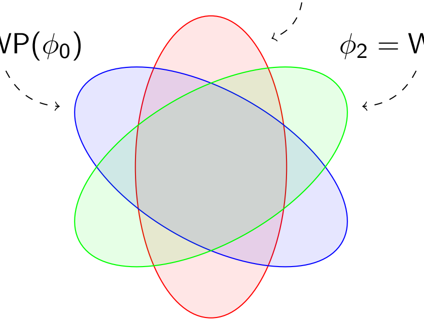
$\phi_0 \wedge \phi_1$

Equivalence checking — intuition

$\phi_0 = \text{accept}^{<} \iff \text{accept}^{>}$

$\phi_1 = \text{WP}(\phi_0)$

$\phi_2 = \text{WP}(\phi_1)$



$\phi_0 \wedge \phi_1 \wedge \phi_2$

Equivalence checking — algorithm

$R \leftarrow \emptyset$
$T \leftarrow \{\text{accept}^{<} \iff \text{accept}^{>}\}$
while $T \neq \emptyset$ **do**
    pop ψ from T
    if not $\bigwedge R \models \psi$ **then**
        $R \leftarrow R \cup \{\psi\}$
        $T \leftarrow T \cup \text{WP}(\psi)$
if $\phi \models \bigwedge R$ **then**
    return true
else
    return false

Equivalence checking — algorithm

$R \leftarrow \emptyset$
$T \leftarrow \{\text{accept}^< \iff \text{accept}^>\}$
while $T \neq \emptyset$ **do**
  pop ψ from T
  if not $\bigwedge R \vDash \psi$ **then**
    $R \leftarrow R \cup \{\psi\}$
    $T \leftarrow T \cup \text{WP}(\psi)$
if $\phi \vDash \bigwedge R$ **then**
  return true
else
  return false

Loop termination: either

- $[\![\bigwedge R]\!]$ shrinks; or
- $[\![\bigwedge R]\!]$ stays the same, T shrinks.

Equivalence checking — algorithm

$R \leftarrow \emptyset$
$T \leftarrow \{\text{accept}^< \iff \text{accept}^>\}$
while $T \neq \emptyset$ **do**
  pop ψ from T
  if not $\bigwedge R \models \psi$ **then**
    $R \leftarrow R \cup \{\psi\}$
    $T \leftarrow T \cup \mathrm{WP}(\psi)$

if $\phi \models \bigwedge R$ **then**
  return true
else
  return false

Loop invariants:

Equivalence checking — algorithm

$R \leftarrow \emptyset$
$T \leftarrow \{\text{accept}^{<} \iff \text{accept}^{>}\}$
while $T \neq \emptyset$ **do**
    pop ψ from T
    if not $\bigwedge R \vDash \psi$ **then**
        $R \leftarrow R \cup \{\psi\}$
        $T \leftarrow T \cup \mathrm{WP}(\psi)$
if $\phi \vDash \bigwedge R$ **then**
    return true
else
    return false

Loop invariants:

- If $c_1 \; [\![\bigwedge(R \cup T)]\!] \; c_2$, then $c_1 \in F \iff c_2 \in F$.

Equivalence checking — algorithm

$R \leftarrow \emptyset$
$T \leftarrow \{\text{accept}^< \iff \text{accept}^>\}$
while $T \neq \emptyset$ **do**
  pop ψ from T
  if not $\bigwedge R \vDash \psi$ **then**
    $R \leftarrow R \cup \{\psi\}$
    $T \leftarrow T \cup \mathrm{WP}(\psi)$
if $\phi \vDash \bigwedge R$ **then**
  return true
else
  return false

Loop invariants:

- If $c_1 \; [\![\bigwedge(R \cup T)]\!] \; c_2$, then $c_1 \in F \iff c_2 \in F$.
- If $c_1 \; [\![\bigwedge(R \cup T)]\!] \; c_2$,

Equivalence checking — algorithm

$R \leftarrow \emptyset$
$T \leftarrow \{\text{accept}^< \iff \text{accept}^>\}$
while $T \neq \emptyset$ **do**
  pop ψ from T
  if not $\bigwedge R \models \psi$ **then**
    $R \leftarrow R \cup \{\psi\}$
    $T \leftarrow T \cup \text{WP}(\psi)$

if $\phi \models \bigwedge R$ **then**
  return true
else
  return false

Loop invariants:

- If $c_1 \, [\![\bigwedge(R \cup T)]\!] \, c_2$, then $c_1 \in F \iff c_2 \in F$.
- If $c_1 \, [\![\bigwedge(R \cup T)]\!] \, c_2$,

Equivalence checking — algorithm

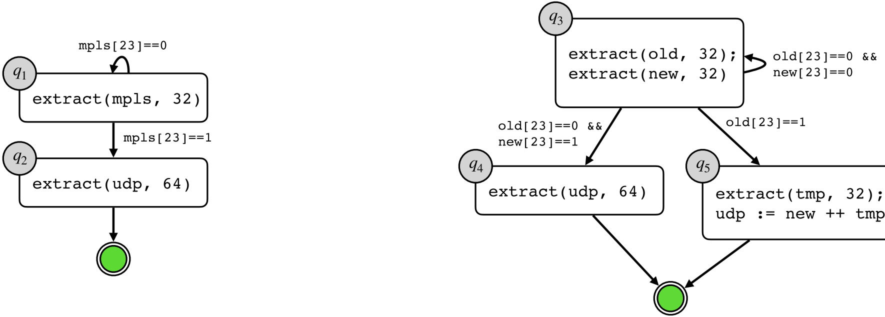
$R \leftarrow \emptyset$
$T \leftarrow \{\text{accept}^< \iff \text{accept}^>\}$
while $T \neq \emptyset$ **do**
  pop ψ from T
  if not $\bigwedge R \models \psi$ **then**
    $R \leftarrow R \cup \{\psi\}$
    $T \leftarrow T \cup \text{WP}(\psi)$

if $\phi \models \bigwedge R$ **then**
  return true
else
  return false

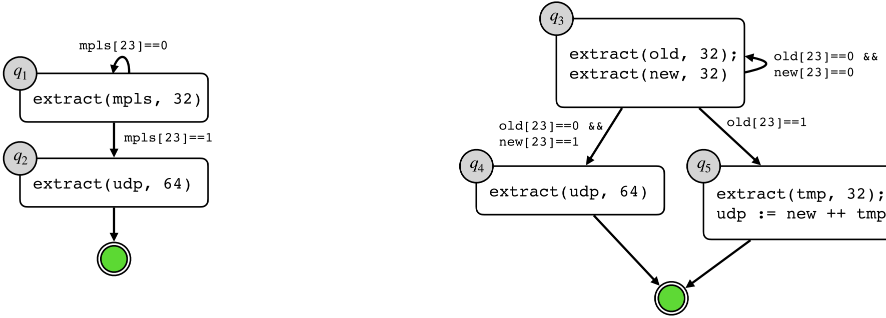
Loop invariants:

- If $c_1 \; [\![\bigwedge(R \cup T)]\!] \; c_2$, then $c_1 \in F \iff c_2 \in F$.
- If $c_1 \; [\![\bigwedge(R \cup T)]\!] \; c_2$,

Optimizations — Pruning the bisimulation



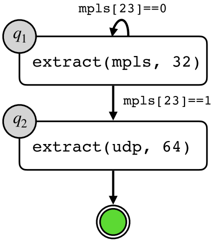
Optimizations — Pruning the bisimulation

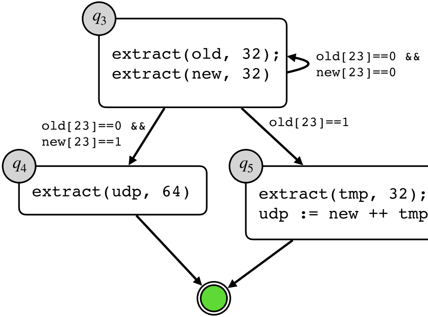


Example (Unreachable pairs)

Left buffer 0, right buffer 13.

Optimizations — Pruning the bisimulation

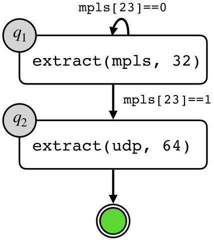


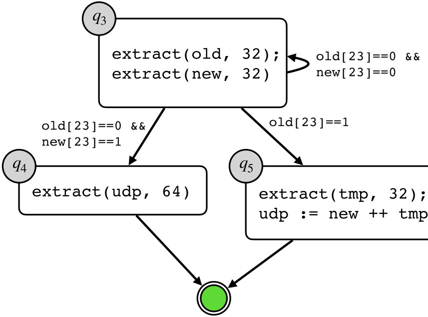


Example (Buffering pairs)

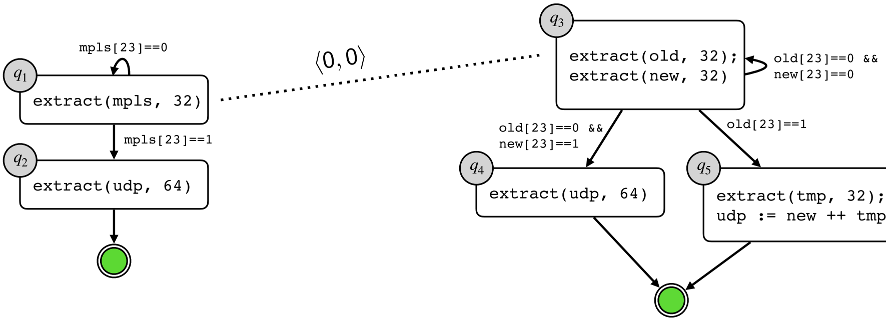
Left buffer 7, right buffer 7.

Optimizations — Pruning the bisimulation

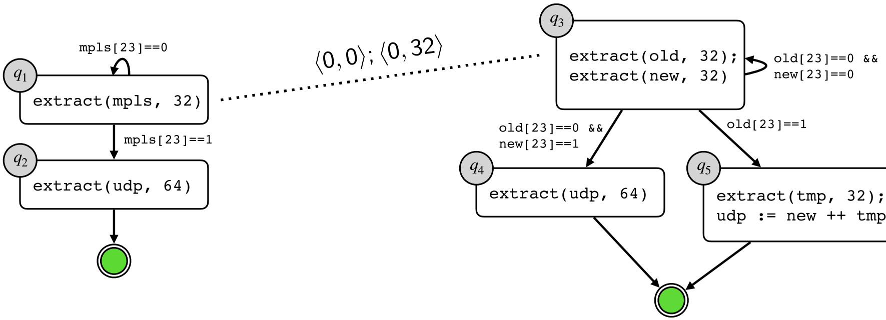




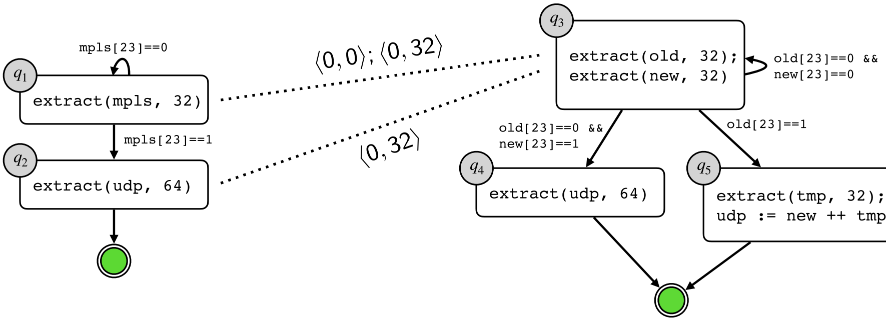
Optimizations — Pruning the bisimulation



Optimizations — Pruning the bisimulation



Optimizations — Pruning the bisimulation



Optimizations — Correctness

Idea: compute *bisimulation with leaps* instead.

$\sharp(c_1, c_2) =$ "no. of bits until next state change"

R is a bisimulation with leaps if for all $c_1 \mathrel{R} c_2$,

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

This is an up-to technique in disguise!

Note: requires adjusting implementation of WP.

Implementation

Implementation — Side-stepping the termination checker

Implementation — Side-stepping the termination checker

Algorithm state as proof rules:

$$\frac{\phi \vDash \bigwedge R}{\texttt{pre\_bisim}\ \phi\ R\ []}\ \textsc{close} \qquad \frac{\bigwedge R \vDash \psi \qquad \texttt{pre\_bisim}\ \phi\ R\ T}{\texttt{pre\_bisim}\ \phi\ R\ (\psi :: T)}\ \textsc{Skip}$$

$$\frac{\bigwedge R \nvDash \psi \qquad \texttt{pre\_bisim}\ \phi\ (\psi :: R)\ (T; \mathsf{WP}(\psi))}{\texttt{pre\_bisim}\ \phi\ R\ (\psi :: T)}\ \textsc{Extend}$$

Lemma (Soundness)

If `pre_bisim` ϕ `[]` I*, then all pairs in* $[\![\phi]\!]$ *are bisimilar.*

Workflow: proof search for `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 $\bigwedge R \vDash \psi$, apply `Skip`.
- If $\bigwedge R \nvDash \psi$, 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 $\bigwedge R \vDash \psi$ and apply `Skip`.
- If `UNSAT`, admit $\bigwedge R \nvDash \psi$ 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.

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)
```

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.
```

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 |= 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).

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)
```

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).

```
interp (R |= phi)
< apply compile_formula.
interp' (compile (R |= 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.
interp' (compile (R |= phi))
< cbn compile.
```

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.
interp' (compile (R |= phi))
< cbn compile.
interp' (FForall (FExists (...))
```

Implementation — Talk to SMT solver

Our approach:

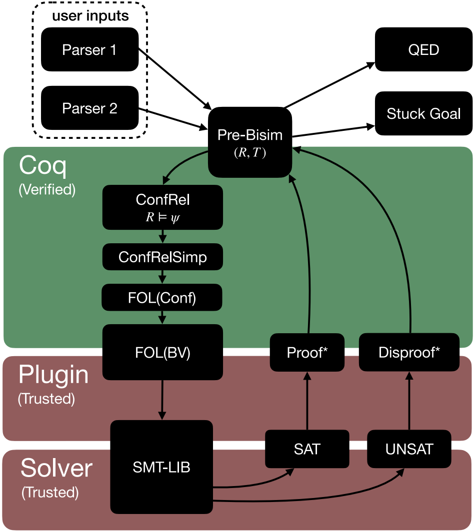
- Series of verified simplifications in Gallina.
- Eventual goal is translated almost literally into SMT query.
- No back-translation — have to trust solver (for now).

```
interp (R |= phi)
< apply compile_formula.
interp' (compile (R |= phi))
< cbn compile.
interp' (FForall (FExists (...))
< verify_interp; admit.
```

Implementation — Demo

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.

Evaluation — Benchmarks

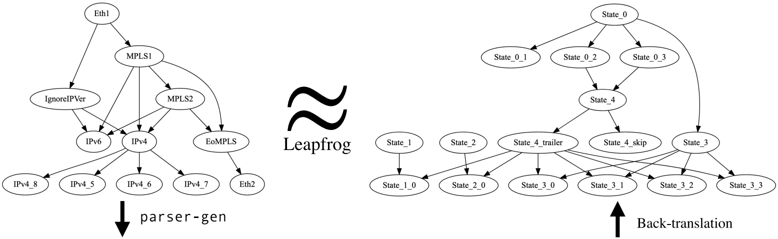
Leapfrog verifies many interesting properties of protocol parsers.

	Name	**States**	**Branched (bits)**	**Total (bits)**	**Time (min)**
Utility	St. rearrangement	5	8	136	0.12
	Variable-length	30	64	632	953.42
	Initialization	10	10	320	15.95
	Speculation	5	2	160	4.12
	Relational	6	64	1056	1.68
	Filtering	6	64	1056	1.18
Applicability	Edge	28	52	3184	528.38
	Service Provider	22	50	2536	1244.5
	Datacenter	30	242	2944	1387.95
	Enterprise	22	176	2144	217.93
	Tr. Validation	30	56	3148	746.2

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.




```
Match: ([ff, 00, 00, ff, ff, 00, 00, 00, 00], [00, 00, 00, 08, 00, 00, 00, 00, 00])  Next-State: 3/255   Adv:  14  Next-Lookup: [0, 0, 0, 0]
Match: ([ff, 00, 00, ff, ff, 00, 00, 00, 00], [00, 00, 00, 88, 47, 00, 00, 00, 00])  Next-State: 4/255   Adv:  16  Next-Lookup: [0, 2, 4, 6]
Match: ([ff, 01, 00, 00, 00, 01, 00, f0, 00], [04, 00, 00, 00, 00, 01, 00, 00, 00])  Next-State: 1/255   Adv:   6  Next-Lookup: [0, 0, 0, 0]
Match: ([ff, 01, 00, f0, 00, 00, 00, 00, 00], [04, 01, 00, 00, 00, 00, 00, 00, 00])  Next-State: 1/255   Adv:   2  Next-Lookup: [0, 0, 0, 0]
...
Match: ([ff, 00, 00, 00, 00, 00, 00, 00, 00], [04, 00, 00, 00, 00, 00, 00, 00, 00])  Next-State: 255/255 Adv:  2   Next-Lookup: [0, 0, 0, 0]
```

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. Armand et al. (2011). "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *CPP*, pp. 135–150. DOI: 10.1007/978-3-642-25379-9_12.
- L. Czajka and C. Kaliszyk (2018). "Hammer for Coq: Automation for Dependent Type Theory". In: *J. Autom. Reason.* 61.1-4, pp. 423–453. DOI: 10.1007/s10817-018-9458-4.
- G. Gibb et al. (2013). "Design principles for packet parsers". In: *ANCS*, pp. 13–24. DOI: 10.1109/ANCS.2013.6665172.
- J. Liu et al. (2018). "p4v: practical verification for programmable data planes". In: *SIGCOMM*, pp. 490–503. DOI: 10.1145/3230543.3230582.
- 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.
- B. Tian et al. (2021). "Aquila: a practically usable verification system for production-scale programmable data planes". In: *SIGCOMM*, pp. 17–32. DOI: 10.1145/3452296.3472937.