Automatically Discovering Abstractions for Network Verification

Devon Loehr
Networks are buggy, and that doesn’t surprise you.
Networks are buggy, and that doesn’t surprise you.
Networks are buggy, and that doesn’t surprise you.
Networks are buggy, and that doesn’t surprise you.
Network Verification

● Sample data plane verification tools:
  ○ Anteater (SIGCOMM ‘11)
  ○ NetPlumber (NSDI ‘13)

● Sample control plane verification tools:
  ○ rcc (NSDI ’05)
  ○ Batfish (NSDI ‘15)
  ○ ARC (SIGGCOMM ‘16)
  ○ NV (PLDI ‘20)
Outline

1. Overview of NV and its capabilities

2. Speeding up verification with *Hiding*

3. Wrap-up
NV: A network verification language

```plaintext
let nodes = 3
let edges = { 0-1; 1-2; }
```

Topology

```
0n -> 1n -> 2n
```
NV: A network verification language

```plaintext
let nodes = 3
let edges = { 0-1; 1-2; }

type attribute = (int, tnode)

let init node =
match node with
| 0n → (0, node)
| _ → (99, node)
```
NV: A network verification language

```
let nodes = 3
let edges = { 0-1; 1-2; }

type attribute = (int, tnode)

let init node =
  match node with
  | 0n → (0, node)
  | _ → (99, node)

let trans edge x =
  let (dist, origin) = x in
  (dist+1, origin)
```
NV: A network verification language

```
let nodes = 3
let edges = { 0-1; 1-2; }

type attribute = (int, tnode)

let init node =
  match node with
  | 0n → (0, node)
  | _ → (99, node)

let trans edge x =
  let (dist, origin) = x in
  (dist+1, origin)

let merge node x y =
  let (xdist, _) = x in
  let (ydist, _) = y in
  if xdist1 < ydist then x else y
```
NV: A network verification language

```plaintext
let nodes = 3
let edges = { 0-1; 1-2; }

type attribute = (int, tnode)

let init node = 
    match node with
    | 0n -> (0, node)
    | _  -> (99, node)

let trans edge x = 
    let (dist, origin) = x in
    (dist+1, origin)

let merge node x y = 
    let (xdist, _) = x in
    let (ydist, _) = y in
    if xdist1 < ydist then x else y
```
NV: A network verification language

```
let nodes = 3
let edges = { 0-1; 1-2; }

type attribute = (int, tnode)

let init node =
  match node with
  | 0n ⇒ (0, node)
  | _  ⇒ (99, node)

let trans edge x =
  let (dist, origin) = x in
  (dist+1, origin)

let merge node x y =
  let (xdist, _) = x in
  let (ydist, _) = y in
  if xdist < ydist then x else y
```
NV: A network verification language

```
let nodes = 3
let edges = { 0-1; 1-2; }

type attribute = (int, tnode)

let init node =
  match node with
  | 0n → (0, node)
  | _ → (99, node)

let trans edge x =
  let (dist, origin) = x in
  (dist+1, origin)

let merge node x y =
  let (xdist, _) = x in
  let (ydist, _) = y in
  if xdist < ydist then x else y
```
NV: A network verification language

```ml
let nodes = 3
let edges = { 0-1; 1-2; }

type attribute = (int, tnode)

let init node = 
  match node with 
  | 0n -> (0, node) 
  | _  -> (99, node)

let trans edge x = 
  let (dist, origin) = x in 
  (dist+1, origin)

let merge node x y = 
  let (xdist, _) = x in 
  let (ydist, _) = y in 
  if xdist < ydist then x else y

let sol = solution {init; trans; merge;}
```

Steady state
NV: A network verification language

Verifying properties

```ocaml
let nodes = 3
let edges = { 0-1; 1-2; }

type attribute = (int, tnode)

let init node =
  match node with
  | 0n -> (0, node)
  | _  -> (99, node)

let trans edge x =
  let (dist, origin) = x in
  (dist+1, origin)

let merge node x y =
  let (xdist, _) = x in
  let (ydist, _) = y in
  if xdist < ydist then x else y

let sol = solution {init; trans; merge;}

assert (forall n : tnode,
  let (_, origin) = sol[n] in
  origin = 0n)
```
Neighbors might send arbitrary messages

```ocaml
let nodes = 4
let edges = { 0-1; 1-2; 3-2; }

type attribute = (int, tnode)
symbolic hijack_attr : int

let init node =
  match node with
  [ 0n -> (0, node)
  | 3n -> (hijack_attr, node)
  | _ -> (99, node)

let trans edge x =
  let (dist, origin) = x in
  (dist+1, origin)

let merge node x y =
  let (xdist, _) = x in
  let (ydist, _) = y in
  if xdist < ydist then x else y

let sol = solution {init; trans; merge;}

assert (forall n : tnode,
  if n = 3n then true else
  let (_, origin) = sol[n] in
  origin = 0n)
```
Neighbors might send arbitrary messages

```ocaml
let nodes = 4
let edges = { 0-1; 1-2; 3-2; }

type attribute = (int, tnode)
symbolic hijack_attr : int

let init node =
  match node with
  | 0n  -> (0, node)
  | 3n  -> (hijack_attr, node)
  | _   ->(99, node)

let trans edge x =
  let (dist, origin) = x in
  (dist+1, origin)

let merge node x y =
  let (xdist, _) = x in
  let (ydist, _) = y in
  if xdist < ydist then x else y

let sol = solution {init; trans; merge;}

assert (forall n : tnode,
  if n = 3n then true else
  let (_, origin) = sol[n] in
  origin = 0n)
```

Might hijack traffic between node 2 and node 0!
Solution: Use an SMT Solver

- Find a *steady state* for the network, where no node prefers any of its neighbors’ attributes to its own
- Simulator computes a steady state, but there may be multiple
- SMT solver checks if the assertion may be violated by *any* steady state
- Requires heavy simplification to translate NV into SMT constraints
Transformation pipeline (for SMT)

User Program → Type Inference → Record Unrolling → Inlining

Tuple Flattening ← Option Unboxing ← Edge Unboxing ← Map Unrolling

Slicing (optional) ← Unit Unboxing ← SMT Encoding ← Z3

BLUE boxes are compositional NV-to-NV transformations
Most blue boxes use a centralized mechanism for specifying transformations
Map Unrolling has been particularly challenging
Maps in NV

- Maps (or dictionaries) are commonly used in networking
- NV maps are \textit{total}

```
1  type tmap = dict[int, bool]
2  (* Total map from int to bool *)
3  let m : tmap = CreateDict false in
4  let m = m[3 := true] in
5  let m = m[7 := true] in
6  let x = m[3] in
7  let y = m[4] in
8  x && y
```
Encoding Map Operations

- Some dictionary operations require quantifiers to encode into SMT

```plaintext
let m' = map f m in ...

forall k, m'[k] = f (m[k])
```
Encoding Map Operations

- Some dictionary operations require quantifiers to encode into SMT
- In general, quantifiers in SMT are not complete

```
let m' = map f m in ...
```

```
forall k, m'[k] = f (m[k])
```
Static keys

- Observation: In real networks, map keys are usually known in advance
  - E.g. Routers originate a fixed, known set of destinations
  - We see expression like $m[3]$, never $m[...complicated\ computation...]$
Static keys

- Observation: In real networks, map keys are usually known in advance
  - E.g. Routers originate a fixed, known set of destinations
  - We see expression like \texttt{m[3]}, never \texttt{m[...complicated computation...]} \[26\]
- Hence we can figure out which keys will be relevant \textit{statically} by simply scanning the program!

```plaintext
1 type tmap = dict[int, bool]
2 (* Total map from int to bool *)
3 let m : tmap = CreateDict false in
4 let m = m[3] := true in
5 let m = m[7] := true in
6 let x = m[3] in
7 let y = m[4] in
8 x && y
```

Only keys used are 3, 4, 7!
Map Unrolling

- Finitize maps by transforming them into tuples, with one element for each key that is used
- Require all map keys in NV programs to be literals
- Doesn’t hinder translation of configs in practice
Overview of NV

- NV is a programming language in which programs are descriptions of networks.
- Networks may be verified either with a simulator or an SMT solver.
- We use a pipeline of compositional transformations to translate NV programs into SMT constraints.
- We encode dictionaries as tuples using Map Unrolling.
Problem: SMT analysis doesn’t scale well
Networks contain a lot of irrelevant information

- Observation: Network operators may not utilize every feature of every network protocol

- Observation: Not all features that are used may be relevant to the property we’re verifying
  - E.g. checking the existence of a path may not require any information about that path’s length

- Idea: Speed up verification by removing irrelevant information from the network
Many SMT constraints may be irrelevant

- Observation: SMT solving is worst-case exponential in the number of variables (for us, this is roughly equal to the number of constraints)

- Observation: Most SMT constraints simply describe the stable state of the network, and are rarely UNSAT. Only a few represent the assertion.

- Idea: hide all the constraints except the assertion, and iteratively unhide them only when they become relevant (CEGAR-style).
Hiding -- Initial Program

1 (* No constraints on x0, y0, z0 *)

2

3 x1 = x0

4 y1 = y0 && x0

5 z1 = z0 || (y0 && x0)

6

7 x2 = x1

8 y2 = y1 && x1

9 z2 = z1 || (y1 && x1)

10

11 !(y1 && y2 || !y1 && !y2)
Hiding -- Iteration 1

Hidden Program

```plaintext
(* No constraints on x0, y0, z0 *)

1 (y1 && y2 || !y1 && !y2)
```

Full Program

```plaintext
(* No constraints on x0, y0, z0 *)

1 (y1 && y2 || !y1 && !y2)

2 x1 = x0

3 y1 = y0 && x0

4 z1 = z0 || (y0 && x0)

5 x2 = x1

6 y2 = y1 && x1

7 z2 = z1 || (y1 && x1)

8 !(y1 && y2 || !y1 && !y2)
```
Hiding -- Iteration 1

Hidden Program

```plaintext
(* No constraints on x0, y0, z0 *)

10 !(y1 && y2 || !y1 && !y2)
```

SAT:
y1 = true, y2 = false

Full Program

```plaintext
(* No constraints on x0, y0, z0 *)

3 x1 = x0
4 y1 = y0 && x0
5 z1 = z0 || (y0 && x0)
6
7 x2 = x1
8 y2 = y1 && x1
9 z2 = z1 || (y1 && x1)
10
11 !(y1 && y2 || !y1 && !y2)
```
Hiding -- Iteration 1

Hidden Program

```
1 (* No constraints on x0, y0, z0 *)
2
3
4
5
6
7
8
9
10
11 !(y1 && y2 || !y1 && !y2)
```

SAT:
y1 = true, y2 = false

Full Program

```
1 (* No constraints on x0, y0, z0 *)
2
3 x1 = x0
4 y1 = y0 && x0
5 z1 = z0 || (y0 && x0)
6
7 x2 = x1
8 y2 = y1 && x1
9 z2 = z1 || (y1 && x1)
10
11 !(y1 && y2 || !y1 && !y2)
12
13 y1 = true
14 y2 = false
```
Hiding -- Iteration 1

Hidden Program

```plaintext
(*) No constraints on x0, y0, z0 *)

!(y1 && y2 || !y1 && !y2)
```

SAT:
y1 = true, y2 = false

Full Program

```plaintext
(*) No constraints on x0, y0, z0 *)

x1 = x0
y1 = y0 && x0
z1 = z0 || (y0 && x0)
x2 = x1
y2 = y1 && x1
z2 = z1 || (y1 && x1)
!(y1 && y2 || !y1 && !y2)
y1 = true
y2 = false
```

UNSAT (Need info on: y1, y2)
Hiding -- Iteration 2

Hidden Program

```
1 (* No constraints on x0, y0, z0 *)
2
3 y1 = y0 && x0
4
5 y2 = y1 && x1
6
7 !(y1 && y2 || !y1 && !y2)
```

Full Program

```
1 (* No constraints on x0, y0, z0 *)
2
3 x1 = x0
4 y1 = y0 && x0
5 z1 = z0 || (y0 && x0)
6
7 x2 = x1
8 y2 = y1 && x1
9 z2 = z1 || (y1 && x1)
10
11 !(y1 && y2 || !y1 && !y2)
```
Hiding -- Iteration 2

**Hidden Program**

```plaintext
(* No constraints on x0, y0, z0 *)

y1 = y0 && x0

y2 = y1 && x1

!(y1 && y2 || !y1 && !y2)
```

**Full Program**

```plaintext
(* No constraints on x0, y0, z0 *)

x1 = x0

y1 = y0 && x0

z1 = z0 || (y0 && x0)

x2 = x1

y2 = y1 && x1

z2 = z1 || (y1 && x1)

!(y1 && y2 || !y1 && !y2)
```

**SAT:**

\[
y_0 = \text{true}, \quad y_1 = \text{true}, \quad y_2 = \text{false} \\
x_0 = \text{true}, \quad x_1 = \text{false}
\]
Hiding -- Iteration 2

Hidden Program

```
1  (* No constraints on x0, y0, z0 *)
2
3  y1 = y0 && x0
4
5  y2 = y1 && x1
6
7  !(y1 && y2 || !y1 && !y2)
```

SAT:
y0 = true, y1 = true, y2 = false
x0 = true, x1 = false

Full Program

```
1  (* No constraints on x0, y0, z0 *)
2
3  x1 = x0
4  y1 = y0 && x0
5  z1 = z0 || (y0 && x0)
6
7  x2 = x1
8  y2 = y1 && x1
9  z2 = z1 || (y1 && x1)
10
11  !(y1 && y2 || !y1 && !y2)
12
13  y0 = y1 = x0 = true
14  x1 = y2 = false
```
Hiding -- Iteration 2

Hidden Program

```plaintext
(* No constraints on x0, y0, z0 *)

y1 = y0 && x0

y2 = y1 && x1

!(y1 && y2 || !y1 && !y2)
```

SAT:

- y0 = true, y1 = true, y2 = false
- x0 = true, x1 = false

Full Program

```plaintext
(* No constraints on x0, y0, z0 *)

x1 = x0

y1 = y0 && x0

z1 = z0 || (y0 && x0)

x2 = x1

y2 = y1 && x1

z2 = z1 || (y1 && x1)

!(y1 && y2 || !y1 && !y2)

y0 = y1 = x0 = true

x1 = y2 = false
```

UNSAT (Need info on: x0, x1)
Hiding -- Iteration 3

Hidden Program

```plaintext
1  (* No constraints on x0, y0, z0 *)
2  x1 = x0
3  y1 = y0 && x0
4
5  y2 = y1 && x1
6  ! (y1 && y2 || !y1 && !y2)
```

Full Program

```plaintext
1  (* No constraints on x0, y0, z0 *)
2
3  x1 = x0
4  y1 = y0 && x0
5  z1 = z0 || (y0 && x0)
6
7  x2 = x1
8  y2 = y1 && x1
9  z2 = z1 || (y1 && x1)
10
11  ! (y1 && y2 || !y1 && !y2)
```
Hiding -- Iteration 3

Hidden Program

```
1 (* No constraints on x0, y0, z0 *)
2
3 x1 = x0
4 y1 = y0 && x0
5
6 y2 = y1 && x1
7
8 !(y1 && y2 || !y1 && !y2)
```

Full Program

```
1 (* No constraints on x0, y0, z0 *)
2
3 x1 = x0
4 y1 = y0 && x0
5 z1 = z0 || (y0 && x0)
6
7 x2 = x1
8 y2 = y1 && x1
9 z2 = z1 || (y1 && x1)
10
11 !(y1 && y2 || !y1 && !y2)
```

UNSAT
Hiding -- Iteration 3

Hidden Program

```
(* No constraints on x0, y0, z0 *)

x1 = x0
y1 = y0 && x0

y2 = y1 && x1

!(y1 && y2 || !y1 && !y2)
```

UNSAT

Full Program

```
(* No constraints on x0, y0, z0 *)

x1 = x0
y1 = y0 && x0
z1 = z0 || (y0 && x0)

x2 = x1
y2 = y1 && x1
z2 = z1 || (y1 && x1)

!(y1 && y2 || !y1 && !y2)
```

Must also be UNSAT!
Hiding - Algorithm Sketch

1. Create two copies of the SMT program -- one full, one with some constraints hidden

2. Check satisfiability for the hidden program
   a. If it’s UNSAT, then so is the full program, so return.
   b. If it’s SAT, test the model on the full program

3. If the model extends to the full program, then it is also SAT, so return the full model

4. Otherwise, refine the hidden program by unhiding some constraints
   a. Add constraints for all variables that appear in the UNSAT core

5. Go to step 2
Hiding - Algorithm Sketch

1. Create two copies of the SMT program -- one full, one with some constraints hidden
2. Check satisfiability for the hidden program
   a. If it’s UNSAT, then so is the full program, so return.
   b. If it’s SAT, test the model on the full program
3. If the model extends to the full program, then it is also SAT, so return the full model
4. Otherwise, refine the hidden program by unhiding some constraints
   a. Add constraints for all variables that appear in the UNSAT core
5. Go to step 2

Guaranteed to terminate after a finite number of iterations, with the same result as the full program!
Experimental Results

<table>
<thead>
<tr>
<th>File</th>
<th>Nodes</th>
<th>Control (seconds)</th>
<th>Hiding (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4</td>
<td>20</td>
<td>0.35</td>
<td>19.9</td>
</tr>
<tr>
<td>fat4pol</td>
<td>20</td>
<td>1.1</td>
<td>94</td>
</tr>
<tr>
<td>sp8</td>
<td>80</td>
<td>3.8</td>
<td>9358</td>
</tr>
</tbody>
</table>
### Experimental Results

<table>
<thead>
<tr>
<th>File</th>
<th>Nodes</th>
<th>Control (seconds)</th>
<th>Hiding (seconds)</th>
<th># Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4</td>
<td>20</td>
<td>0.35</td>
<td>19.9</td>
<td>112</td>
</tr>
<tr>
<td>fat4pol</td>
<td>20</td>
<td>1.1</td>
<td>94</td>
<td>197</td>
</tr>
<tr>
<td>sp8</td>
<td>80</td>
<td>3.8</td>
<td>9358</td>
<td>435</td>
</tr>
</tbody>
</table>
### Experimental Results

<table>
<thead>
<tr>
<th>File</th>
<th>Nodes</th>
<th>Control (seconds)</th>
<th>Hiding (seconds)</th>
<th># Iterations</th>
<th>% Hidden</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4</td>
<td>20</td>
<td>0.35</td>
<td>19.9</td>
<td>112</td>
<td>0.0%</td>
</tr>
<tr>
<td>fat4pol</td>
<td>20</td>
<td>1.1</td>
<td>94</td>
<td>197</td>
<td>32.4%</td>
</tr>
<tr>
<td>sp8</td>
<td>80</td>
<td>3.8</td>
<td>9358</td>
<td>435</td>
<td>16.8%</td>
</tr>
</tbody>
</table>
### Experimental Results

<table>
<thead>
<tr>
<th>File</th>
<th>Nodes</th>
<th>Control (seconds)</th>
<th>Hiding (seconds)</th>
<th># Iterations</th>
<th>% Hidden</th>
<th>Last Iteration (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp4</td>
<td>20</td>
<td>0.35</td>
<td>19.9</td>
<td>112</td>
<td>0.0%</td>
<td>0.39</td>
</tr>
<tr>
<td>fat4pol</td>
<td>20</td>
<td>1.1</td>
<td>94</td>
<td>197</td>
<td>32.4%</td>
<td>0.49</td>
</tr>
<tr>
<td>sp8</td>
<td>80</td>
<td>3.8</td>
<td>9358</td>
<td>435</td>
<td>16.8%</td>
<td>1.5</td>
</tr>
</tbody>
</table>
Future Work

- Heuristics for unhiding variables
- DSL for specifying which variables should start hidden
Related Work on Hiding

- Hiding-style techniques were first proposed by Robert Kurshnan in 1994
  - Maintains relationships between variable using a variable dependency graph
  - It was also inspiration for the original CEGAR paper in 2000

- In 2007, Wang, Kim and Gupta proposed Hybrid CEGAR, which combines hiding with predicate abstraction

- The Corral verifier for Boogie (2011) practices a similar technique by only inlining a few functions, then adding more as needed.
Comparison of Hiding to Other Abstraction Techniques

● CEGAR algorithm
  ○ Generates possibly-spurious counterexamples, then refines its abstraction

● Guaranteed to terminate

● No false positives or negatives

● Subset of existing constraints
  ○ Can only use relationships that exist in the original constraints
  ○ Can’t replace data structures or relationships with more abstract versions
  ○ Could be combined with such techniques, however
In Summary...

- I presented my work on developing NV, a programming language for network verification
- I worked on a pipeline of simplifications for encoding NV into SMT constraints
- I wrote an algorithm called Hiding which aims to speed up verification by removing irrelevant information
- Initial tests for hiding indicate that it can discover effective abstractions, but takes too long to do so
- Future work involves heuristics and hints to make hiding converge faster
Questions?