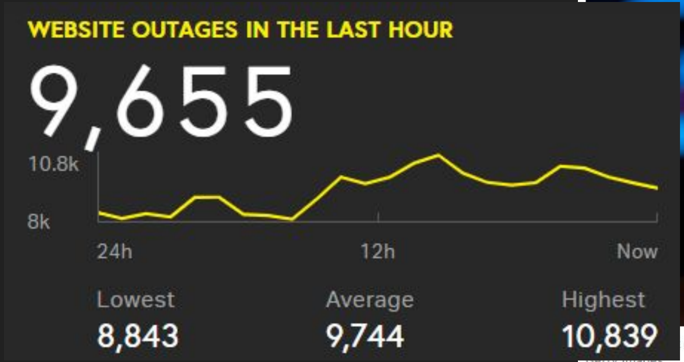
Automatically Discovering Abstractions for Network Verification

Devon Loehr

Networks are buggy, and that doesn't surprise you

Google Compute Engine	Historic
GCE20003	Began 26 March 2020, lasting 11 hours 7 minutes
GCE20002	Began 26 March 2020, lasting 54 minutes
GCE20001	Began 28 January 2020, lasting 28 hours 7 minutes
GCE19012	Began 07 December 2019, lasting 3 hours 6 minutes
GCE19011	Began 03 December 2019, lasting 21 minutes
GCE19010	Began 11 November 2019, lasting 1 hour 43 minutes
GCE19008	Began 31 October 2019, lasting 45 hours 30 minutes
GCE19007	Began 22 October 2019, lasting 3 hours 43 minutes
GCE19006	Began 24 September 2019, lasting 5 hours 8 minutes
GCE19005	Began 05 September 2019, lasting 13 hours 17 minutes
GCE19003	Began 02 June 2019, lasting 3 hours 54 minutes
GCE19002	Began 15 May 2019, lasting 5 hours 1 minute
GCE19001	Began 11 May 2019, lasting 2 hours 11 minutes



Google outage hits Gmail, Snapchat and Nest

Company investigating after Cloud Platform problem causes email delivery failures

expected to solve the problem with Gmail in the near future. Photograph: NurPhoto/NurPhoto via Getty Images

Networks are buggy, and that doesn't surprise you

Google Compute Engine Historic

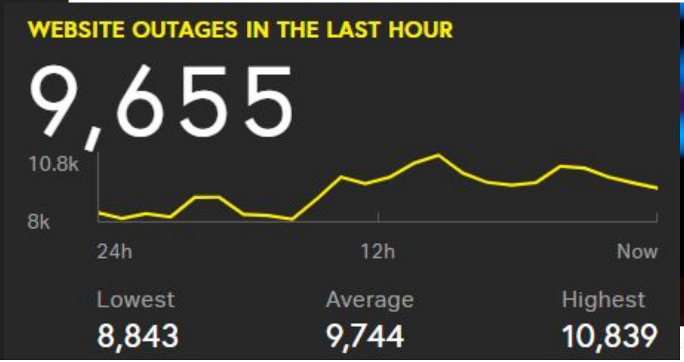
GCE20003	Began 26 March 2020, lasting 11 hours 7 minutes
GCE20002	Began 26 March 2020, lasting 54 minutes
GCE20001	Began 28 January 2020, lasting 28 hours 7 minutes
GCE19012	Began 07 December 2019, lasting 3 hours 6 minutes
GCE19011	Began 03 December 2019, lasting 21 minutes
GCE19010	Began 11 November 2019, lasting 1 hour 43 minutes
GCE19008	Began 31 October 2019, lasting 45 hours 30 minutes
GCE19007	Began 22 October 2019, lasting 3 hours 43 minutes
GCE19006	Began 24 September 2019, lasting 5 hours 8 minutes
GCE19005	Began 05 September 2(
GCE19003	Began 02 June 2019, las
GCE19002	Began 15 May 2019, lasting 5 hours 1 minute
GCE19001	Began 11 May 2019, lasting 2 hours 11 minutes

Google outage hits Gmail, Snapchat and Nest

after Cloud Platform problem causes email

expected to solve the problem with Gmail in the near future. Photograph: NurPhoto/NurPhoto via

GitHub Suffers Outage Amid Microsoft Cloud Capacity Crunch



Apple iMessage, Mail, and other iCloud services are 'experiencing slower than normal performance'

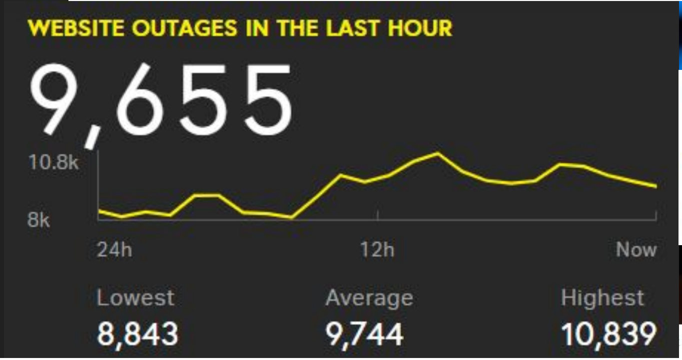
Networks are buggy, and that doesn't surprise you

Comcast Outage : Comcast Internet Down (Xfinity internet not working) : Xfinity Outage

by Ankit kumar — March 25, 2020 in News, Technology 0

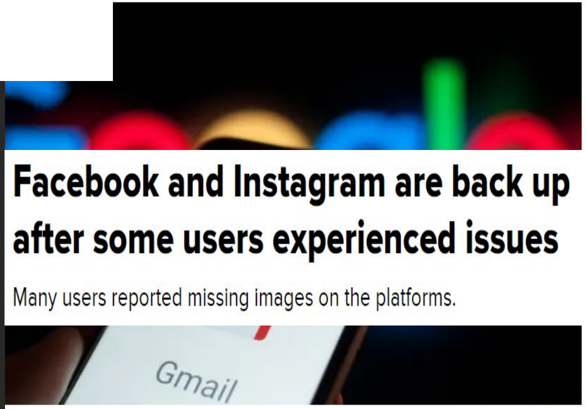
GCE19006	Began 24 September 2019, lasting 5 hours 8 minutes
GCE19005	Began 05 September 2(
GCE19003	Began 02 June 2019, las
GCE19002	Began 15 May 2019, lasting 5 hours 1 minute
GCE19001	Began 11 May 2019, lasting 2 hours 11 minutes

GitHub Suffers Outage Amid Microsoft Cloud Capacity Crunch



Google outage hits Gmail, Snapchat and Nest

after Cloud Platform problem causes email



expected to solve the problem with Gmail in the near future. Photograph: NurPhoto/NurPhoto via

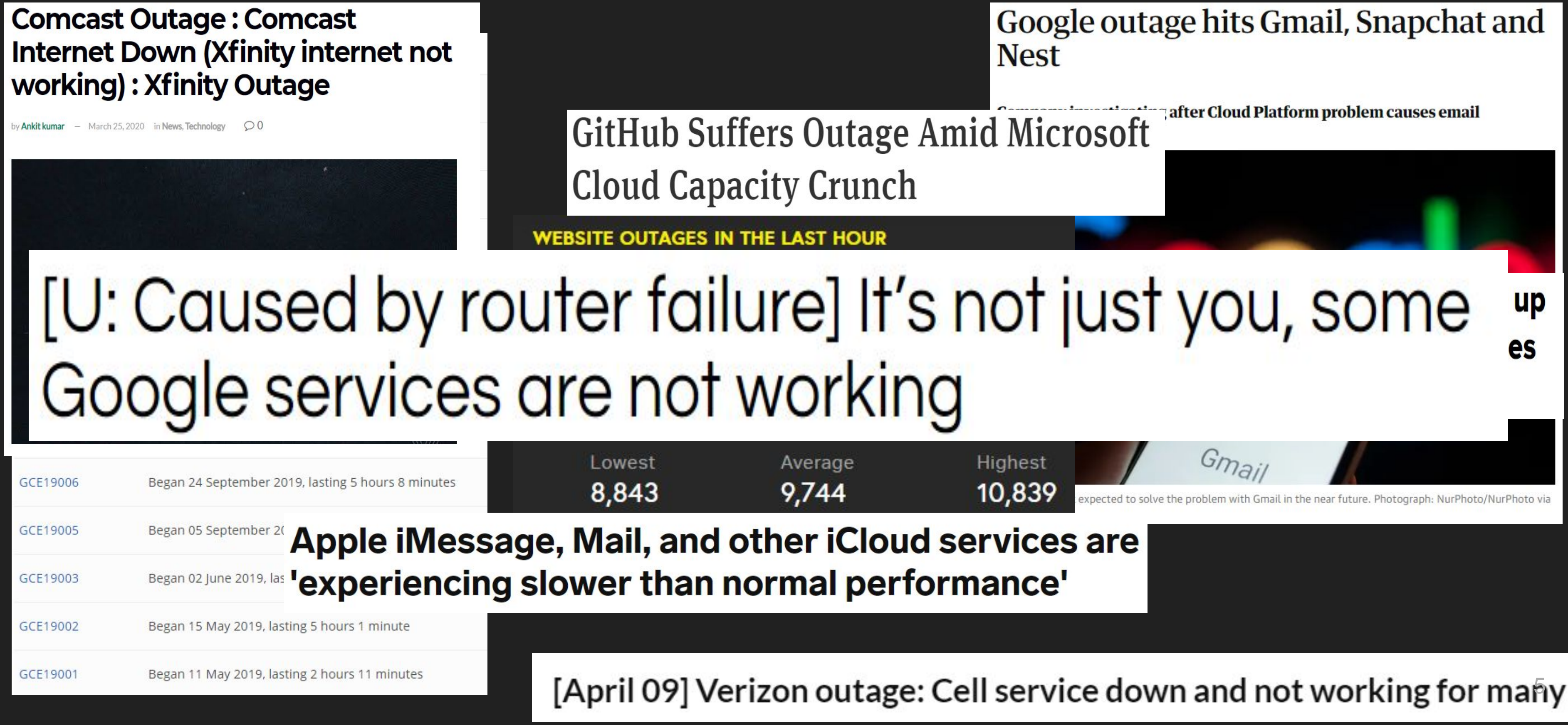
Facebook and Instagram are back up after some users experienced issues

Many users reported missing images on the platforms.

Apple iMessage, Mail, and other iCloud services are 'experiencing slower than normal performance'

[April 09] Verizon outage: Cell service down and not working for many

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

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

Topology

NV: A network verification language

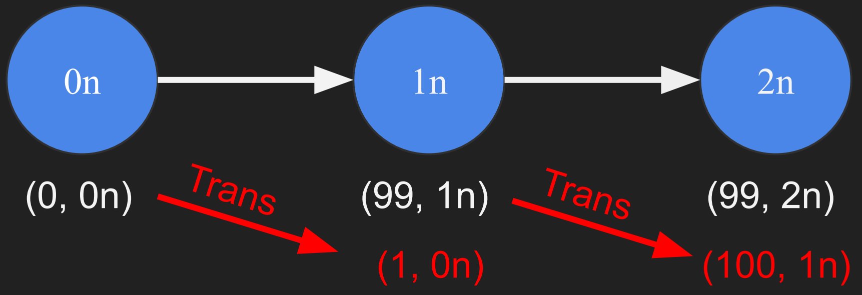
```
let nodes = 3
let edges = { 0-1; 1-2; }

type attribute = (int, tnode)

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

Attributes

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)
```

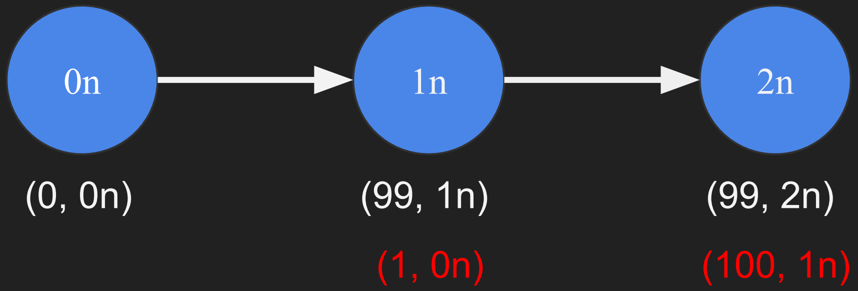
Passing messages



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
```

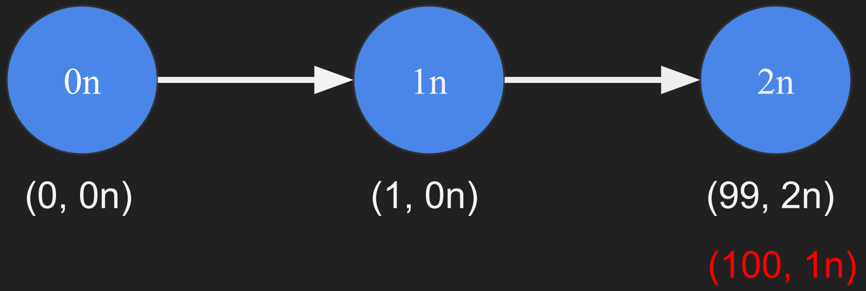
Receiving messages



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
```

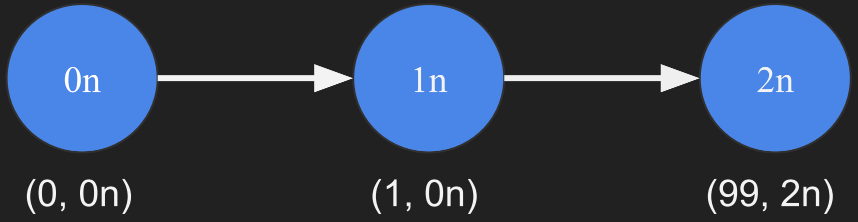
Receiving messages



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
```

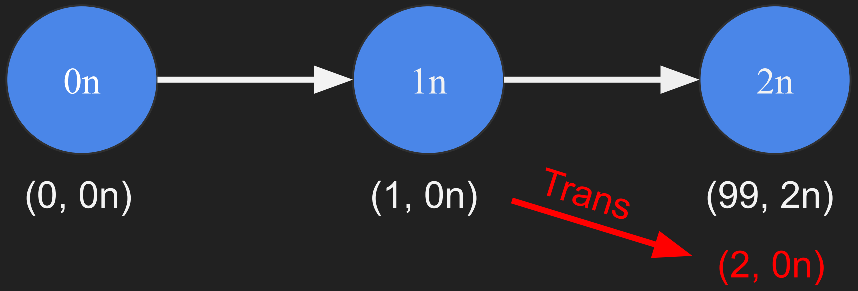
Receiving messages



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
```

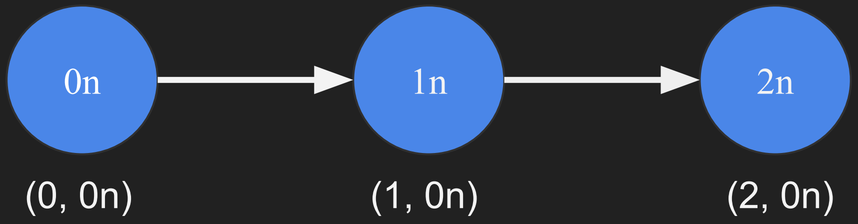
Receiving messages



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

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

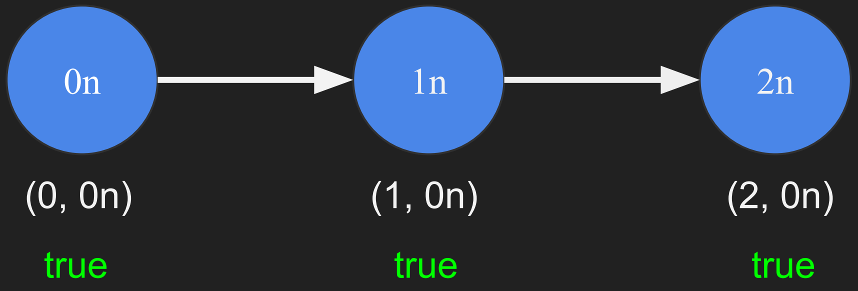
Steady state



NV: A network verification language

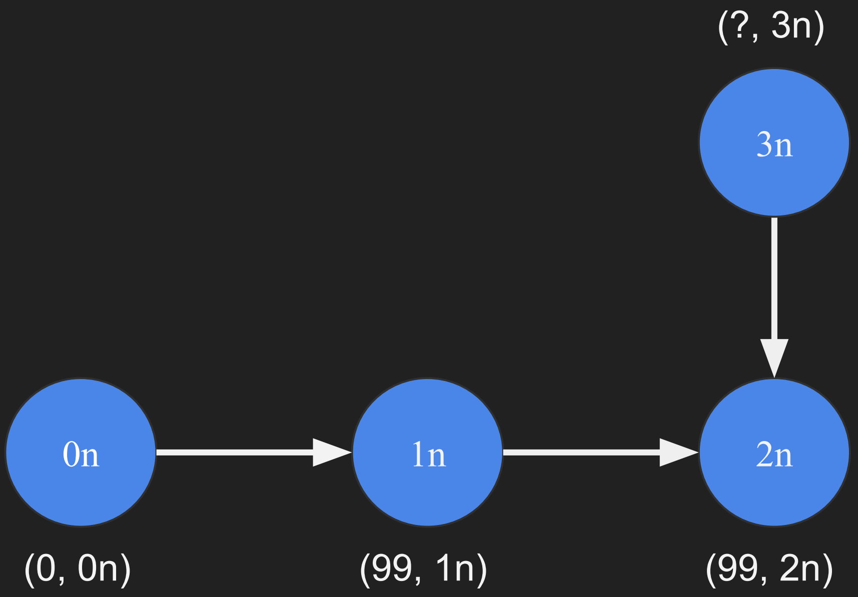
```
let nodes = 3
let edges = { 0-1; 1-2; }

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

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

assert (forall n : tnode,
        let (_, origin) = sol[n] in
        origin = 0n)
```

Verifying properties



Neighbors might send arbitrary messages

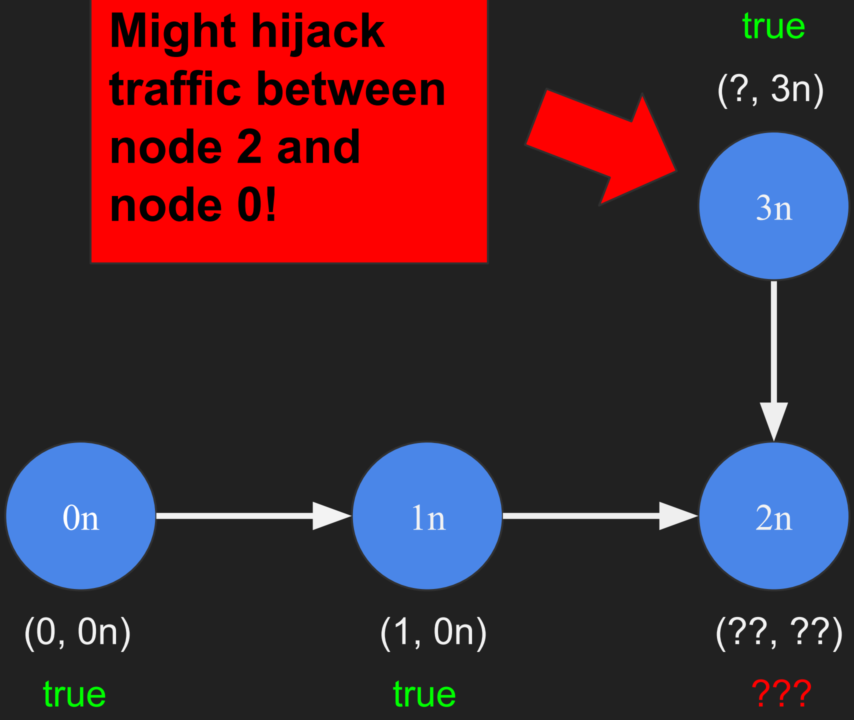
```
let nodes = 4
let edges = { 0-1; 1-2; 3-2; }

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 xdist1 < 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

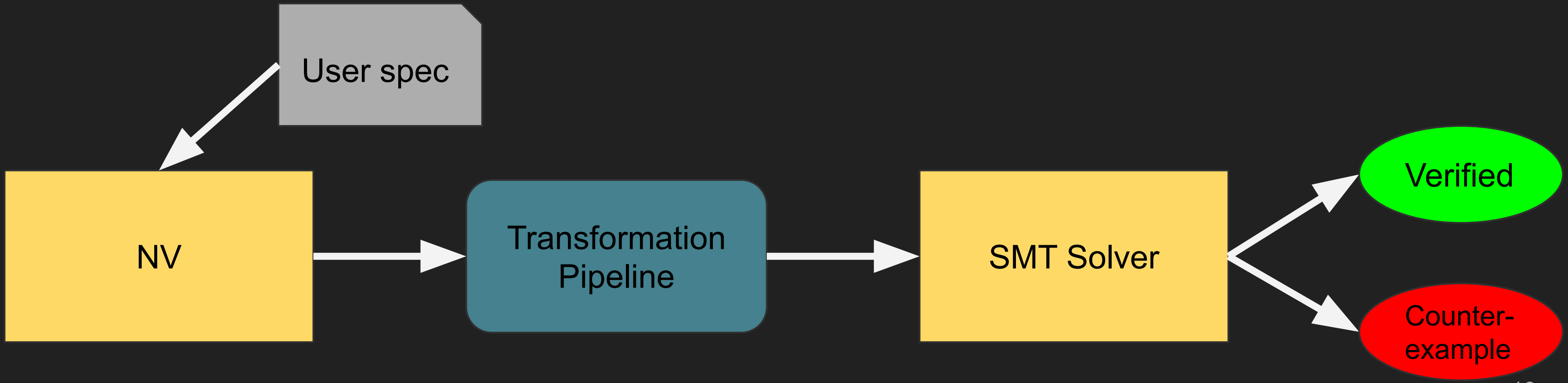
```
let nodes = 4
let edges = { 0-1; 1-2; 3-2; }

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 xdist1 < 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!

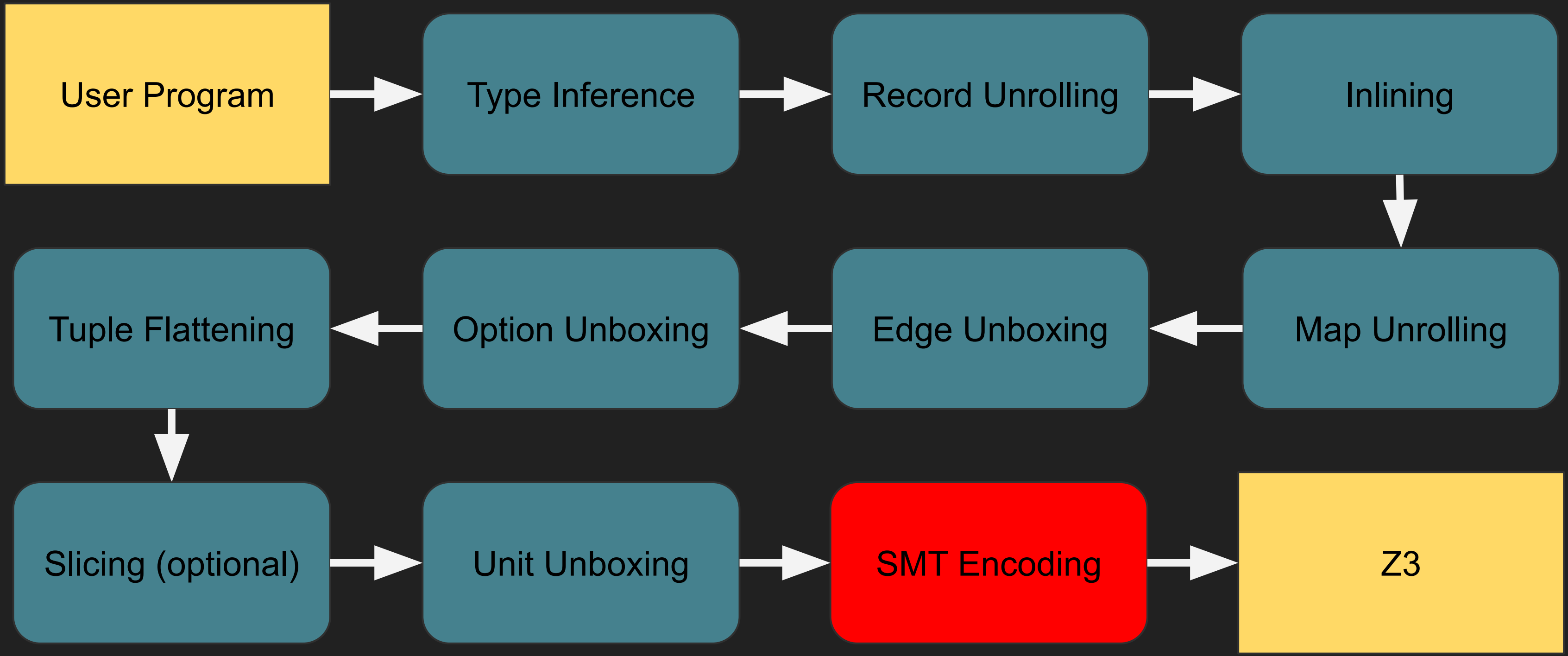


Solution: Use an SMT Solver

- 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



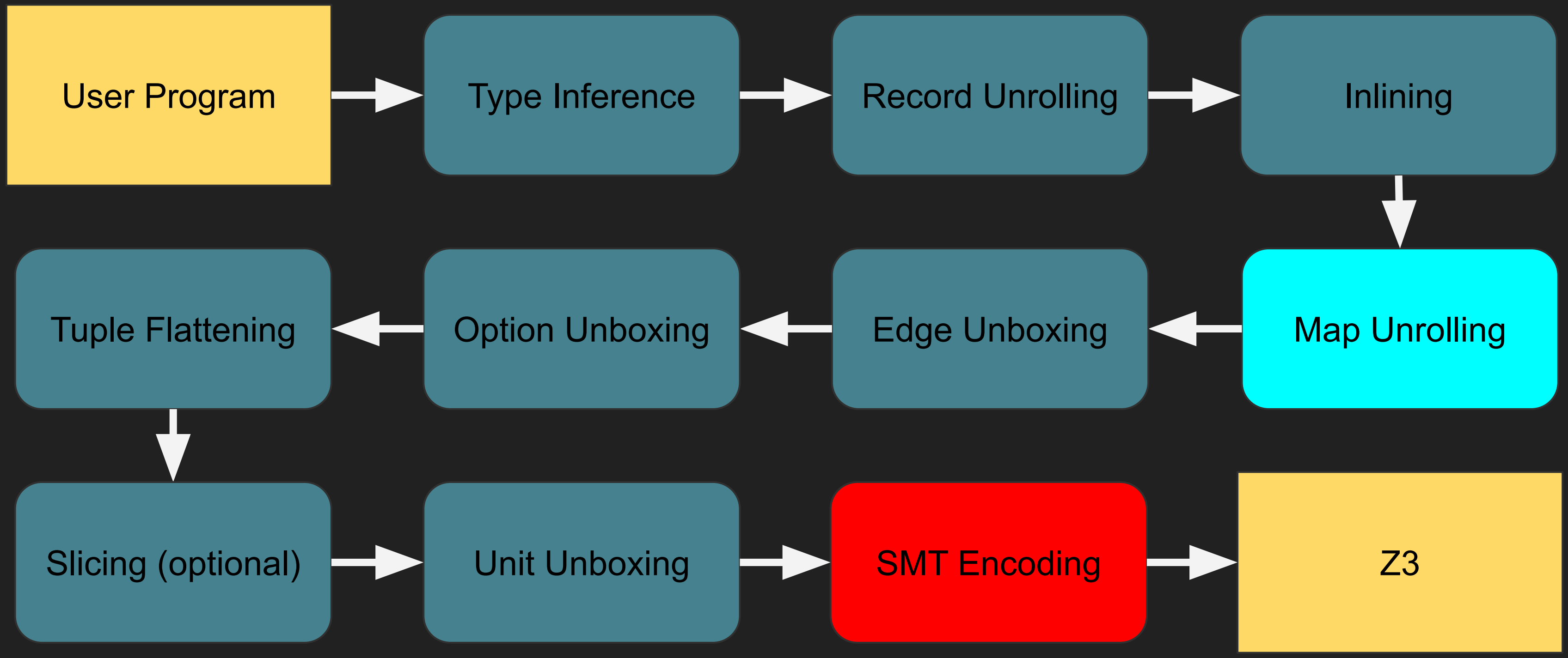
Transformation pipeline (for SMT)



BLUE boxes are compositional NV-to-NV transformations

Most blue boxes use a centralized mechanism for specifying transformations

Transformation pipeline (for SMT)



Map Unrolling has been particularly challenging

Maps in NV

- Maps (or dictionaries) are commonly used in networking
- NV maps are *total*

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

Encoding Map Operations

- Some dictionary operations require quantifiers to encode into SMT

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

↓ SMT Encoding

```
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 ...
```

SMT Encoding

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

SMT Solver

Unknown

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 `m[3]`, never `m[...complicated computation…]`
- Hence we can figure out which keys will be relevant *statically* by simply scanning the program!

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

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

```
type tmap = (bool, bool, bool)
            (* (m[3], m[4], m[7]) *)
let m : tmap = (false, false, false) in
let m = (true, false, false) in
let m = (true, false, true) in
let x = match m with | (v, _, _) → v in
let y = match m with | (_, v, _) → v in
x && y
```

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

```
(* 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)
```

Hiding -- Iteration 1

Hidden Program

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









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

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)
```

Hiding -- Iteration 1

Hidden Program

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









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

SAT:
y1 = true, y2 = false

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)
```

Hiding -- Iteration 1

Hidden Program

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

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

SAT:
y1 = true, y2 = false

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)

y1 = true
y2 = false
```

Hiding -- Iteration 1

Hidden Program

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









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

SAT:
y1 = true, y2 = false

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)

y1 = true
y2 = false
```

UNSAT (Need info on: y1, y2)

Hiding -- Iteration 2

Hidden Program

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


y1 = y0 && x0



y2 = y1 && x1


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

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)
```

Hiding -- Iteration 2

Hidden Program

```
(* 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

```
(* 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)
```

Hiding -- Iteration 2

Hidden Program

```
(* 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

```
(* 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
```

Hiding -- Iteration 2

Hidden Program

```
(* 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

```
(* 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

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

x1 = x0
y1 = y0 && x0



y2 = y1 && x1


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

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)
```

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)
```

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

File	Nodes	Control (seconds)	Hiding (seconds)
sp4	20	0.35	19.9
fat4pol	20	1.1	94
sp8	80	3.8	9358

Experimental Results

File	Nodes	Control (seconds)	Hiding (seconds)	# Iterations
sp4	20	0.35	19.9	112
fat4pol	20	1.1	94	197
sp8	80	3.8	9358	435

Experimental Results

File	Nodes	Control (seconds)	Hiding (seconds)	# Iterations	% Hidden
sp4	20	0.35	19.9	112	0.0%
fat4pol	20	1.1	94	197	32.4%
sp8	80	3.8	9358	435	16.8%

Experimental Results

File	Nodes	Control (seconds)	Hiding (seconds)	# Iterations	% Hidden	Last Iteration (seconds)
sp4	20	0.35	19.9	112	0.0%	0.39
fat4pol	20	1.1	94	197	32.4%	0.49
sp8	80	3.8	9358	435	16.8%	1.5

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 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?