Network Verification: From Algorithms to the Real World

Wenxuan Zhou
Veriflow & UIUC
A simple idea about complexity…

Networks are so complex it’s hard to know they’re doing the right thing.

Let’s automate.
Inside a typical enterprise network

Layer 1 protocols (physical layer)
USB Physical layer
Ethernet physical layer including 10 BASE T, 100 BASE T, 100 BASE TX, 100 BASE FX, 1000 BASE T and other variants
varieties of 802.11 Wi-Fi physical layers
DSL
Configs use many protocols & features

Layer 4 (transport layer or Host-to-Host layer)
AH
TCP
UDP

Layer 5 (session layer or application layer in DoD)
NetBIOS File sharing and name resolution protocol - the basis of file sharing in windows
NFS Network File System

Layer 7 (application layer)
BitTorrent
BGP
DNS
DHCP
FTP
HTTP
HTTPS
IRC
NTP
POP3
RTP
SSH
SMTP
SNMP
Telnet
TFTP
URL

List of protocols commonly encountered by CCNAs
https://learningnetwork.cisco.com/docs/DOC-25649
Distributed route computation
Distributed route computation
Distributed route computation
Distributed route computation
Distributed route computation
Distributed route computation
Distributed route computation
handle(packet p)
  if p.port != 80
    drop
  if p.ipAddr is in 128.0.0.0/8 then
    forward out port 8
  else if p.ipAddr is 10.5.45.43 then
    prepend MPLS header with label 52
    forward out port 42
  ....
Ensuring correct operations today

Manual spot-checking (pings, traceroutes)

Monitoring of events & flows

Screenshot from Scrutinizer
NetFlow & sFlow analyzer,
snmp.co.uk/scrutinizer/
Networks are complex

– Survey of network operators [Kim, Reich, Gupta, Shahbaz, Feamster, Clark, USENIX NSDI 2015]
Networks are complex

89% of operators never sure that config changes are bug-free

– Survey of network operators
  [Kim, Reich, Gupta, Shahbaz, Feamster, Clark,
  USENIX NSDI 2015]
Networks are complex

89% of operators never sure that config changes are bug-free

82% concerned that changes would cause problems with existing functionality

– Survey of network operators
[Kim, Reich, Gupta, Shahbaz, Feamster, Clark, USENIX NSDI 2015]
Network Verification

The process of proving whether an abstraction of the network satisfies intended network-wide properties.
The process of proving whether an abstraction of the network satisfies intended network-wide properties.
Network Verification

The process of proving whether an abstraction of the network satisfies intended network-wide properties.
Network Verification

The process of proving whether an abstraction of the network satisfies intended network-wide properties...

“Host A should be connected to host B.”
Network Verification

The process of proving whether an abstraction of the network satisfies intended network-wide properties.

“Host A should be connected to host B.”

“Host A should not be able to reach service B on any server.”
Network Verification

The process of proving whether an abstraction of the network satisfies intended network-wide properties.

“Host A should be connected to host B.”

“Host A should not be able to reach service B on any server.”

“No packet should fall into a loop.”
Network Verification

The process of proving whether an **abstraction** of the network satisfies intended network-wide **properties**.

“Host A should be connected to host B.”

“Host A should not be able to reach service B on any server.”

“No packet should fall into a loop.”

“All packets should follow shortest paths.”
Network Verification

The process of proving whether an abstraction of the network satisfies intended network-wide properties.
Network Verification

The process of proving whether an abstraction of the network satisfies intended network-wide properties.
Network Verification

The process of proving whether an abstraction of the network satisfies intended network-wide properties.
Network Verification

The process of proving whether an abstraction of the network satisfies intended network-wide properties.
Network Verification

The process of proving whether an abstraction of the network satisfies intended network-wide properties.
Network Verification

The process of proving whether an **abstraction** of the network satisfies intended network-wide **properties**.
Network Verification

The process of proving whether an abstraction of the network satisfies intended network-wide properties.

- **Configuration**
  - Configuration verification

- **Control software**
  - Controller verification & verifiable control languages

- **Data plane state**
  - Data plane verification

- **Packet processing**
  - Software switch verification
Data plane verification
Data plane verification

Verify the network as close as possible to its actual behavior
Data plane verification

Verify the network as close as possible to its actual behavior
Data plane verification

Verify the network as close as possible to its actual behavior

• Insensitive to control protocols
Data plane verification

Verify the network as close as possible to its actual behavior

• Insensitive to control protocols
• Accurate model
Data plane verification

Verify the network as close as possible to its actual behavior

- Insensitive to control protocols
- Accurate model
- *Checks current snapshot*
Verify the network as close as possible to its actual behavior

- Insensitive to control protocols
- Accurate model
- *Checks current snapshot*
- but can be foundation for config verification
Data plane verification architecture
Data plane verification architecture
Data plane verification architecture
Data plane verification architecture
Data plane verification architecture

“Can any packet starting at A reach B?”
“Can any packet starting at A reach B?”
Data plane verification architecture

“Can any packet starting at A reach B?”

Verifier

Diagnosis
A little calculation…
A little calculation…

# theoretical packets
A little calculation…

# theoretical packets

\[ = 2^{(#\text{header bits})} \times #\text{injection points} \]
A little calculation…

# theoretical packets

\[ = 2^{(#\text{header bits})} \times #\text{injection points} \]
\[ = 2^{(18 \text{ byte ethernet} + 20 \text{ byte IPv4})} \times 10,000 \text{ ports} \]
\[ = 3.25 \times 10^{95} \text{ possible packets} \]
A little calculation…

# theoretical packets

\[ = 2^{(#\text{header bits})} \times #\text{injection points} \]

\[ = 2^{(18 \text{ byte ethernet} + 20 \text{ byte IPv4})} \times 10,000 \text{ ports} \]

\[ = 3.25 \times 10^{95} \text{ possible packets} \]
A little calculation…

# theoretical packets

\[ = 2^{(#\text{header\ bits})} \times \#\text{injection points} \]

\[ = 2^{(18\ \text{byte ethernet} + 20\ \text{byte IPv4})} \times 10,000\ \text{ports} \]

\[ = 3.25 \times 10^{95}\ \text{possible packets} \]

Estimated # atoms in observable universe

Grains of sand on earth’s beaches

\[ 5.6 \times 10^{21} \]

\[ 10^{80} \]

\[ 10^{95} \]
A-to-B query with bitmask
A-to-B query with bitmask
A-to-B query with bitmask
A-to-B query with bitmask
A-to-B query with bitmask

Packet: $x[0] \ x[1] \ x[2] \ \ldots \ \ x[n]$
A-to-B query with bitmask

Packet: $x[0] \ x[1] \ x[2] \ldots \ x[n]$
A-to-B query with bitmask

Packet: $x[0] \ x[1] \ x[2] \ ... \ x[n]$
A-to-B query with bitmask

Packet: $x[0] \ x[1] \ x[2] \ ... \ x[n]$
A-to-B query with bitmask

Packet: $x[0] \ x[1] \ x[2] \ ... \ x[n]$
A-to-B query with bitmask

Packet: $x[0] \ x[1] \ x[2] \ \ldots \ x[n]$

$$\left( x_4 \lor x_7 \lor \bar{x}_{1} \right) \land (\ldots) \land (\ldots) \land (\ldots)$$

NP-complete!
Anteater’s solution

Express data plane and invariants as SAT

- ...up to some max # hops
- Dynamic programming to deal with exponential number of paths
- Model packet transformations with vector of packet “versions” & constraints across versions

Check with off-the-shelf SAT solver (Boolector)
Experiences with real network

Evaluated Anteater with operational network

• ~178 routers supporting >70,000 machines
• Predominantly OSPF, also uses BGP and static routing
• 1,627 FIB entries per router (mean)
• State collected using operator’s SNMP scripts

Revealed 23 violations of 3 invariants in 2 hours

<table>
<thead>
<tr>
<th></th>
<th>Loop</th>
<th>Packet loss</th>
<th>Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Being fixed</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Stale config.</td>
<td>0</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>Total alerts</td>
<td>9</td>
<td>17</td>
<td>2</td>
</tr>
</tbody>
</table>
CAN WE VERIFY NETWORKS IN REAL TIME?

VeriFlow: Verifying Network-Wide Invariants in Real Time

Khurshid, Zou, Zhou, Caesar, Godfrey

HotSDN’12 best paper, NSDI’13
Not so simple
Not so simple

Challenge #1: Obtaining real time view
Not so simple

Challenge #1: Obtaining real time view

Challenge #2: Verify quickly
“Service S reachable only through firewall?”

“Is segment isolated?”
VeriFlow [NSDI’13] architecture

Thin, standard interface to data plane (e.g. OpenFlow)
VeriFlow [NSDI’13] architecture
Here that the PC be valid. The PV replies with the PCx where
the form as RSA. The client then sends a request to the PV of
and have each generated a public-private key pair e
in Figure rl. Before the process begins, the client and PV
quest sl.

PV the server does not trust, it can fall back to a qWH third parties can exist in parallell. If the client uses a
DNS servers are examples in today's Internet. not unprecedentedx certificate authorities and the root
is that servers need to trust a third partyl. But this is
delay for each new server or domainl. The disadvantage
The advantage is that a client can avoid paying an RTT
web sites or content distribution networksl. that attract the same user frequentlyi such as popular
a single RTTl. This will be highly e
RTTs eas in TCPfi and subsequent connections require
the server directlyl. Thus, the first connection takes two
level request to the server. Flery, it can contact
a PC from the PV prior to initiating the applicationk.
The first time a client contacts a domain, it obtains
PV run by its domain at a known location e
that avoids replay attacksl. in a way multiple requests to place cryptographic proof of provek.
tographic proof that the PV recently verified that the
with a

\[ \text{PC} = \{ \text{Kpub}, \text{pv}, c, t, d \} \]

4.2 Verifying provenance without a handshake
guaranteei yet without introducing an RTT delay. this section is to demonstrate a practical means for a
server to verify source provenance similar to the qWH's
discussion in

4.2.2 Obtaining a Provenance Certificate

The protocol by which a client obtains a PC is shown
The above two solutions, and multiple di
Firsti the PV may simply be the web server itselfi or a
Lifesaver leverages cryptographic proof to verify the
provenance verifier (PV)

4.2.1 Choosing a Provenance Verifier

VeriFlow [NSDI’13] architecture
VeriFlow [NSDI’13] architecture

- **app**
- **app**
- **SDN controller**

VeriFlow

Logically centralized controller

Thin, standard interface to data plane (e.g. OpenFlow)
Verifying invariants quickly

Overview:

1. Limit the search space
2. Represent forwarding behaviors using graphs
3. Run Light-weight graph-based algorithm to check reachability properties
Verifying invariants quickly

Overview:

1. Limit the search space
2. Represent forwarding behaviors using graphs
3. Run Light-weight graph-based algorithm to check *reachability properties*
Verifying invariants quickly

Overview:
1. Limit the search space
2. Represent forwarding behaviors using graphs
3. Run Light-weight graph-based algorithm to check reachability properties
Verifying invariants quickly

Overview:
1. Limit the search space
2. Represent forwarding behaviors using graphs
3. Run Light-weight graph-based algorithm to check reachability properties
Verifying invariants quickly

Overview:

1. Limit the search space
2. Represent forwarding behaviors using graphs
3. Run Light-weight graph-based algorithm to check **reachability properties**
Verifying invariants quickly

Overview:

1. Limit the search space
2. Represent forwarding behaviors using graphs
3. Run Light-weight graph-based algorithm to check reachability properties
MODELING DYNAMIC NETWORKS

Enforcing Customizable Consistency Properties in Software-Defined Networks

Zhou, Jin, Croft, Caesar, Godfrey
NSDI 2015
Timing uncertainty
Timing uncertainty

Controller

Rule 1

Switch A → Switch B
Timing uncertainty

Controller

Remove rule 1 (delayed)

Switch A

Rule 1

Switch B
Timing uncertainty

Remove rule 1 (delayed)
Timing uncertainty

Remove rule 1 (delayed)

Switch A → Switch B

Possible network states:
Timing uncertainty

Possible network states:

- 
- 
- 

Controller

Switch A

Switch B

Remove rule 1 (delayed)

Install rule 2

Rule 1
Timing uncertainty

Possible network states:
Timing uncertainty

Controller

Remove rule 1 (delayed)

Install rule 2

Rule 1

Switch A

Switch B

Possible network states:
Timing uncertainty

Controller

Switch A

Switch B

Remove rule 1 (delayed)

Install rule 2

Rule 1

Rule 2

Possible network states:
Timing uncertainty

One solution: “consistent updates”  
Uncertainty-aware verification
Uncertainty-aware verification
Update synthesis via verification

CCG

Stream of Updates

Controller

Verifier

Verification Engine

Network Model

Safe?

Yes

No

A

B

C

D

E

F

G

H

Confirmation
Update synthesis via verification

Controller -> Stream of Updates

CCG

Update queue

No -> Safe?

Yes

Verifier

Verification Engine

Network Model

A should reach B

A -> C -> D -> E

F -> G -> H -> B
Update synthesis via verification

**Controller**


**CCG**

- Update queue: No
- Safe?: Yes

**Verifier**

- Verification Engine
- Network Model

**Confirmations**

- A should reach B

Diagram:

- Network: A, B, C, D, E, F, G, H
Update synthesis via verification

Controller

Stream of Updates

CCG

Update queue

Verifier

No

Safe?

Yes

Verification

Engine

Network Model

Confirmations

1 mod A->C to A->F
2 add F->G
3 add G->H
4 add H->B

A should reach B

A

B

C

D

E

F

G

H
Update synthesis via verification

Controller

CCG

Update queue

Verifier

Verification Engine

Network Model

Stream of Updates

A should reach B

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>mod A-&gt;C to A-&gt;F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>add F-&gt;G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>add G-&gt;H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>add H-&gt;B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Confirmanations

C

D

E

F

G

H

A

B
Update synthesis via verification

- **Controller**
  - Stream of Updates
  - 1: mod A->C to A->F
  - 2: add F->G
  - 3: add G->H
  - 4: add H->B

- **CCG**
  - **Update queue**
  - **Safe?**
  - 4: Yes

- **Verifier**
  - **Verification Engine**
  - **Network Model**

- Diagram:
  - **A should reach B**
  - Network: A → C → D → E → B, F → G → H
Update synthesis via verification

Controller

Stream of Updates

CCG

Update queue

Verifier

Verification Engine

Network Model

A should reach B

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

A

2

1

3

4

B

C

D

E

F

G

H

Yes

No

Safe?
Update synthesis via verification

Stream of Updates

Verifier
Verification Engine
Network Model

Controller

CCG

Update queue

Safe?

No

Yes

Confirinations

A should reach B

1
mod A->C to A->F

2
add F->G

3
add G->H

4
add H->B

A
C
D
E
F
G
H
B
Update synthesis via verification

Controller

Stream of Updates

CCG

Update queue

Safe?

Verifier

Verification Engine

Network Model

Confirmations

A should reach B

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

A
B
C
D
E
F
G
H

1  2  1  3

© 2017 Veriflow. All rights reserved.
Update synthesis via verification

CCG

Controller

Update queue

Verifier

Verification Engine

Network Model

Stream of Updates

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

A should reach B

Confirms
Update synthesis via verification

Controller

Stream of Updates

CCG

Update queue

Verifier

Verification Engine

Network Model

Confirmations

A should reach B

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

A should reach B

Network Model

A

B

F

G

H

C

D

E
Update synthesis via verification

Controller

Stream of Updates

CCG

Update queue

Verifier

Verification Engine

Network Model

Confirmation:

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

A should reach B

A \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G \rightarrow H \rightarrow B
Update synthesis via verification

Controller

Stream of Updates

CCG

Update queue

Yes

Safe?

No

Verifier

Verification Engine

Network Model

Confirmations

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

A should reach B
Update synthesis via verification

Controller

Stream of Updates

CCG

Update queue

Verifier

Verification Engine

Network Model

Confirmations

A should reach B

1
mod A->C to A->F

2
add F->G

3
add G->H

4
add H->B
Update synthesis via verification

Controller

CCG

Update queue

Verifier

Verification Engine

Network Model

Stream of Updates

1. mod A→C to A→F
2. add F→G
3. add G→H
4. add H→B

A should reach B

Confirmations

Safe?

Yes

No

A

B

C

D

E

F

G

H

1

2
Update synthesis via verification

CCG

Update queue

Controller

Verifier

Verification Engine

Network Model

Stream of Updates

Safe?

Yes

No

1 mod A→C to A→F

2 add F→G

3 add G→H

4 add H→B

A should reach B

A

C

D

E

F

G

H

B
Update synthesis via verification

Controller

Stream of Updates

CCG

Update queue

Safe?

Verifier

Verification Engine

Network Model

Confirmations

A should reach B

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

A should reach B

Network Model
Update synthesis via verification

Controller

CCG

Update queue

Verifier

Network Model

Verifications

Yes

Safe?

No

A should reach B

Stream of Updates

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B
Update synthesis via verification

Controller

Update queue

CCG

No

Safe?

Yes

Verifier

Verification Engine

Network Model

Stream of Updates

A should reach B

Confirmations

1
mod A->C to A->F

2
add F->G

3
add G->H

4
add H->B

A

B

C

D

E

F

G

H
Update synthesis via verification

Stream of Updates

Controller

CCG
Update queue

Verifier
Verification Engine
Network Model

Safe?

A should reach B

Confirmation

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

A
B
C
D
E
F
G
H
Update synthesis via verification

Controller

Stream of Updates

CCG

Update queue

Verifier

Verification Engine

Network Model

No

Safe?

Yes

Confirmations

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

A should reach B

A

C

D

E

B

F

G

H
Update synthesis via verification

CCG

Controller

Stream of Updates

Update queue

Verifier

Verification Engine

Network Model

Confirmations

A should reach B

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

A should reach B
Update synthesis via verification

Controller

Stream of Updates

CCG

Update queue

Verifier

Verification Engine

Network Model

A should reach B

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

Safe?

Yes

No
Update synthesis via verification

Stream of Updates

CCG

Controller

Update queue

Verifier

Verification Engine

Network Model

Confirmaions

A should reach B

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

1

2

3

4

1

2

3

4

A

B

C

D

E

F

G

H

Safe?

No

Yes
Update synthesis via verification

1. mod A→C to A→F
2. add F→G
3. add G→H
4. add H→B

A should reach B

CCG

Controller

Stream of Updates

Verification

Engine

Network

Model

Confirmations

Update queue

Safe?

Yes

No
Update synthesis via verification

Controller

Update queue

CCG

Stream of Updates

Verifier

Verification Engine

Network Model

Confirmations

A should reach B

A

B

C

D

E

F

G

H

1. mod A→C to A→F
2. add F→G
3. add G→H
4. add H→B

Safe?

Yes

No
Update synthesis via verification

Controller

Stream of Updates

CCG

Update queue

Verifier

Verification Engine

Network Model

Confirmations

1. mod A→C to A→F
2. add F→G
3. add G→H
4. add H→B

A should reach B

A

B

C

D

E

F

G

H
Update synthesis via verification

Controller

Stream of Updates

CCG

Update queue

No

Safe?

Yes

Verifier

Verification Engine

Network Model

Confirmations

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

A should reach B
Update synthesis via verification

Stream of Updates

Controller

Update queue

CCG

Safe?

Verifier

Verification Engine

Network Model

No

Yes

1. mod A→C to A→F
2. add F→G
3. add G→H
4. add H→B

A should reach B
Update synthesis via verification

CCG → Update queue → Safe?

Controller

Stream of Updates

Verifier

Verification Engine → Network Model

Confirmation

A should reach B

1. mod A→C to A→F
2. add F→G
3. add G→H
4. add H→B
Update synthesis via verification

CCG → Update queue

Controller

Stream of Updates

Verifier

Verification Engine

Network Model

Yes → Safe?

No → CCG

Confirractions

1. mod A->C to A->F
2. add F->G
3. add G->H
4. add H->B

A should reach B

Enforcing dynamic correctness with heuristically maximized parallelism
VERIFICATION IN THE REAL WORLD:
WHAT DID WE LEARN?
Industry efforts

Three startups pursuing general-purpose network verification for enterprises

Special purpose efforts

• Hyperscale clouds
• Major network device manufacturer

Gartner grouping verification in “intent-based networking” category
Industry efforts

Three startups pursuing general-purpose network verification for enterprises

Special purpose efforts

- Hyperscale clouds
- Major network device manufacturer

Gartner grouping verification in “intent-based networking” category

What have we learned?
1. The Need is Real
1. The Need is Real
1. The Need is Real
1. The Need is Real

Network Complexity
1. The Need is Real

Network Complexity

59% say growth in complexity has led to more frequent outages

[Dimensional Research]
1. The Need is Real

Network Complexity

59% say growth in complexity has led to more frequent outages

[Dimensional Research]
1. The Need is Real

Network Complexity

59% say growth in complexity has led to more frequent outages
[Dimensional Research]

Change

22,000 changes/mo. at DISA [S. Zabel, 2016]
1. The Need is Real

Network Complexity

59% say growth in complexity has led to more frequent outages
[Dimensional Research]

Change

22,000 changes/mo. at DISA [S. Zabel, 2016]

Manual Processes
1. The Need is Real

**Network Complexity**
59% say growth in complexity has led to more frequent outages
[Dimensional Research]

**Change**
22,000 changes/mo. at DISA [S. Zabel, 2016]

**Manual Processes**
69% use manual checks (most common technique)
[Dimensional Research]
2. How is it actually useful?
2. How is it actually useful?

Availability & Resilience
2. How is it actually useful?

Availability & Resilience

Network Segmentation
2. How is it actually useful?

Availability & Resilience

Network Segmentation

Continuous Compliance
2. How is it actually useful?

- Availability & Resilience
- Network Segmentation
- Continuous Compliance
- Incident Response
3. Extracting the abstraction: not easy
Software verification

```c
#include <stdio.h>

int main(int argc, char** argv) {
    if (argv >= 2) {
        printf("Hello world, %s!", argv[1]);
    }
    return 0;
}
```

- Given program as input
- Assume formal specification of programming language
3. Extracting the abstraction: not easy

Software verification

Data plane verification

- No universal API to extract state (LCD: SSH + CLI “show” commands”)
- No formal spec of how that state relates to functionality
- Vendor-specific behaviors

```
#include <stdio.h>

int main(int argc, char** argv) {
    if (argv >= 2) {
        printf("Hello world, %s!", argv[1]);
    }
    return 0;
}
```

- Given program as input
- Assume formal specification of programming language
3. Extracting the abstraction: not easy

Data plane verification

- No universal API to extract state (LCD: SSH + CLI “show” commands”)
- No formal spec of how that state relates to functionality
- Vendor-specific behaviors

Broadcom’s OF-DPA 1.0 Abstract Switch
3. Extracting the abstraction: not easy

Data plane verification

- No universal API to extract state (LCD: SSH + CLI “show” commands”)
- No formal spec of how that state relates to functionality
- Vendor-specific behaviors

Some hope: Vendor-specific APIs, OpenConfig
4. Model / Verifier separation works

**Predictive model**

(“forwarding graph”)

![Diagram of model and verifier separation](image)
The obvious implementation of the above exchange is the client contacting the 
PV directly. Thus, the first connection takes two RTTs and subsequent connections 
require one RTT if the client was reachable at a certain IP address. After obtaining 
a provenance certificate (PC), the client can use it for secure communication. 
Each request sent by the client includes the PC, which the PV verifies. 

The PV replies with a PC that corresponds to cryptographic proof that the PV 
recently verified the client's identity. The PC is valid for a certain duration, 
which is determined by the current time and sets the PC will remain valid. 
The PV sets the PC's validity based on the current time and the PV's internal 
maximum time, perhaps on a daily or weekly basis.

To obtain a PC, the client must first obtain a request certificate (RC). 
The RC is a secure hash function of the request packet sent to servers, in a way 
that avoids replay attacks. This allows the client to prove the authenticity of the 
connection by including the RC in subsequent requests. 

The PV may be any party trusted by the server. We envision two common use cases: 

1. A web server may be the PV, or it may be a trusted third party running a PV service. 
2. Trusted third parties could run a PV service for domains that the client does not trust. 

First, the PV may simply be the web server itself, or a trusted third party. 
In this case, the client would verify the server's identity by checking the 
PC. Second, trusted third parties could run a PV service on behalf of 
domains that the client does not trust. 

Lifesaver leverages cryptographic proof to verify the provenance of 
connections and uses PCs and RCs to achieve this. PCs are a single UDP 
message that is sent to the PV, and the RC is a hash of the request packet sent to 
servers. In our implementation, however, each message would use TCP, 
proving provenance via TCP's predictability. 

4. Model / Verifier separation works

4.2 Verifying provenance without a handshake

4.2.2 Obtaining a Provenance Certificate

The protocol by which a client obtains a PC is shown in the diagram. 
The above two solutions, and multiple different implementations, exist. 
First, the PV may simply be the web server itself, or a trusted third party. 
Second, trusted third parties could run a PV service for domains that the client does not trust.

To do this, the client begins by constructing a message of the form 

\[ \text{PC} = \{ K \text{priv}, \text{pub} \text{c}, \text{a}, \text{d}, \text{c}, \text{t}, \text{d} \} \text{K} \text{priv} \text{c} \text{pub} \text{c} \text{a} \text{d} \text{c} \text{t} \text{d} \]

The client then opens a transport connection to the PV and includes the PC in the handshake 
message. The PV replies with the PC, and the client verifies the PC by checking the cryptographic proof. 

The advantage of using PCs is that a client can avoid paying an RTT for each new server or domain. 
This is less vulnerable to DoS attacks and more efficient. The disadvantage is that servers need to trust a third party. 
But this is not unprecedented; certificate authorities and the root DNS servers are examples in today's Internet.

If the client uses a DH key exchange, it can contact a server using the Lifesaver protocol. 
In our implementation, however, each message would use TCP, proving provenance via TCP's predictability.

The obvious implementation of the above exchange is the client contacting the 
PV directly. Thus, the first connection takes two RTTs and subsequent connections require one RTT if the client was reachable at a certain IP address. After obtaining a provenance certificate (PC), the client can use it for secure communication. Each request sent by the client includes the PC, which the PV verifies.
4. Model / Verifier separation works

verify visualize search API

Predictive model (“forwarding graph”)
5. We need a shift in thought
5. We need a shift in thought
5. We need a shift in thought

Network as individual devices

Individual config knobs
5. We need a shift in thought

Network as individual devices

Individual config knobs
5. We need a shift in thought

Network as individual devices  
Individual config knobs

Network as one system  
End-to-end intent
Data plane verification

Static

• On static reachability in IP networks [Xie, Zhan, Maltz, Zhang, Greenberg, Hjalmtysson, Rexford, INFOCOM ’05]
  - Essentially early form of data plane verification
  - Computed reachable sets with IP forwarding rules
• FlowChecker [Al-Shaer, Al-Haj, SafeConfig ’10]
• Anteater [Mai, Khurshid, Agarwal, Caesar, G., King, SIGCOMM’11]
• Header Space Analysis [Kazemian, Varghese, and McKeown, NSDI ’12]
• Network-Optimized Datalog (NoD) [Lopes, Bjørner, Godefroid, Jayaraman, Varghese, NSDI 2015]

Real time (incremental)

• VeriFlow [Khurshid, Zou, Zhou, Caesar, G., HotSDN’12, NSDI’13]
• NetPlumber [Kazemian, Chang, Zeng, Varghese, McKeown, Whyte, NSDI ’13]
• CCG [Zhou, Jin, Croot, Caesar, G., NSDI’15]
Optimizations

- **Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks** [Zeng, Zhang, Ye, Google, Jeyakumar, Ju, Liu, McKeown, Vahdat, NSDI’14]
- **Atomic Predicates** [Yang, Lam, ToN’16]
- **ddNF** [Bjorner, Juniwal, Mahajan, Seshia, Varghese, HVC’16]
Configuration verification

- **RCC** (Detecting BGP config faults w/static analysis) [Feamster & Balakrishnan, USENIX ’05]
- **ConfigAssure** [Narain et al, ’08]
- **ConfigChecker** [Al-Shaer, Marrero, El-Atawy, ICNP ‘09]
- **Batfish** [Fogel, Fung, Pedrosa, Walraed-Sullivan, Govindan, Mahajan, Millstein, NSDI’15]
- **Bagpipe** [Weitz, Woos, Torlak, Ernst, Krishnamurthy, Tatlock, NetPL’16 & OOPSLA’16]
Richer verification

Richer data plane models

- **Software Dataplane Verification** [Dobrescu, Argyraki, NSDI’14]
- **SymNet** [Stoenescu, Popovici, Negreanu, Raiciu, SIGCOMM’16]
- **Mutable datapaths** [Panda, Lahav, Argyraki, Sagiv, Shenker, NSDI’17]

Verifiable controllers & control languages

- **NICE** [Canini, Venzano, Perešini, Kostić, Rexford, NSDI’12]
- **NetKAT** [Anderson, Foster, Guha, Jeannin, Kozen, Schlesinger, Walker, POPL’14]
- **Kinetic: Verifiable Dynamic Network Control** [Kim, Gupta, Shahbaz, Reich, Feamster, Clark, NSDI’15]
FUTURE RESEARCH DIRECTIONS FOR VERIFICATION
Research Directions
Research Directions

1. Pushing the limit towards richer models

- Software pipelines
- Verifiable SDN Controllers
- Stateful networks
- Higher layer concepts
Research Directions

1. Pushing the limit towards richer models
   - Software pipelines
   - Verifiable SDN Controllers
   - Stateful networks
   - Higher layer concepts

2. Moving from verification to remediation
Research Directions

1. Pushing the limit towards richer models
   - Software pipelines
   - Verifiable SDN Controllers
   - Stateful networks
   - Higher layer concepts

2. Moving from verification to remediation

3. Making it easy for users to express intent