Network Verification
Past, Present, and Future

Nate Foster
Cornell
A Bit of History...

The Design Philosophy of the DARPA Internet Protocols

David D. Clark*
Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, MA. 02139


Abstract

The Internet protocol suite, TCP/IP, was first proposed fifteen years ago. It was developed by the Defense Advanced Research Projects Agency (DARPA), and has been used widely in military and commercial systems. While there have been papers and specifications that describe how the protocols work, it is sometimes difficult to deduce from these why the protocol is as it is. For example, the Internet protocol is based on a connectionless or datagram mode of service. The motivation for this has been greatly misunderstood. This paper attempts to capture some of the early reasoning which shaped the Internet protocols.

1. Introduction

For the last 15 years¹, the Advanced Research Projects Agency of the U.S. Department of Defense has been developing a suite of protocols for packet switched networks. The protocols are designed to be used in a variety of applications, including file transfer, terminal access, and electronic mail. The protocols are designed to be robust and reliable, and to provide a high level of service to users.

The design philosophy of the Internet architecture is still evolving. Sometimes a new extension challenges one of the design principles, but in any case an understanding of the history of the design provides a necessary context for current design extensions. The connectionless configuration of ISO protocols has also been colored by the history of the Internet suite, so an understanding of the Internet design philosophy may be helpful to those working with ISO.

This paper catalogs one view of the original objectives of the Internet architecture, and discusses the relation between these goals and the important features of the protocols.
“While tools to verify logical correctness are useful, both at the specification and implementation stage, they do not help with the severe problems that often arise related to performance”
A Bit of History…

"While tools to verify logical correctness are useful, both at the specification and implementation stage, they do not help with the severe problems that often arise related to What’s Different?"
A Bit of History…

“While tools to verify logical correctness are useful, both at the specification and implementation stage, they do not help with the severe problems that often arise related to:

- **Scale and complexity**
  - Today’s networks orders of magnitude bigger
  - And expected to provide richer levels of service

What’s Different?
A Bit of History…

While tools to verify logical correctness are useful, both at the specification and implementation stage, they do not help with the severe problems that often arise related to:

- **Scale and complexity**
  - Today’s networks orders of magnitude bigger
  - And expected to provide richer levels of service

- **New architectures**
  - Software-defined networking
  - Programmable data planes

What’s Different?
A Bit of History…

While tools to verify logical correctness are useful, both at the specification and implementation stage, they do not help with the severe problems that often arise related to

• Scale and complexity
  – Today’s networks orders of magnitude bigger
  – And expected to provide richer levels of service

• New architectures
  – Software-defined networking
  – Programmable data planes

• Formal methods advances
  – Sophisticated modeling techniques
  – Fast solvers (e.g., SAT/SMT)
Imagine if we could…

Find faults in configurations
Imagine if we could…

Find faults in configurations

Make changes with confidence!
Imagine if we could…

Find faults in configurations

Make changes with confidence!

Gain portability across platforms
This Talk

Formal verification of networks:
an idea whose time has come!

Plan

• Single-node verification
• Network-wide verification
• Future directions
Data Plane Verification
Conventional Validation
Conventional Validation

Q: how do we know they work?
Conventional Validation

Q: how do we know they work?
A: by testing!
Conventional Validation

Q: how do we know they work?

A: by testing!

- Expensive: many packet formats & protocols
- Pay cost once, during manufacturing
Programmable Data Plane

...how do they work?

Barefoot Tofino chip

Cisco Nexus 3400 Switch
Programmable Data Plane

...how do they work?
Programmable Data Plane

- Benefits of programmability
  - Flexible customization
  - Rapid innovation
  - Novel uses of network

But conventional validation approaches no longer scale
Programmable Data Plane

- Benefits of programmability
  - Flexible customization
  - Rapid innovation
  - Novel uses of network
- But conventional validation approaches no longer scale

Insight: can use language-based analysis techniques
Anatomy of a P4 Program

Types
- Headers and Instances
  - header_type ethernet_t {
    fields {
      dst_addr:48;
      src_addr:48;
      ether_type:16;
    }
  }
  header_type ipv4_t {
    fields {
      pre_tcl:64;
      ttl:8;
      protocol:8;
      checksum: 16;
      src_addr:32;
      dst_addr: 32;
    }
  }
  header ethernet_t ethernet;
  header ipv4_t ipv4;

Parsers
- parser start
  - extract(ethernet);
  - return select(ethernet.ether_type) {
    0x800: parse_ipv4;
    default: ingress;
  }
- parser parse_ipv4 {
  - extract(ipv4);
  - return ingress;
}

Actions
- action allow() {
  modify_field(standard_metadata.egress_spec,1); 
}
- action deny() { drop(); }
- action nop() { }
- action rewrite(src_addr,dst_addr) {
  modify_field(ipv4.dst_addr,src_addr);
  modify_field(ipv4.dst_addr,dst_addr);
}

Tables
- table acl {
  reads {
    ipv4.src_addr:lpm;
    ipv4.dst_addr:lpm;
  }
  actions { allow; deny; }
  size: 1024;
}
- table nat {
  reads { ipv4.dst_addr:lpm; }
  actions { rewrite; nop; }
  default_action: nop();
  size: 8192;
}

Controls
- control ingress {
  apply(acl);
  apply(nat);
}
P4 Machine Model

PISA [SIGCOMM 2013]
Protocol-Independent Switch Architecture

Parser  Ingress  Queuing  Egress  Deparser

Traffic Manager
Running Example: Firewall

- P4 is a low-level language → many gotchas
- Let’s explore by example
Running Example: Firewall

- P4 is a low-level language → many gotchas
- Let’s explore by example

```p4
action allow() {
    modify_field(std_meta.egress_spec, 1);
}

action deny() { drop(); }

table acl {
    reads { ipv4.dst_addr: lpm; }
    actions { allow; deny; }
}

control ingress { apply(acl); }
```
Running Example: Firewall

• P4 is a low-level language → many gotchas

• Let’s explore by example

```p4
action allow() {
    modify_field(std_meta.egress_spec, 1);
}

action deny() { drop(); }

table acl {
    reads { ipv4.dst_addr: lpm; }
    actions { allow; deny; }
}

control ingress { apply(acl); }
```

What could possibly go wrong?
What if packet isn’t IPv4?
ipv4 header will be invalid

What goes wrong?
Table reads arbitrary values
→ Intended policy violated

Can read values from a previous packet
→ Side channel vulnerability!

Real programs are complicated:
validity is hard to keep track of in your head

Property #1: header validity
What if acl table misses?
The forwarding behavior is unspecified by the language

What goes wrong
How packets are forwarded becomes target dependent
• May not do what you expect!
• Code not portable across different targets

Property #2: egress determinacy
Let’s add 4in6 tunneling!

table tunnel_decap {
  ...
  actions { decap_4in6; } 
}

action decap_4in6() {
  copy_header(ipv4, inner_ipv4);
  remove_header(inner_ipv4);
}

table tunnel_term {
  ...
  actions { term_4in6; } 
}

action term_4in6() {
  remove_header(ipv6);
  modify_field(ethernet.ether_type, 0x0800);
}
Let’s add 4in6 tunneling!
A peek behind the curtain...

**PISA [SIGCOMM 2013]**
Protocol-Independent Switch Architecture

In PISA, state is copied verbatim from ingress to egress...
A peek behind the curtain...

PISA [SIGCOMM 2013]
Protocol-Independent Switch Architecture

Implementations often use a parser/deparser to bridge state
What if the architecture reparses the packet?

IPv4 and IPv6 are mutually exclusive in the parser

Property #3: reparseability
Another peek behind the curtain...

- Hardware devices have limited resources
- Compilers have options to improve utilization
  - Mutually exclusive headers can be overlaid in memory!
  - e.g., if headers are mutually exclusive in parser, they are likely to stay so in the rest of the program
What if headers share memory?
  • IPv4 and IPv6 might be overlaid

What goes wrong
  Data corruption!

Real-world parsers are complicated
Hard for programmers to keep track of mutually exclusive states

Property #4: mutual exclusion of headers
Types of Properties

General safety properties
- Header validity
- Arithmetic-overflow checking
- Index bounds checking (header stacks, registers, meters, …)

Architectural properties
- Unambiguous forwarding
- Reparseability
- Mutual exclusion of headers
- Correct metadata usage (e.g., read-only metadata)

Program-specific properties
- Custom assertions in P4 program — e.g., IPv4 ttl correctly decremented
Challenge #1: Imprecise Semantics

- Language specification doesn’t define a precise the semantics of every program
Challenge #1: Imprecise Semantics

- Language specification doesn’t define a precise the semantics of every program
- We defined a semantics for P4 by translation to GCL (an intermediate language)
Challenge #1: Imprecise Semantics

- Language specification doesn’t define a precise the semantics of every program
- We defined a semantics for P4 by translation to GCL (an intermediate language)
- Tested semantics
  - Symbolically executed GCL to generate input-output pairs for a suite of representative programs
Challenge #1: Imprecise Semantics

- Language specification doesn’t define a precise the semantics of every program
- We defined a semantics for P4 by translation to GCL (an intermediate language)
- Tested semantics
  - Symbolically executed GCL to generate input-output pairs for a suite of representative programs
  - Ran with Barefoot P4 compiler and Tofino simulator
Challenge #2: Control-Plane Model

A P4 program is really only “half” of a program

- Table rules are not statically known
- Populated by the control plane at run time

```plaintext
table acl {
    reads {
        ipv4.dstAddr: lpm;
    }
    actions { allow; deny; }
}
```

<table>
<thead>
<tr>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0.1.99</td>
<td>deny</td>
</tr>
<tr>
<td>*</td>
<td>accept</td>
</tr>
</tbody>
</table>
Challenge #2: Control-Plane Model

- A P4 program is really only “half” of a program
  - Table rules are not statically known
  - Populated by the control plane at run time

```p4
table acl {
  reads {
    ipv4.dstAddr: lpm;
  }
  actions { allow; deny; }
}
```

( @[Action] acl <hit> (allow); std_meta.egress_spec := 1) + ( @[Action] acl <hit> (deny); std_meta.egress_spec := 511) + @[Action] acl <miss>

Tables translated into unconstrained nondeterministic choice
Challenge #2: Control-Plane Model

- A P4 program is really only “half” of a program
  - Table rules are not statically known
  - Populated by the control plane at run time
- Control planes are carefully programmed
  - Tables rarely take arbitrary actions

```p4
table acl {
  reads {
    ipv4.dst_addr: lpm;
  }
  actions { allow; deny; } }
```

Tables translated into unconstrained nondeterministic choice

```p4
([Action] acl <hit> (allow);
  std_meta.egress_spec := 1)
+ ([Action] acl <hit> (deny);
  std_meta.egress_spec := 511)
+ @[ Action] acl <miss>
```
Challenge #2: Control-Plane Model

- A P4 program is really only “half” of a program
  - Table rules are not statically known
  - Populated by the control plane at run time
- Control planes are carefully programmed
  - Tables rarely take arbitrary actions
- To rule out false positives, need to somehow model behavior of control plane

```plaintext
table acl {
  reads {
    ipv4.dst_addr: lpm;
  }
  actions { allow; deny; }
}

(@[Action] acl <hit> (allow);
  std_meta.egress_spec := 1)
+ (@[Action] acl <hit> (deny);
  std_meta.egress_spec := 511)
+ @[ Action] acl <miss>
```

Tables translated into unconstrained nondeterministic choice
Symbolic Control-Plane

- Provided as second input to p4v
- Constrains choices made in tables
- Written in domain-specific syntax
Symbolic Control Plane

- Provided as second input to p4v
- Constrains choices made in tables
- Written in domain-specific syntax

Example

P4 Source Code

```p4
table tunnel_decap {
    ...
    actions { decap_4in6; }
}
table tunnel_term {
    ...
    actions { term_4in6; }
}
```

Symbolic Control Plane

```p4
assume
    action(tunnel_decap) == decap_4in6
iff
    action(tunnel_term) == term_4in6
```
Challenge #3: Annotation Burden

Many tools require programmers to annotate their code with assumptions and assertions.

p4v can automatically generate assertions for many properties using simple command-line flags.

Currently supported:
- Header validity
- Egress determinacy
- Reparseability
- All valid headers deparsed
- Expression definedness
- Index bounds
Challenge #4: Scaling Up

- Not using compositional verification
  - Needs annotations at component boundaries

- Not using symbolic execution
  - Exponential path explosion means explicitly path exploration intractable

- Instead, generate single verification condition
  - Valid iff program satisfies assertions on all execution paths
  - SMT solver can check or provide a counter-example

- Also apply standard optimizations
  - Loop unrolling
  - Constant folding
  - Constant propagation
  - Dead-code elimination
p4v Architecture

Control-Plane Interface → Source Program

GCL Program

Annotated → Optimized

Verification Condition

Z3

[Result] Passed

[Result] Failed
[Counterexample]
  [ Parser ] start
  [ Parser ] _parse_ethernet
  [ Packet ] ethernet.dst_addr = 0x000000000000
  [ Packet ] ethernet.src_addr = 0x000000000000
  [ Packet ] ethernet.ether_type = 0xf7ff
  [ Assert ] (not (= ipv4.valid 1w0))
1. Start with P4 Program
2. Annotate with assertions
3. Translate to imperative code
4. Apply standard optimizations
5. Generate first-order formula
6. Send to Z3 SMT Solver
7. Success or counterexample
   • Input packet
   • Program trace
   • Violated assertion
Evaluation: Header Validity

Graph showing verification time (s) for various systems with and without W/Interface.
Evaluation: switch.p4

Statistics

- 5,599 LoC
- 58 parser states
- 120 match–action tables

Control-plane interface

- 758 LoC
- ~2 days’ programmer effort
- Default actions (31)
- Fabric wellformedness (14)
- Table actions (66)
- Guarded reads (10)
- Action data (14)

Found 10 bugs

- Parser bugs (2)
- Action flaws (4)
- Infeasible control-plane (3)
- Invalid table read (1)
Network-Wide Verification

ProbNetKAT [POPL ’17, PLDI ’19]
From Devices to Networks

Network topology:

Network config:
shortest path

“Will my packet get delivered?”

Verification Tool

✘

✔
But What About Failures?

Network topology:

```
[Network topology diagram showing SRC, intermediate node, and DST nodes connected by arrows.]
```

Network config:
shortest path

Failure model:
links fail iid with prob 1%

“Will my packet get delivered?”

Verification Tool

[Image of a person with a question mark and hands raised in confusion.]
A language for modeling & reasoning about networks probabilistically.

Prob + NetKAT

probabilistic primitive: \( p \oplus_r q \)

network primitives: \( f:=n, \text{dup} \)
A language for modeling & reasoning about networks probabilistically.

ProbNetKAT

NetKAT

KAT

Prob

Net

KA

network primitives

regular expressions

boolean tests

probabilistic primitives

f:=n, dup

+,

p ⊕ r q

f=n, dup

+,

f=n
Probabilistic Verification

**Network topology:**

```
  SRC  ---  DST
```

**Network config:**
shortest path

**Failure model:**
links fail iid with prob 1%

```
Will my packet get delivered?
```

McNetKAT \\
\[ w.p. 98.01\% \]
Probabilistic Verification

Network topology:

- SRC
- DST

Network config:
- shortest path
- + detour unif. at random

Failure model:
- links fail iid with prob 1%
- + at most 2 failures

McNetKAT

"Will my packet get delivered?"
Probabilistic Verification

Network topology:

Network config:
- shortest path
- + detour unif. at random

Failure model:
- links fail iid with prob 1%
- + at most 2 failures

McNetKAT

“Will my packet get delivered?”

w.p. 1
Challenge: Infinite Paths

Network config: shortest path + detour unif. at random

Failure model: links fail iid with prob 1% + at most 2 failures
Challenge: Infinite Paths

Network config:
shortest path
+ detour unif. at random

Failure model:
links fail iid with prob 1%
+ at most 2 failures

delivery w.p. 1 despite infinite loop?
Challenge Infinite Path

Network config:
shortest path
+ detour unif. at random

Failure model:
links fail iid with prob 1%
+ at most 2 failures

infinite loop has probability 0
Semantics via Markov Chains

Insight: ProbNetKAT programs describe random functions
Insight: ProbNetKAT programs describe random functions.

These functions can be modeled as **Markov chains**:
- states are packet sets: $S = 2^{P_k}$
- chain is given by transition matrix $M \in [0,1]^{S \times S}$
- $M(a, b) = \text{probability of producing output } b \text{ on input } a$
- Compute fixed points using theory of absorbing Markov chains
McNetKAT Implementation

Insight: can exploit sparse structure of network programs

Code:

```plaintext
if pt=1 then
  pt ← 2 ⊕ 0.5 pt ← 3
else if pt=2 then
  pt ← 1
else if pt=3 then
  pt ← 1
else
  drop
```

Network model → Compile → Symbolic IR → Convert → Sparse matrix → Solve
Transition matrix models the input-output behavior and can be analyzed using standard techniques.
F10: A Fault-Tolerant Engineered Network
Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson

*University of Washington*
Motivation

• short-term failures in data centers are common
Case Study

F10: A Fault-Tolerant Engineered Network
Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson

University of Washington

Motivation

• short-term failures in data centers are common
• application performance suffers
Case Study

F10: A Fault-Tolerant Engineered Network
Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson
University of Washington

Motivation

- short-term failures in data centers are common
- application performance suffers
- despite 1:1 redundancy!
**Case Study**

**F10: A Fault-Tolerant Engineered Network**
Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson  
*University of Washington*

**Motivation**

- Short-term failures in data centers are common
- Application performance suffers
- Despite 1:1 redundancy!
Case Study

F10: A Fault-Tolerant Engineered Network
Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson
University of Washington

Motivation

- short-term failures in data centers are common
- application performance suffers
- despite 1:1 redundancy!

Solution
Case Study

F10: A Fault-Tolerant Engineered Network
Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson
University of Washington

Motivation

- short-term failures in data centers are common
- application performance suffers
- despite 1:1 redundancy!

Solution

- detect failures of neighboring links & switches...
Case Study

F10: A Fault-Tolerant Engineered Network
Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, Thomas Anderson
University of Washington

Motivation

- short-term failures in data centers are common
- application performance suffers
- despite 1:1 redundancy!

Solution

- detect failures of neighboring links & switches...
- ...and route around them
An ABFatTree is much like a regular FatTree
An ABFatTree is much like a regular FatTree.
Case Study: Topology

An ABFatTree is much like a regular FatTree
Case Study: Topology

An ABFatTree is much like a regular FatTree
Case Study: Topology

An ABFatTree is much like a regular FatTree

But it provides shorter detours around failures
An ABFatTree is much like a regular FatTree

But it provides shorter detours around failures
Case Study: Topology

An ABFatTree is much like a regular FatTree

But it provides shorter detours around failures
Case Study: Routing Schemes

We implemented F10 as a series of refinements:

- F10\(_0\)
- F10\(_3\)
- F10\(_3,5\)
Case Study: Routing Schemes

We implemented F10 as a series of refinements:

- **F10₀**: Shortest path routing
- **F10₃**
- **F10₃,₅**
Case Study: Routing Schemes

We implemented F10 as a series of refinements.

- **F10₀**: shortest path routing
- **F10₃**: F10₀ + 3-hop rerouting
- **F10₃,₅**
We implemented F10 as a series of refinements:

- **F10₀**: shortest path routing
- **F10₃**: F10₀ + 3-hop rerouting
- **F10₃,₅**: F10₃ + 5-hop rerouting
Case Study: Routing Schemes

We implemented F10 as a series of refinements:

- **F10₀**: Shortest path routing
- **F10₃**: F10₀ + 3-hop rerouting
- **F10₃,5**: F10₃ + 5-hop rerouting

Requires per-packet state
Case Study: k-resilience

We verified k-resilience using ProbNetKAT
We verified k-resilience using ProbNetKAT

<table>
<thead>
<tr>
<th>$k$</th>
<th>F10$_0$</th>
<th>F10$_3$</th>
<th>F10$_{3,5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1</td>
<td>✕</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2</td>
<td>✕</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>✕</td>
<td>✕</td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td>✕</td>
<td>✕</td>
<td>✓</td>
</tr>
<tr>
<td>$\infty$</td>
<td>✕</td>
<td>✕</td>
<td>✓</td>
</tr>
</tbody>
</table>

$k$ = number of failures  ✔ = 100% packet delivery
Case Study: k-resilience

We verified k-resilience using ProbNetKAT

<table>
<thead>
<tr>
<th>$k$</th>
<th>$F10_0$</th>
<th>$F10_3$</th>
<th>$F10_{3,5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>1</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>2</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>3</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>4</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>∞</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

$k$ = number of failures  
✓ = 100% packet delivery
Case Study: k-resilience

We verified k-resilience using ProbNetKAT

<table>
<thead>
<tr>
<th>k</th>
<th>F10₀</th>
<th>F10₃</th>
<th>F10₃,₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>✗</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>✗</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sophistication of Routing Scheme

An uninitialized flag caused all packets to be dropped!

k = number of failures  ✔️ = 100% packet delivery
## Case Study: k-resilience

After fixing the bug...

<table>
<thead>
<tr>
<th>$k$</th>
<th>$F10_0$</th>
<th>$F10_3$</th>
<th>$F10_{3,5}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>1</td>
<td>❌</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>2</td>
<td>❌</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>3</td>
<td>❌</td>
<td>❌</td>
<td>✔️</td>
</tr>
<tr>
<td>4</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>$\infty$</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
</tbody>
</table>

$k =$ number of failures  ✔️ = 100% packet delivery
Case Study: Delivery Probability

We analyzed packet loss when link failures increase

![Graph showing delivery probability vs. link failure probability](image)

Can quantify improvements when using rerouting
Case Study: Delivery Probability

We analyzed packet loss when link failures increase

Near-perfect packet delivery despite extreme failure model!

Can quantify improvements when using rerouting
Case Study: Expected Path Stretch

The price of resilience: increased paths lengths

AB FatTree outperforms regular FatTree
Case Study: Expected Path Stretch

The price of resilience: increased paths lengths

Figure 10 shows the expected hop-count of paths taken by packets conditioned on their delivery. Both F10\textsuperscript{3} and F10\textsuperscript{3,5} deliver packets with high probability even at high failure probabilities, as we saw in Figure 8. However, a higher probability of link-failure implies that it becomes more likely for these schemes to invoke rerouting, which increases hop count. Hence, we see the increase in expected hop-count as failure probability increases. F10\textsuperscript{3,5} uses 5-hop rerouting to achieve more resilience compared to F10\textsuperscript{3}, which performs only 3-hop rerouting, and this leads to slightly higher expected hop-count for F10\textsuperscript{3,5}. We see that the increase is more significant for FatTree in contrast to AB FatTree because FatTree only supports 5-hop rerouting.

As the failure probability increases, the probability of delivery for packets that are routed via the core layer decreases significantly for F10\textsuperscript{0} (recall Figure 8). Thus, the distribution of delivered packets shifts towards those with direct 2-hop path via an aggregation switch (such as packets from $s_2$ to $s_1$), and hence the expected hop-count decreases slightly.

Regular FatTree only has long backup paths

ABFatTree outperforms regular FatTree
Evaluation: Scalability

McNetKAT scales to networks of realistic size

> 1000 nodes
Semantically, this construct is equivalent to a cascade of conditionals; but the native backend compiles it in parallel using a map-reduce-style strategy, using one process per core by default.

To evaluate the impact of parallelization, we compiled two representative FatTree models ($p = 14$ and $p = 16$) using ECMP routing on an increasing number of cores. With $m$ cores, we used one master machine together with $r = \lceil m/16 - 1 \rceil$ remote machines, adding machines one by one as needed to obtain more physical cores. The results are shown in Figure 8. We see near linear speedup on a single machine, cutting execution time by more than an order of magnitude on our 16-core test machine. Beyond a single machine, the speedup depends on the complexity of the submodels for each switch—the longer it takes to generate the matrix for each switch, the higher the speedup. For example, with a $p = 16$ FatTree, we obtained a 30x speedup using 40 cores across 3 machines.

Comparison with other tools. Bayonet [15] is a state-of-the-art tool for analyzing probabilistic networks. Whereas McNetKAT has a native backend tailored to the networking domain and a backend based on a probabilistic model checker, Bayonet programs are translated to a general-purpose probabilistic language which is then analyzed by the symbolic inference engine PSI [16]. Bayonet's approach is more general, as it can model queues, state, and multi-packet interactions under an asynchronous scheduling model. It also supports Bayesian inference and parameter synthesis. Moreover, Bayonet is fully symbolic whereas McNetKAT uses a numerical linear algebra solver [7] (based on floating point arithmetic) to compute limits.

To evaluate how the performance of these approaches compares, we reproduced an experiment from the Bayonet paper that analyzes the reliability of a simple routing scheme in a family of “chain” topologies indexed by $k$, as shown in Figure 9. For $k = 1$, the network consists of four switches organized into a diamond, with a single link that fails with probability $p_{\text{fail}} = 1/1000$. For $k > 1$, the network consists of $k$ diamonds linked together into a chain as shown in Figure 9. Within each diamond, switch $S_0$ forwards packets with equal probability to switches $S_1$ and $S_2$, which in turn forward to switch $S_3$. However, $S_2$ drops the packet if the link to $S_3$ fails. We analyze the probability that a packet originating at $H_1$ is successfully delivered to $H_2$. Our implementation does not exploit the regularity of these topologies.

Figure 10 gives the running time for several tools on this benchmark: Bayonet, hand-written PRISM, ProbNetKAT with the PRISM backend (PPNK), and ProbNetKAT with the native backend (PNK). Further, we ran the PRISM tools in exact and approximate mode, and we ran the ProbNetKAT backend on a single machine and on the cluster. Note that both axes in the plot are log-scaled. We see that Bayonet scales to 32 switches in about 25 minutes, before hitting the one hour time limit and 64 GB memory limit at 48 switches. ProbNetKAT answers the same query for 2048 switches in under 10 seconds and scales to over 65000 switches in about 50 minutes on a single core, or just 2.5 minutes using a cluster of 24 machines. PRISM scales similarly to ProbNetKAT, and performs best using the hand-written model in approximate mode.

Overall, this experiment shows that for basic network verification tasks, ProbNetKAT’s domain-specific backend based on specialized data structures and an optimized linear-algebra library [7] can outperform an approach based on a general-purpose solver.
Guarded Kleene Algebra with Tests
Verification of Uninterpreted Programs in Nearly Linear Time

STEFFEN SMOLKA, Cornell University, USA
NATE FOSTER, Cornell University, USA
JUSTIN HSU, University of Wisconsin–Madison, USA
TOBIAS KAPPÉ, University College London, UK
DEXTER KOZEN, Cornell University, USA
ALEXANDRA SILVA, University College London, UK

Guarded Kleene Algebra with Tests (GKAT) is a variation on Kleene Algebra with Tests (KAT) that arises by restricting the union (+) and iteration (∗) operations from KAT to predicate-guarded versions. We develop the (co)algebraic theory of GKAT and show how it can be efficiently used to reason about imperative programs. In contrast to KAT, whose equational theory is PSPACE-complete, we show that the equational theory of GKAT is (almost) linear time. We also provide a full Kleene theorem and prove completeness for an analogue of Salomaa’s axiomatization of Kleene Algebra.

This paper is dedicated to Laurie J. Hendren (1958–2019), whose passion for teaching and research inspired us and many others. Laurie’s early work on McCAT [Erosa and Hendren 1994] helped us understand the limits of Guarded Kleene Algebra with Tests and devise a suitable definition of well-nestedness that underpins our main results.

CCS Concepts: • Theory of computation → Program schemes; Program reasoning.

Additional Key Words and Phrases: uninterpreted programs, program equivalence, program schemes, guarded automata, coalgebra, Kleene algebra with Tests
Future Directions
Open Problems
Open Problems

Distributed Control

Open Problems

Distributed Control

Mutable State and Concurrency
VNM [NSDI ’17], Stateful NetKAT [PLDI ’15]
Open Problems

Distributed Control

Mutable State and Concurrency
VNM [NSDI ’17], Stateful NetKAT [PLDI ’15]

Scheduling and Queueing
UPS [NSDI ’16], PIFO [SIGCOMM ’16]
Open Problems

Distributed Control

Mutable State and Concurrency
VNM [NSDI ’17], Stateful NetKAT [PLDI ’15]

Scheduling and Queueing
UPS [NSDI ’16], PIFO [SIGCOMM ’16]

Hybrid Deployments
Panopticon [ATC ’14], Fibbing [SIGCOMM ’15]
Conclusion

• Many opportunities for bridging between networking and other areas of computing
• Classic problems have been reopened, and new ones are rapidly emerging
• A rare chance to re-think the foundations of networking from the ground up!
Thank You!

http://www.cs.cornell.edu/~jnfoster