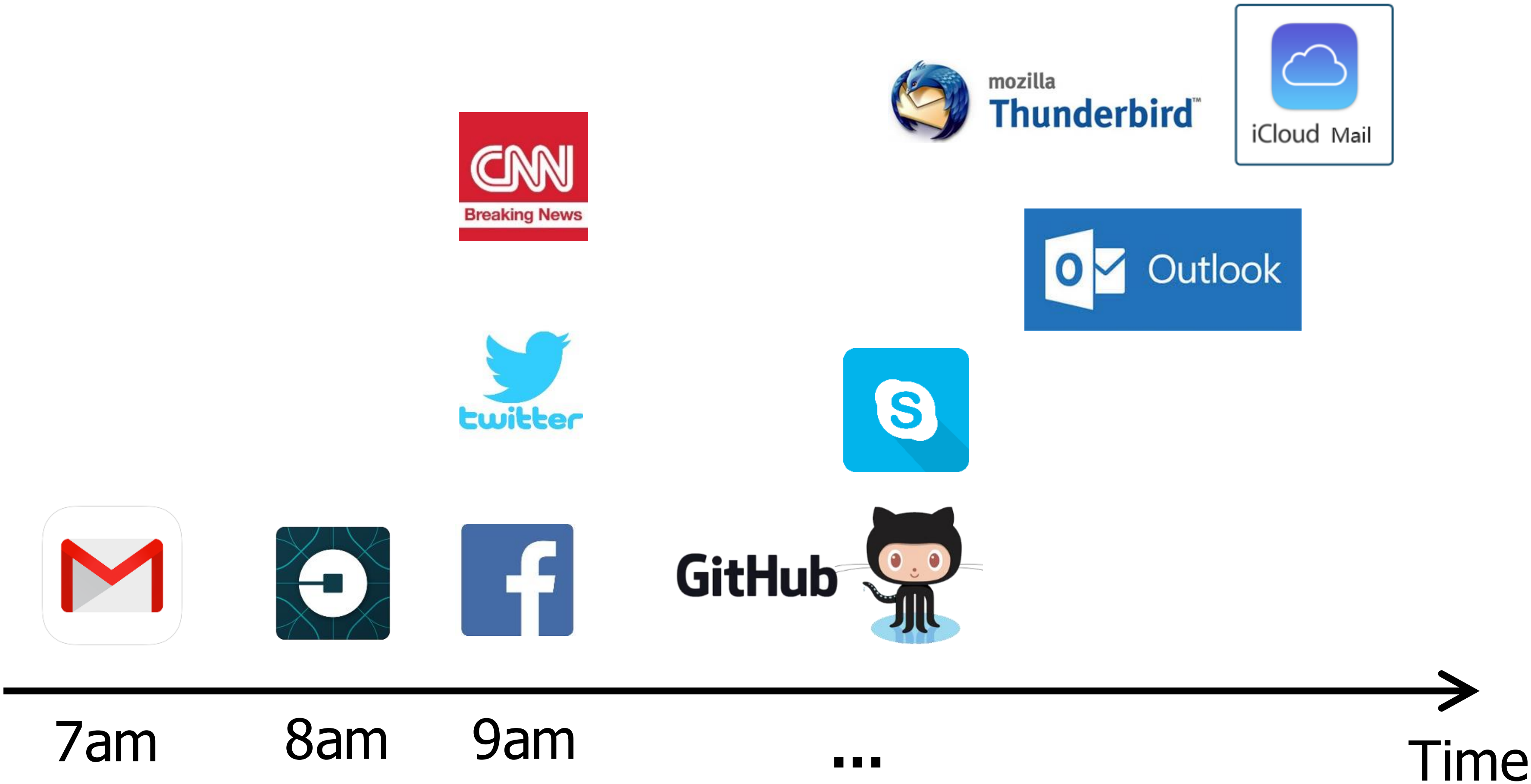
Programmable In-network Security

A vision for network security in the next generation

Ang Chen

University of Michigan

The modern life runs on network services



- We rely on network services every day!

However, networks are plagued with attacks

'Internet Of Things' hacking a
wide

AWS outage knocks Amazon, Netf
Tinder and IMDb in MEGA data c

SCIENTIFIC AMERICAN

Power grid cyber attacks keep the Pentagon up at night

The New York Times

Cyberattacks: Stuxnet and Flame

Forbes

Amazon.com goes
$66,240 per minute

ZDNet

Global DNS outage hits Microsoft Azure customers

wns cost countries $2.4
billion last year

Skype calling knocked offline by

USA TODAY

Hacked home devices caused
massive Internet outage

CRN

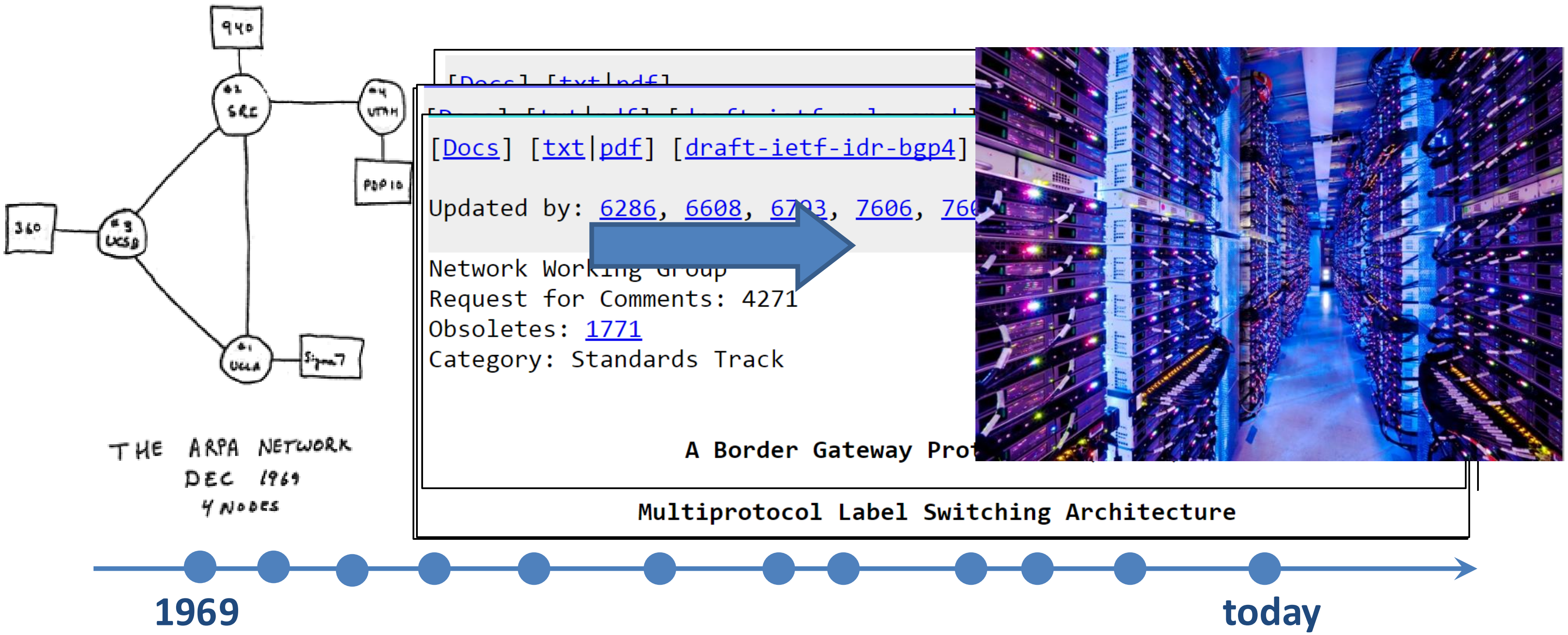
Azure Nightmare: Cust
Intermittent Outage

BUSINESS INSIDER

'Lizard Squad' took down Xbox Live and PlayStation network

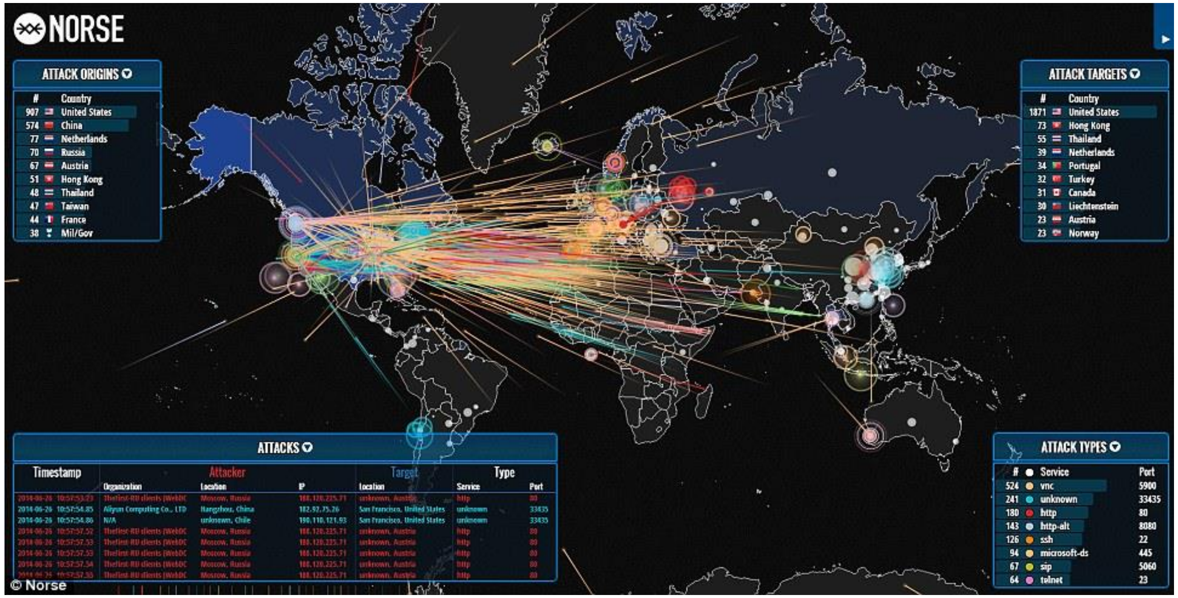
'Maintenance' caused
nationwide internet outage

How did we get here?

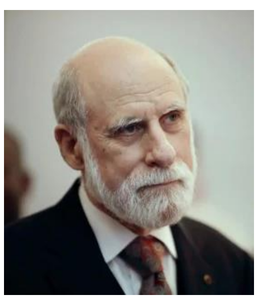


- Networks have evolved significantly over the years

Challenge: network security



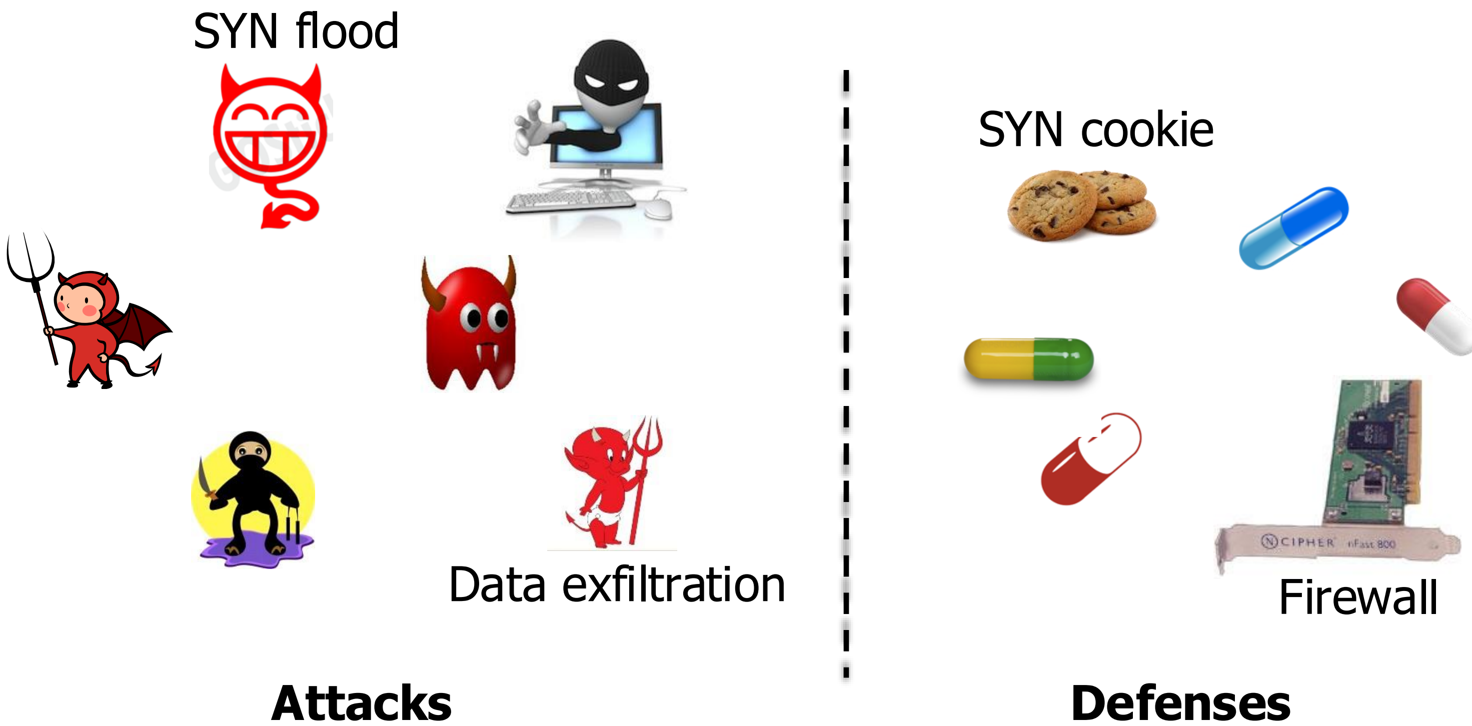
- Our ability to secure them hasn't caught up.
  - The network infrastructure itself doesn't have security support



"The Internet is brittle and fragile and too easy to take down. It's a conduit for criminal activity."

--- Vint Cerf

Today's approach 1: Bolt-on protection



- Keep networks unchanged, deploy protections elsewhere
  - E.g., middleboxes, end host software, ..
- Advantage: Immediately practical 
- Disadvantage: Not a fundamental solution 

Today's approach 2: Clean slate design

Internet re-imagined!

- Redesign the Internet to be secure
  - i.e., Future Internet Architectures
- Advantage: Addresses the root cause
- Disadvantage: Expensive changes

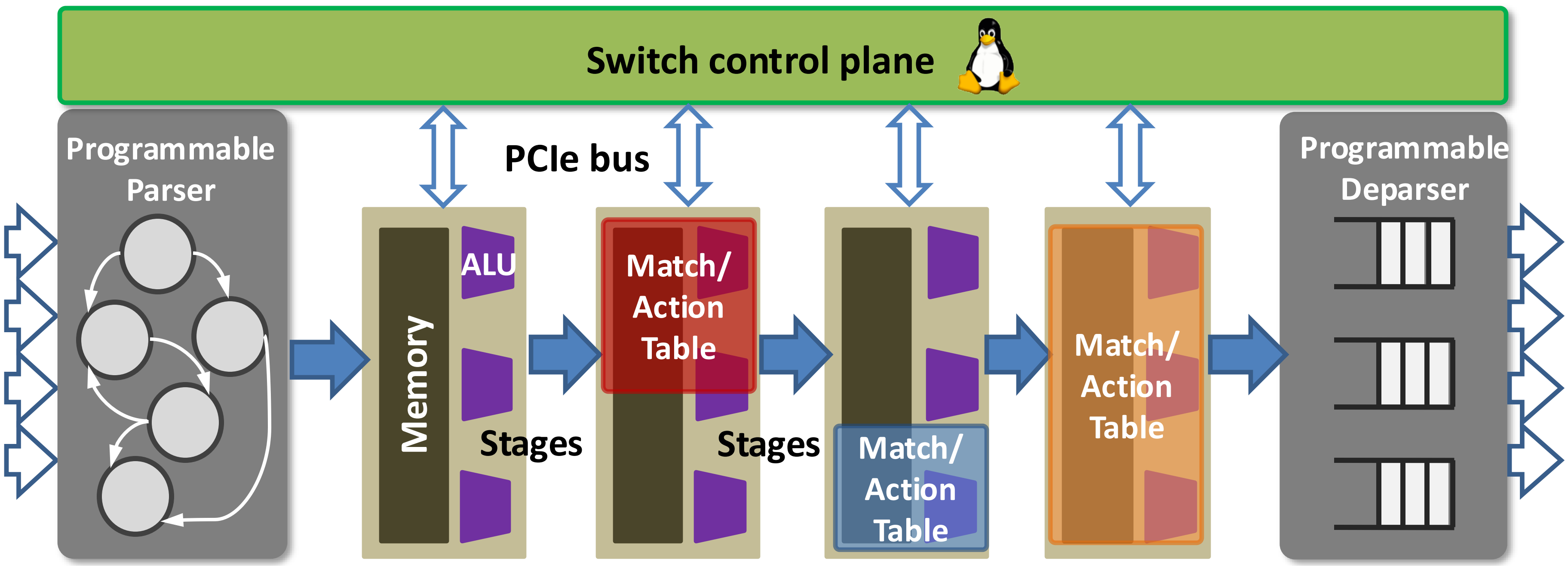
The best of both worlds?

Built-in security

Compatibility

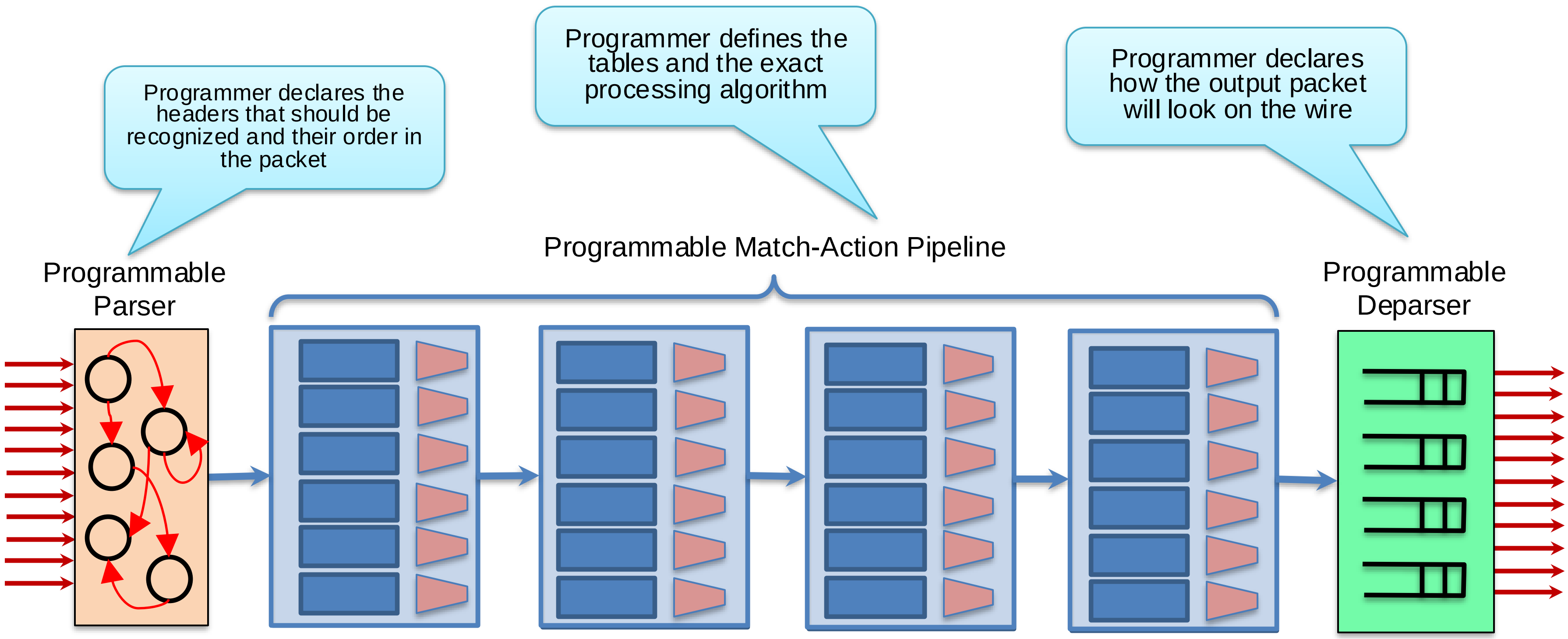
- Programmable In-network Security (Poise)
  - Architect security back to the network core
  - Without making intrusive modifications

Opportunity: Network programmability



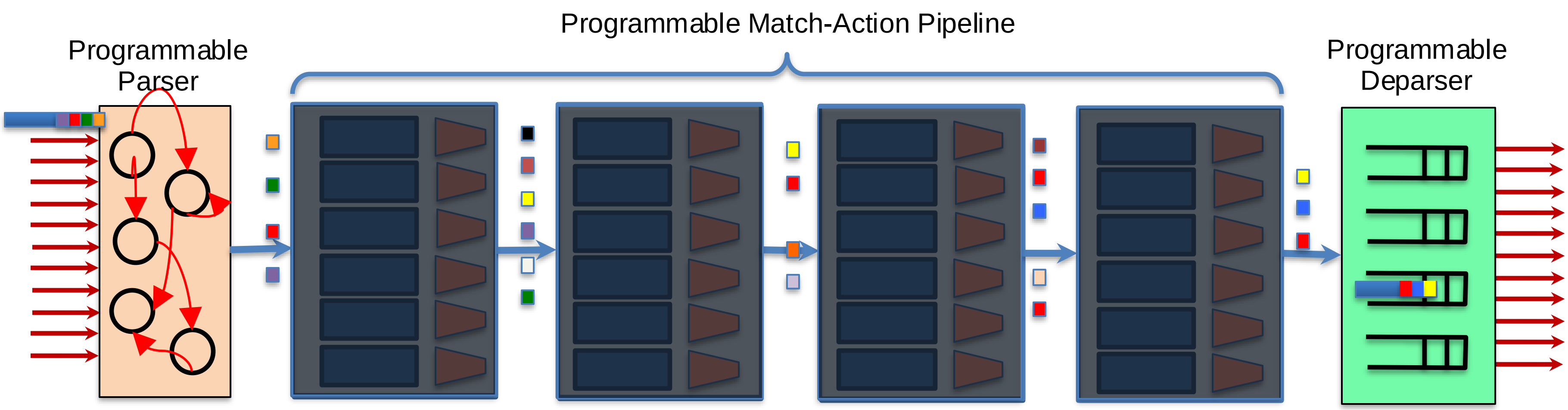
- Programmable switches
  - Controlled in high-level languages (e.g., P4)
  - Customized protocols, persistent state, custom match/actions
  - All at linespeed (Tbps)!
- We can develop defenses directly in the network!

Programming Protocol-independent Packet Processors



P4 in Action

- Packet is parsed into individual headers (parsed representation)
- Headers and intermediate results can be used for matching and actions
- Headers can be modified, added or removed
- Packet is deparsed (serialized)



V1Model Standard Metadata

```
struct standard_metadata_t {
   bit<9>  ingress_port;
   bit<9>  egress_spec;
   bit<9>  egress_port;
   bit<32> clone_spec;
   bit<32> instance_type;
   bit<1>  drop;
   bit<16> recirculate_port;
   bit<32> packet_length;
   bit<32> enq_timestamp;
   bit<19> enq_qdepth;
   bit<32> deq_timedelta;
   bit<19> deq_qdepth;
   bit<48> ingress_global_timestamp;
   bit<32> lf_field_list;
   bit<16> mcast_grp;
   bit<1>  resubmit_flag;
   bit<16> egress_rid;
   bit<1>  checksum_error;
}
```

- **`ingress_port`** - the port on which the packet arrived
- **`egress_spec`** - the port to which the packet should be sent to

$P4_{16}$ Program Template (V1Model)

```
#include <core.p4>
#include <v1model.p4>
/* HEADERS */
struct metadata { ... }
struct headers {
  ethernet_t   ethernet;
  ipv4_t       ipv4;
}
/* PARSER */
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t smeta) {
  ...
}
/* CHECKSUM VERIFICATION */
control MyVerifyChecksum(in headers hdr,
                         inout metadata meta) {
   ...
}
/* INGRESS PROCESSING */
control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t std_meta) {
  ...
}
```

```
/* EGRESS PROCESSING */
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t std_meta) {
  ...
}
/* CHECKSUM UPDATE */
control MyComputeChecksum(inout headers hdr,
                          inout metadata meta) {
  ...
}
/* DEPARSER */
control MyDeparser(inout headers hdr,
                   inout metadata meta) {
  ...
}
/* SWITCH */
V1Switch(
  MyParser(),
  MyVerifyChecksum(),
  MyIngress(),
  MyEgress(),
  MyComputeChecksum(),
  MyDeparser()
) main;
```

P4$_{16}$ Hello World (V1Model)

```
#include <core.p4>
#include <v1model.p4>
struct metadata {}
struct headers {}

parser MyParser(packet_in packet,
   out headers hdr,
   inout metadata meta,
   inout standard_metadata_t standard_metadata) {

    state start { transition accept; }
}

control MyVerifyChecksum(inout headers hdr, inout metadata
meta) {   apply {  }   }

control MyIngress(inout headers hdr,
   inout metadata meta,
   inout standard_metadata_t standard_metadata) {
apply {
        if (standard_metadata.ingress_port == 1) {
            standard_metadata.egress_spec = 2;
        } else if (standard_metadata.ingress_port == 2) {
            standard_metadata.egress_spec = 1;
        }
    }
}
```

```
control MyEgress(inout headers hdr,
   inout metadata meta,
   inout standard_metadata_t standard_metadata) {
    apply {  }
}

control MyComputeChecksum(inout headers hdr, inout metadata
meta) {
     apply { }
}

control MyDeparser(packet_out packet, in headers hdr) {
    apply { }
}

V1Switch(
   MyParser(),
   MyVerifyChecksum(),
   MyIngress(),
   MyEgress(),
   MyComputeChecksum(),
   MyDeparser()
) main;
```

P4$_{16}$ Hello World (V1Model)

```
#include <core.p4>
#include <v1model.p4>
struct metadata {}
struct headers {}

parser MyParser(packet_in packet, out headers hdr,
   inout metadata meta,
   inout standard_metadata_t standard_metadata) {
    state start { transition accept; }
}

control MyIngress(inout headers hdr, inout metadata meta,
   inout standard_metadata_t standard_metadata) {
    action set_egress_spec(bit<9> port) {
        standard_metadata.egress_spec = port;
    }
    table forward {
        key = { standard_metadata.ingress_port: exact; }
        actions = {
            set_egress_spec;
            NoAction;
        }
        size = 1024;
        default_action = NoAction();
    }
    apply {   forward.apply();    }
}
```

```
control MyEgress(inout headers hdr,
   inout metadata meta,
   inout standard_metadata_t standard_metadata) {
    apply {  }
}

control MyVerifyChecksum(inout headers hdr, inout metadata
meta) {   apply { }    }

control MyComputeChecksum(inout headers hdr, inout metadata
meta) {   apply { }    }

control MyDeparser(packet_out packet, in headers hdr) {
    apply { }
}

V1Switch( MyParser(), MyVerifyChecksum(), MyIngress(),
MyEgress(), MyComputeChecksum(), MyDeparser() ) main;
```

Key	Action ID	Action Data
1	set_egress_spec ID	2
2	set_egress_spec ID	1

Poise vs. today's software defenses

Three key advantages

- Per-packet visibility
  - Can detect needle-in-a-haystack attacks
- Per-packet dynamicity
  - Can respond as fast as dynamic changing attacks
- Scale-free defense
  - Naturally scales with network size and speed

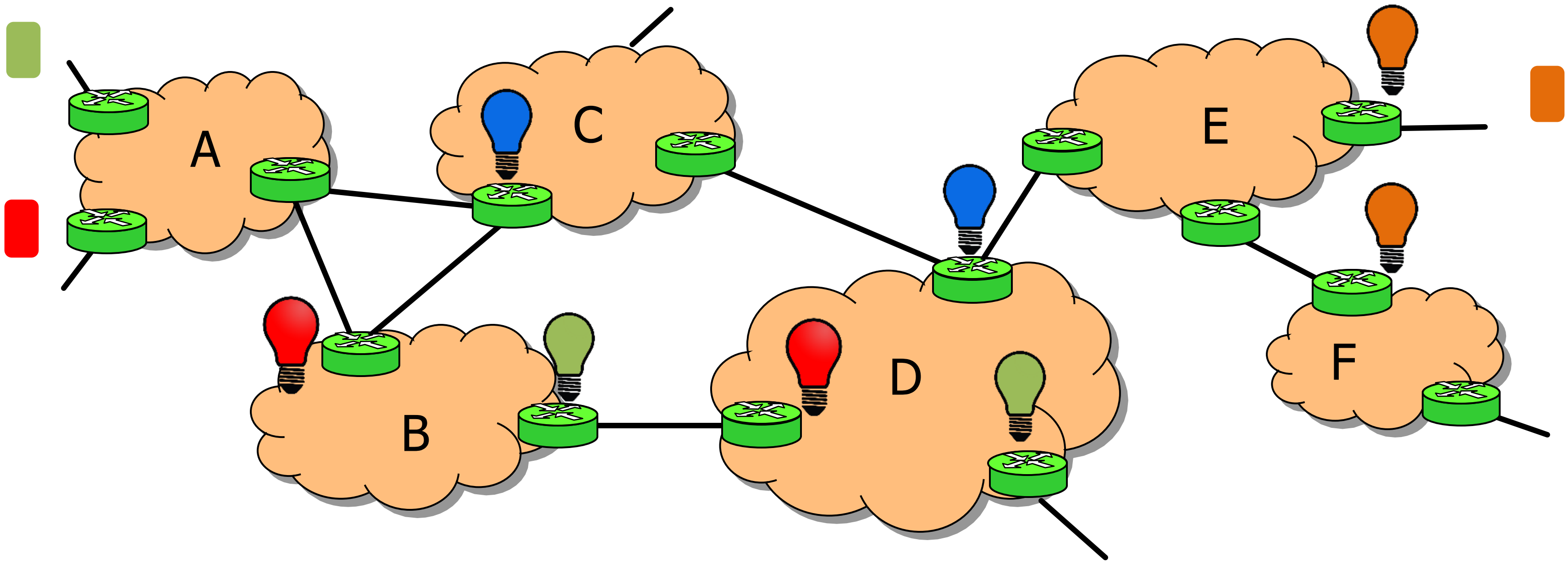
- At the same time, compatible with legacy networking

Roadmap: The switch as a defense platform

Security "apps"

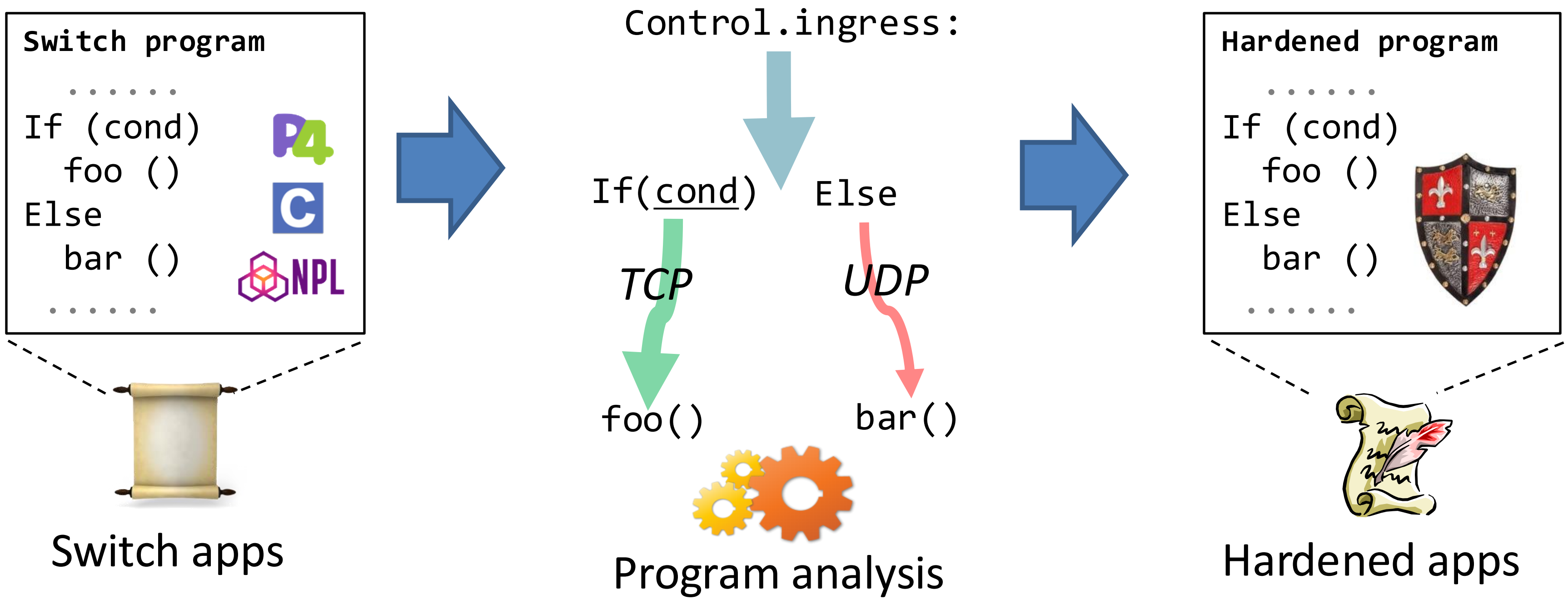
- Step 1: Design a range of switch-based security defenses
  - Switches dynamically activate the needed defenses
  - Progress so far:
    - Data exfiltration via covert channels (USENIX Sec'20)
    - Access control, RDMA security (USENIX Sec'20+'22)..

Roadmap: The network as a defense fleet



- Architect a wide range of defenses into network paths
  - Networks mitigate attacks as they route traffic
  - Progress so far: Link flooding attacks (USENIX Sec'21)

Roadmap: Securing the defenses

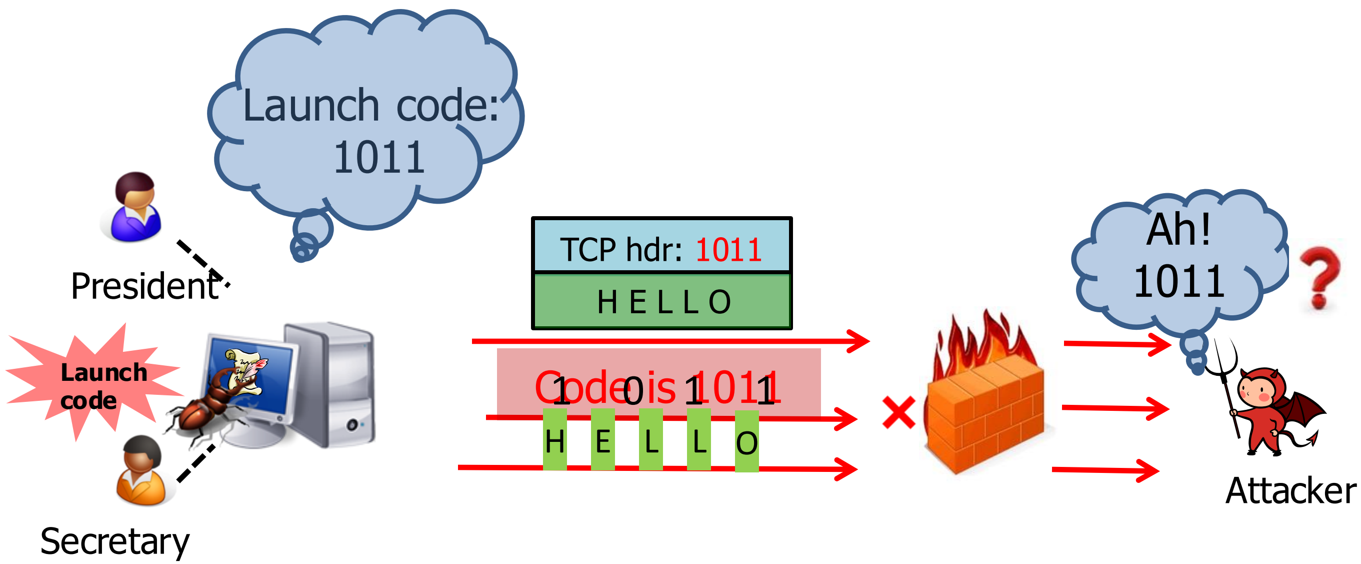


- Ensure that security apps are themselves secure
  - Program analyses to gain high assurance
  - Progress so far: Adversarial trace discovery (ASPLOS'21)

Outline

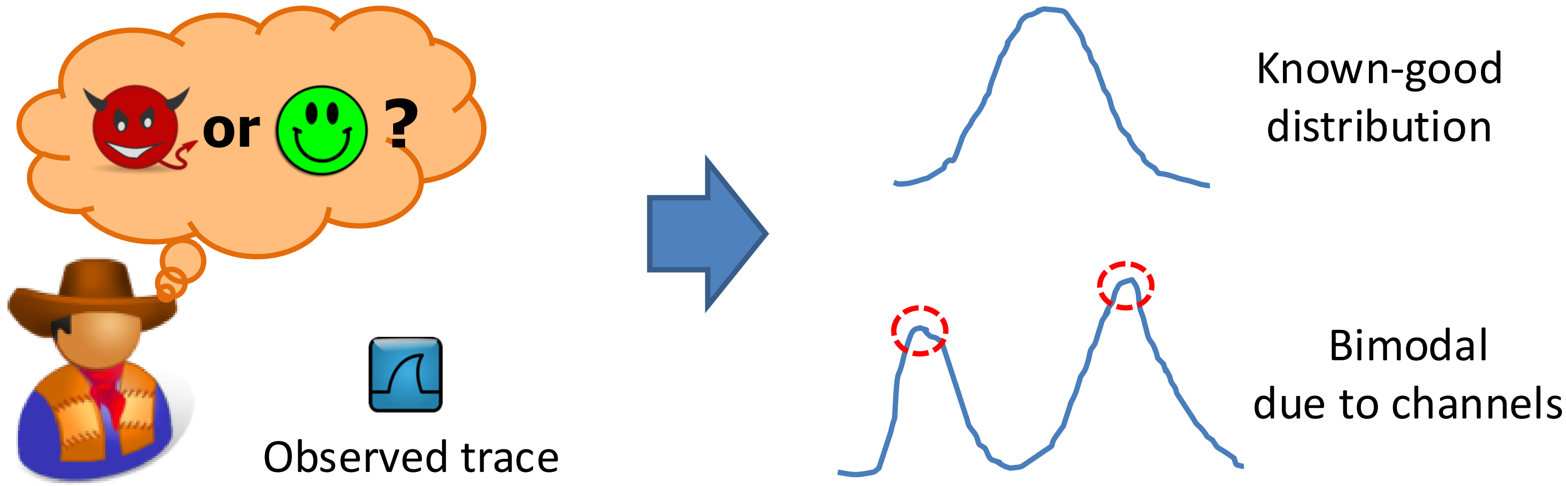
- Programmable In-network Security
  - The switch as a defense platform
  - Securing the switch apps
  - The network as a defense fleet
- Future work
- Summary

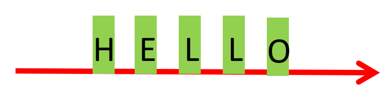
Motivation: Mitigating network covert channels



- Network covert channels:
  - Storage channels: changing the packet **header fields**.
    - E.g., TCP ISN (1997), TTL (2004), Partial ACK (2009)
  - Timing channels: changing the **timing** of packets
    - E.g., IP-layer (2004), TCP-layer (2008), PHY-layer (2014)

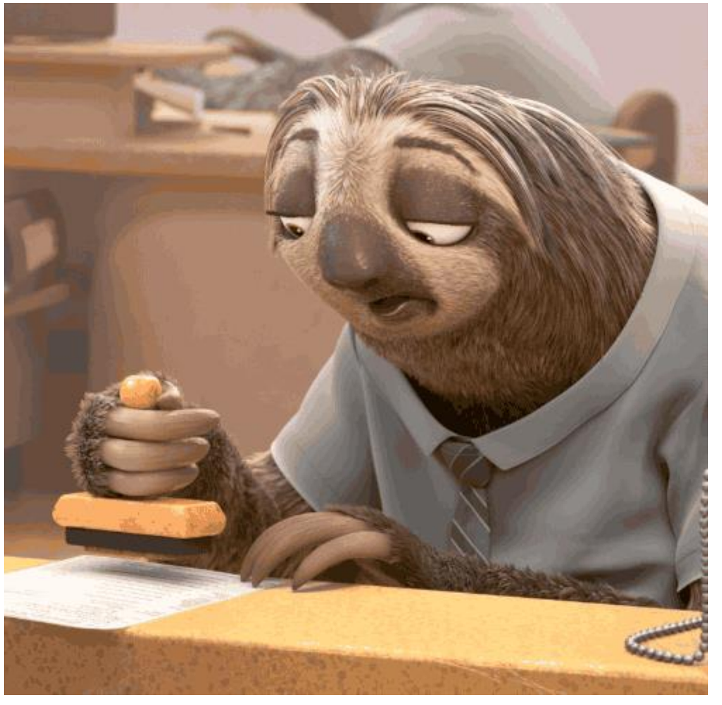
Detection algorithms are never perfect



- Consider network timing channels 

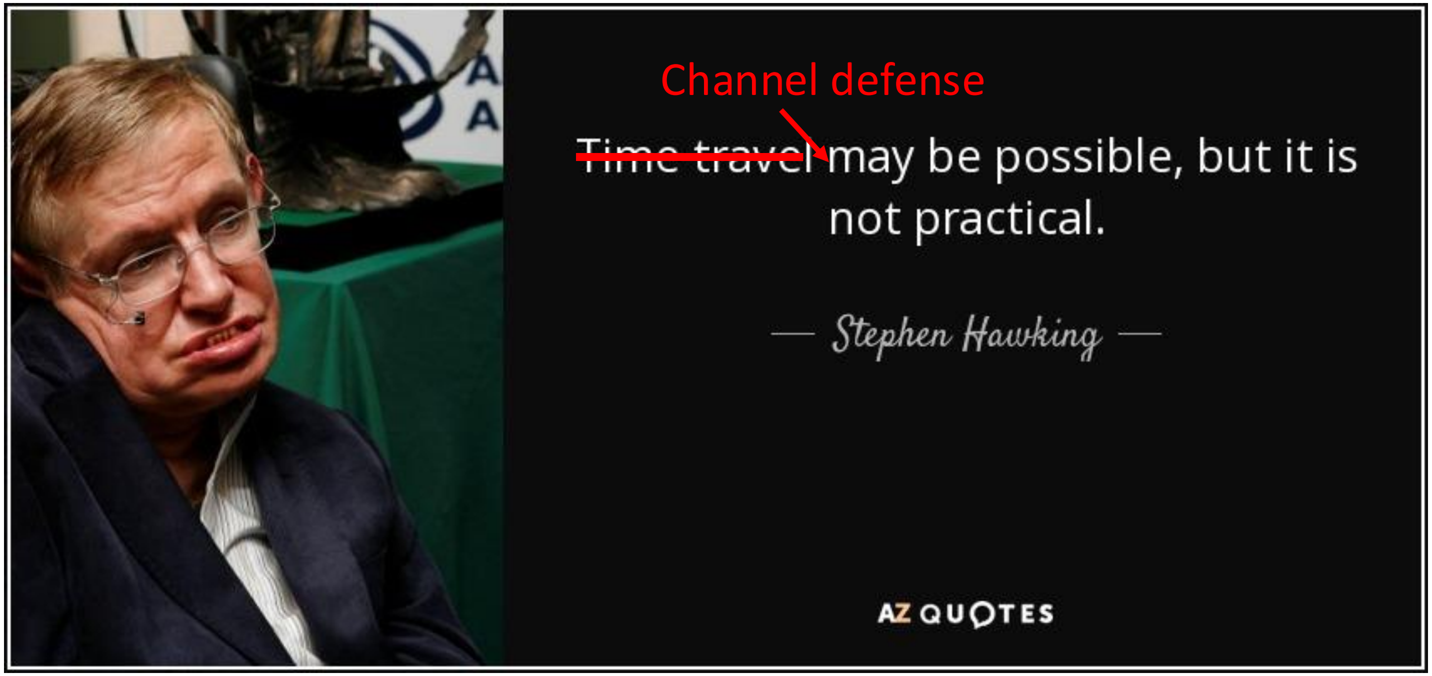
  - Packet timing is non-deterministic
  - E.g., due to congestion, server processing speeds, ..
- Approach: statistical tests against known-good traces
  - Can result in false positives/negatives

Mitigation algorithms hurt performance



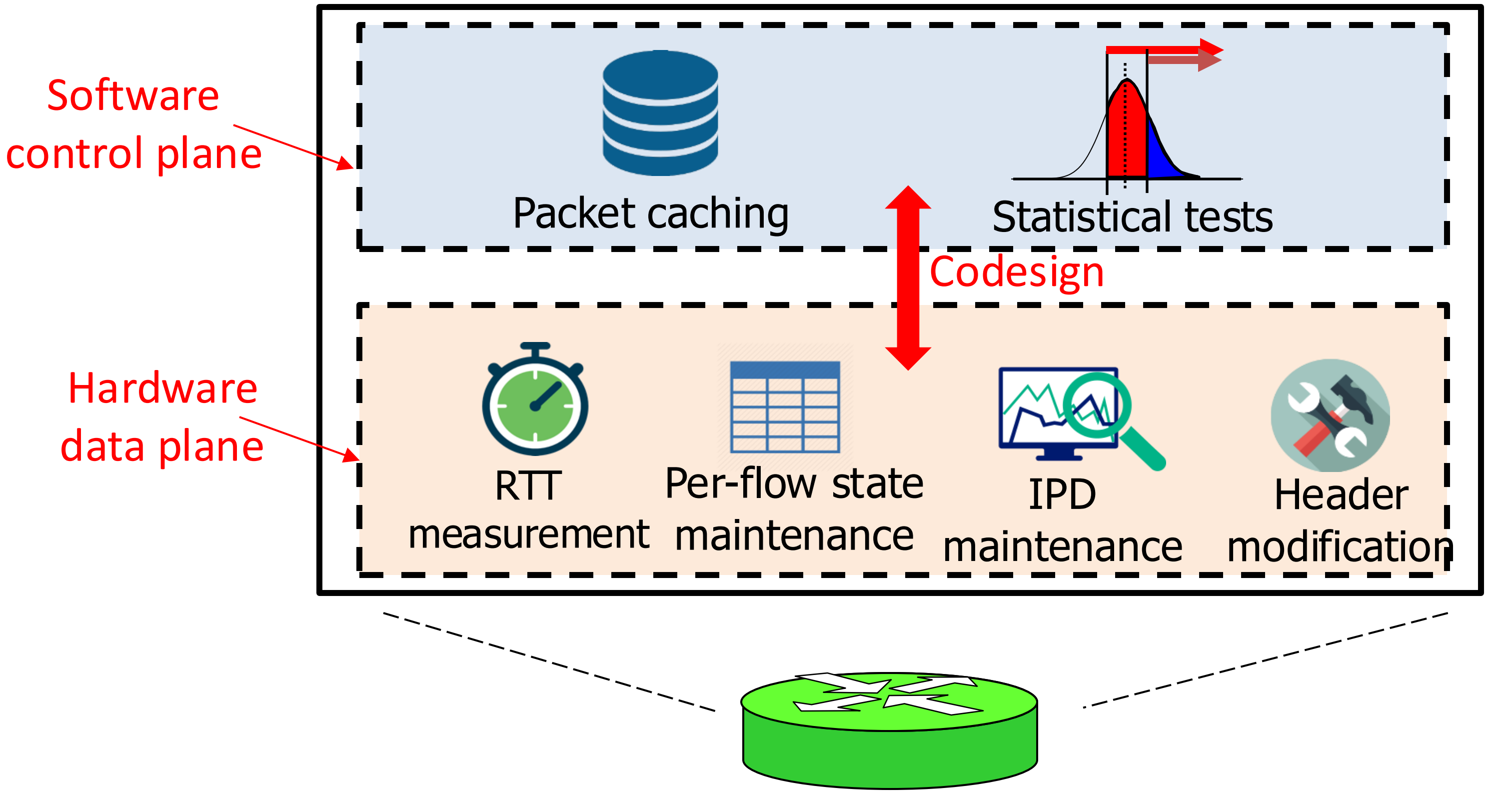
- Timing channel mitigation
  - E.g., Randomly delay packets to destroy timing modulation
- Storage channel mitigation
  - E.g., scrub header fields by adding offset to TCP SEQs

Implication: Performance penalty



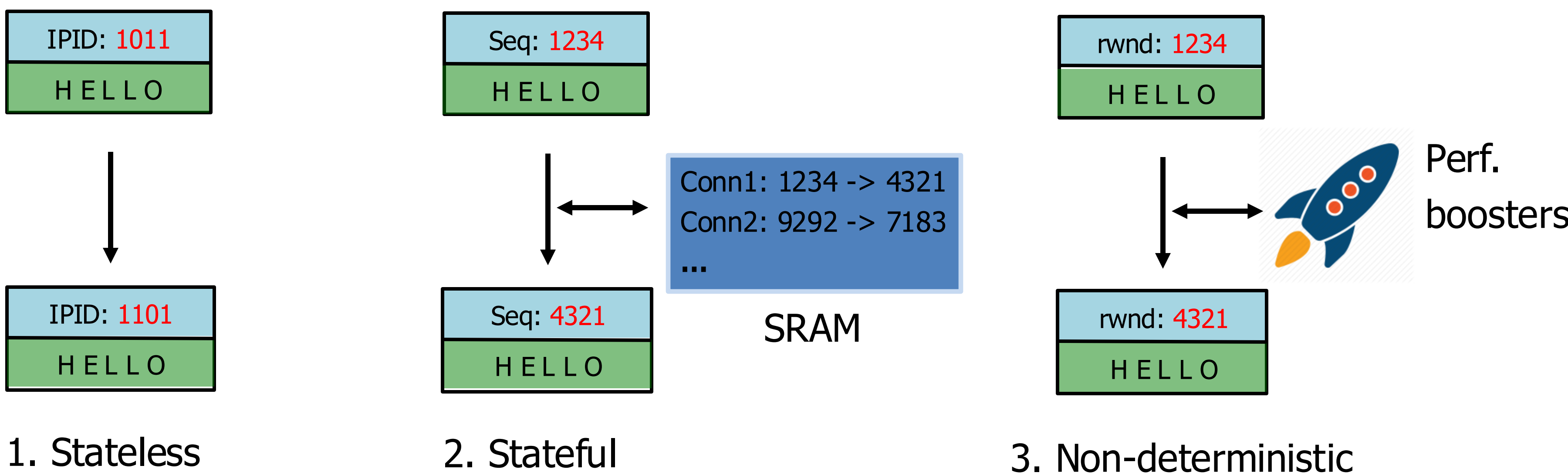
- Existing defenses incur heavy performance penalty
  - Needs to process every single packet
    - → Modern networks have high speeds (Tbps)
  - Imperfect detection leads to collateral damage
    - → Normal traffic may be adversely impacted

Goal: Performance-preserving defenses



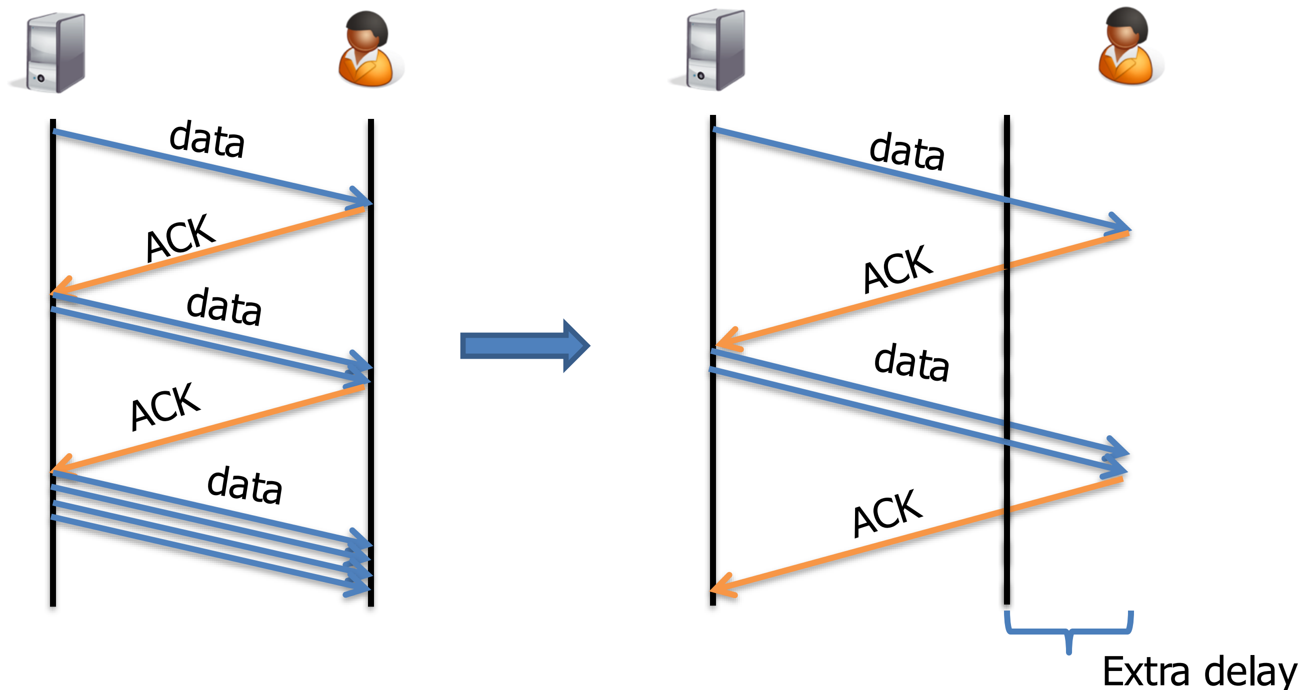
- Per-packet scrubbing at hardware speeds
- Manipulate TCP congestion control to mask latency
- Batch operations in software for TCP reliability

Per-packet defense



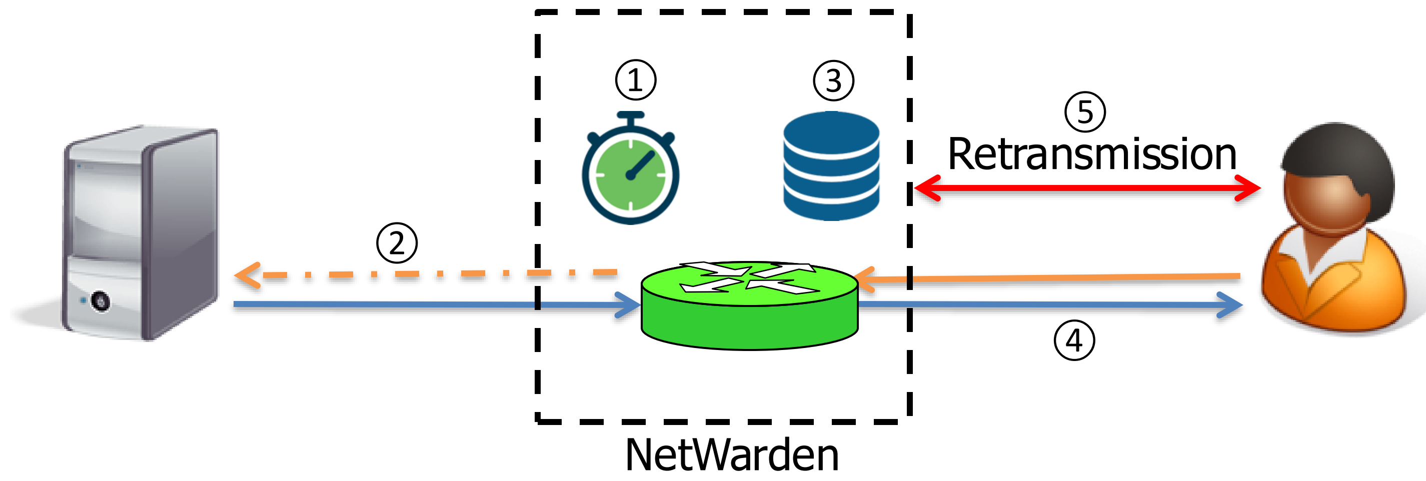
- The P4 data plane performs per-packet defenses
  - Stateless: IPID, TTL, …
  - Stateful: TCP sequence number, ACK, ..
  - Non-deterministic: receive window, timing channels..

Manipulating TCP congestion control



- Consider the timing channel defense:
  - Adding delay to packets disrupts TCP throughput
- ACK boosters:
  - Optimistically ACK packets from the switch
  - Create the illusion of the same RTT for the sender

ACK boosters



- Creates the illusion of the same RTT despite extra delay
  - (1) Measures (actual) RTT in data plane
  - (2) Generates ACK optimistically to trigger more data
  - (3) Caches extra packets for reliability
  - (4)-(5) Serves the client from the cache
- Similar as a "performance-enhancing TCP proxy"
  - But accelerated in P4 hardware

Hardware/software codesign

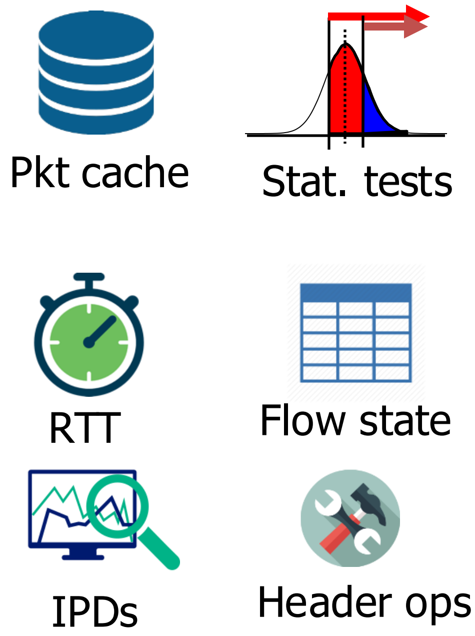
Switch control plane

- Computation: Batch operations
- Memory: Growing state

- Minimize crosstalk

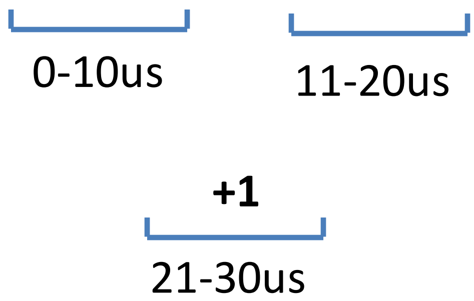
- Computation: Per-packet operations
- Memory: Constant state

Switch data plane

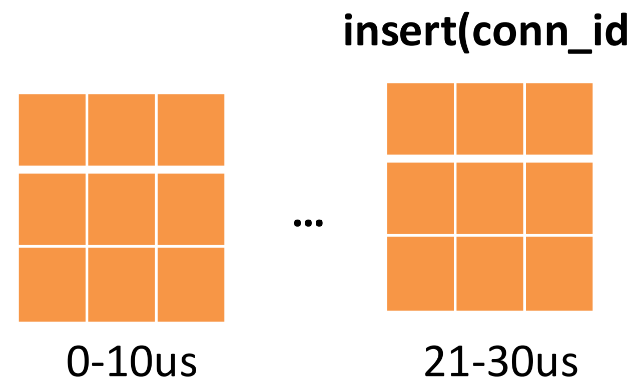


- Challenge: P4 has a restricted programming model
  - Cannot perform statistical tests, packet caching ..
- Solution: Division of labor between SW+HW
  - Principles are generally applicable to other defenses

Other techniques: Computing IPDs



IPD Intervalization:
No per-pkt state



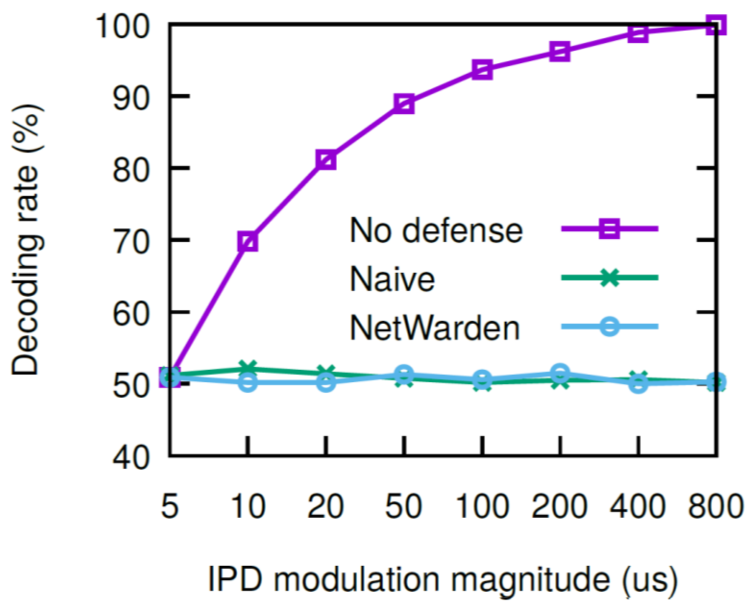
IPD Sketching:
No per-conn state

- Challenges: for computing inter-packet gaps
  - Per-packet operations needed
  - But also accumulates state very fast (2B per packet)
- Solution: two approximation techniques

Evaluation setup

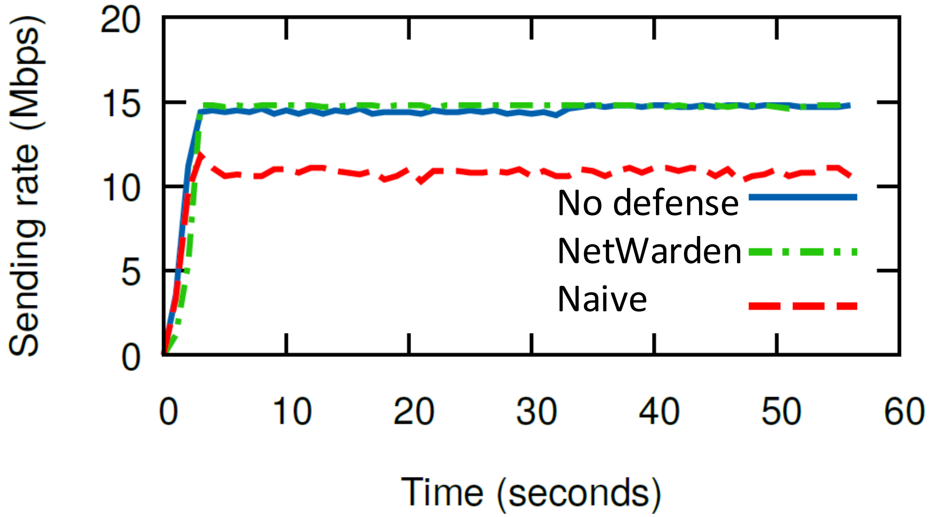
- NetWarden prototype:
  - Runs in Tofino Wedge 100BF-32X switch.
  - 2500 LoC of P4 + 3000 LoC of C+Python
- Threat model:
  - A compromised server + a trusted P4 switch running NetWarden
  - Leak a 2048-bit RSA key via covert channel.
- Real world applications:
  - Apache servers, FTP servers, Nodejs servers
- Baseline:
  - No defense: No covert channel defenses are deployed.
  - Naïve defense: Covert channel defenses without performance preservation

Defense effectiveness



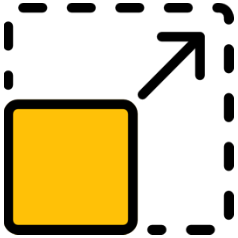
- Naïve defense: renders decoding to a random guess.
- NetWarden: very close to a random guess.
- → Mitigates covert channels effectively.

Performance preservation

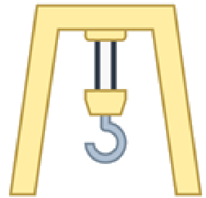


- Naïve defense incurs 25% performance penalty.
- NetWarden only has 1% performance deviation
- → Mitigates channels with minimal performance loss.

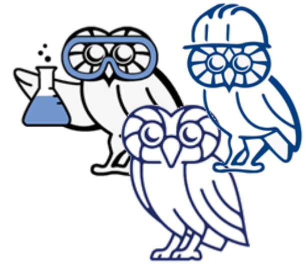
Other results



System scalability



System overhead



Different TCP variants

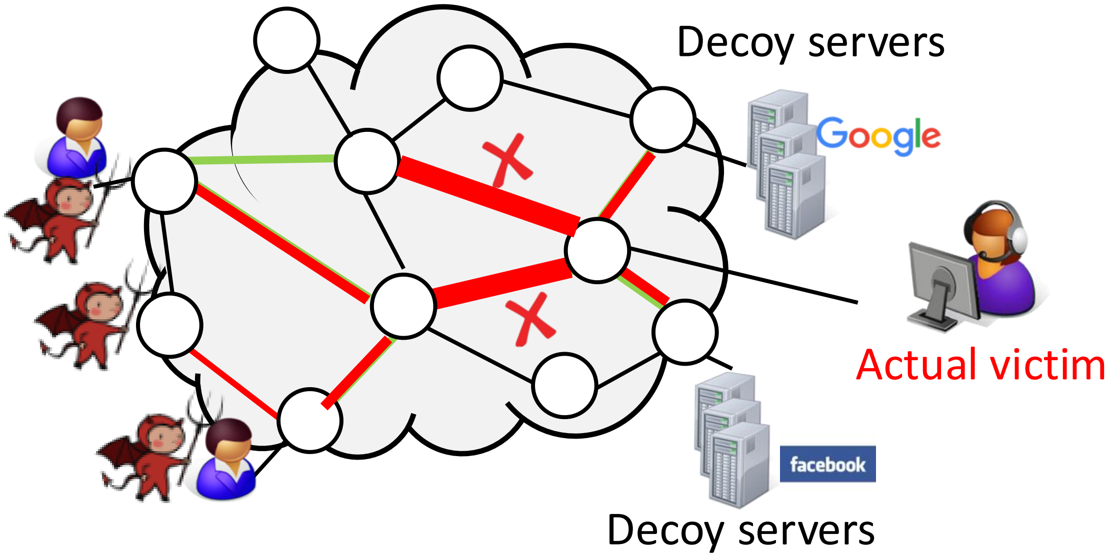
Complex Applications

Self-defense techniques

Outline

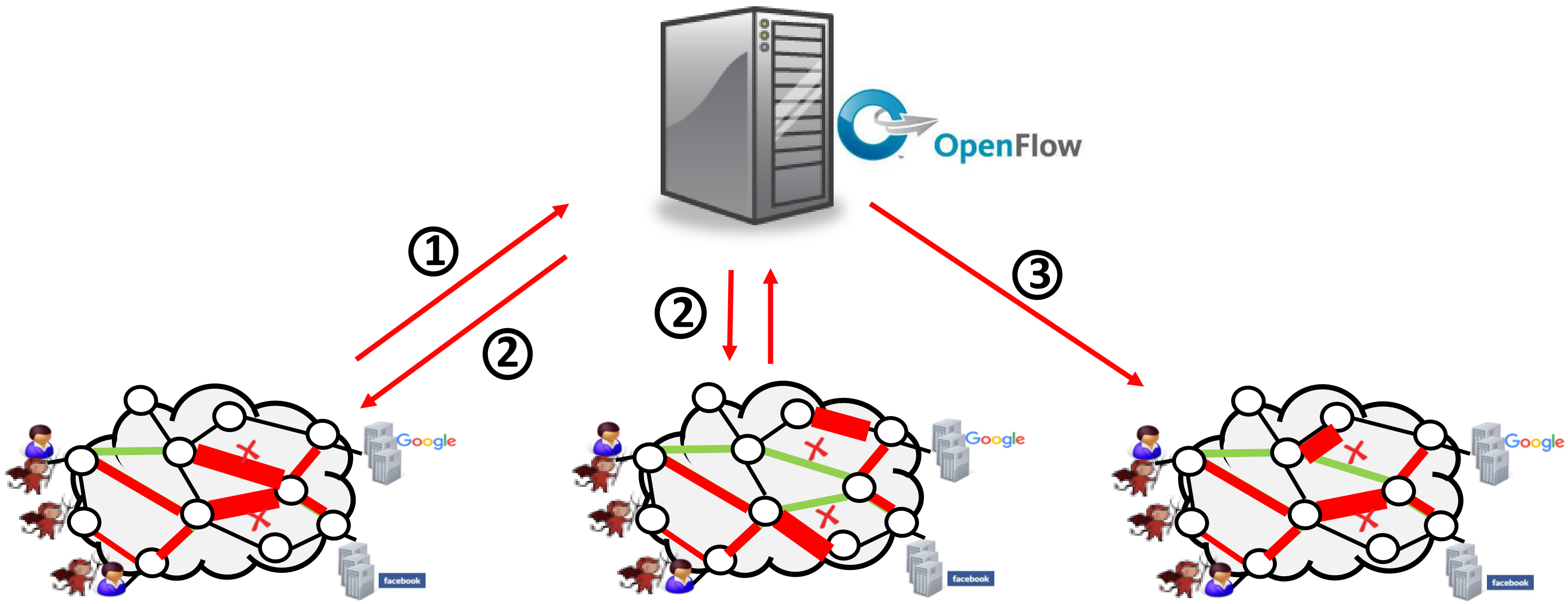
- Programmable In-network Security
  - The switch as a defense platform
  - The network as a defense fleet
  - Securing the defenses
- Future work
- Summary

Motivation: Network-wide attacks



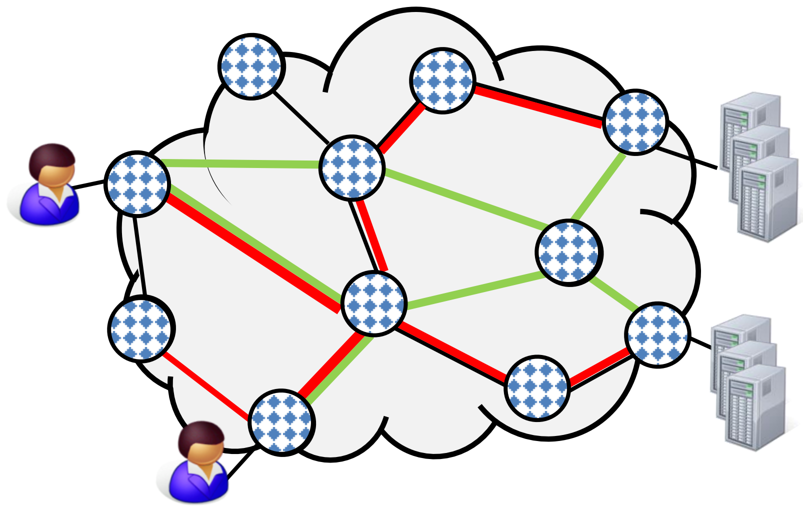
- Individual switches have local views, but global views and actions are needed
- Example: Link-flooding attacks [USENIX Security'21]
  - Congest critical links to take down remote victim
  - Attackers can easily change traffic type, link target, ..

State of the art: SDN defenses



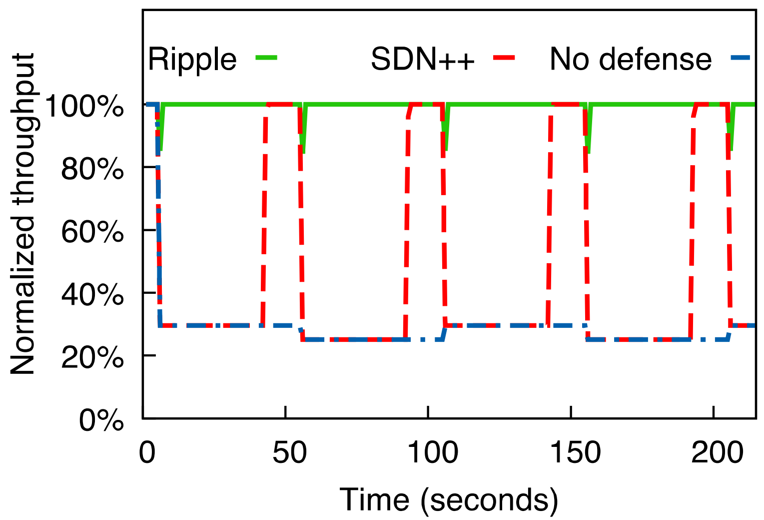
- A central controller samples traffic, computes decisions, and installs rerouting/filtering rules
- Limitation: Cannot handle dynamic attacks
  - Changing botnet composition
  - Changing traffic patterns
  - Changing target links

Ripple: A fully decentralized defense



- Programmable switches coordinate in data plane
- A panoramic view as defined in a policy language
- Switches extract local signals, propagate them across the network, and reconstruct the global view

Ripple mitigates fast-changing attacks

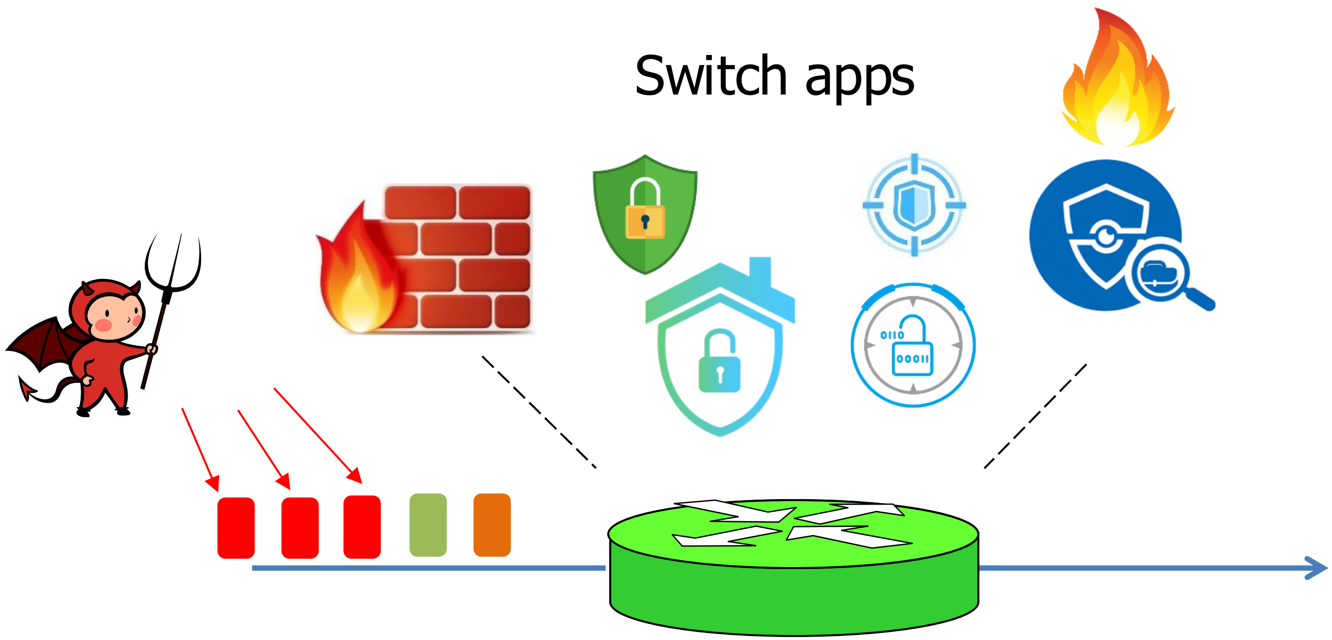


- Setup: Dynamic attacks based on "Crossfire" [SP '13]
- Baseline solutions: SDN variants
- Ripple can mitigate attacks in real time, whereas SDN defenses constantly lag behind.

Outline

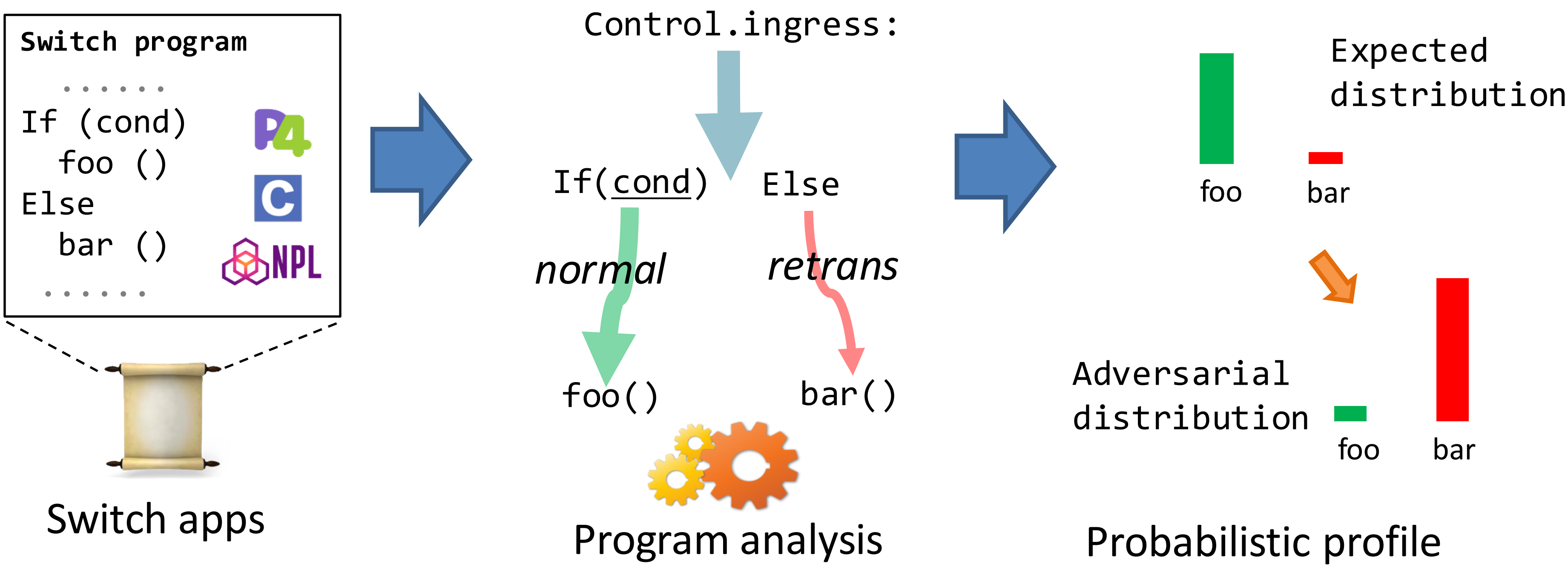
- Programmable In-network Security
  - The switch as a defense platform
  - The network as a defense fleet
  - Securing the switch apps
- Other work
- Summary

Motivation: Adversarial testing



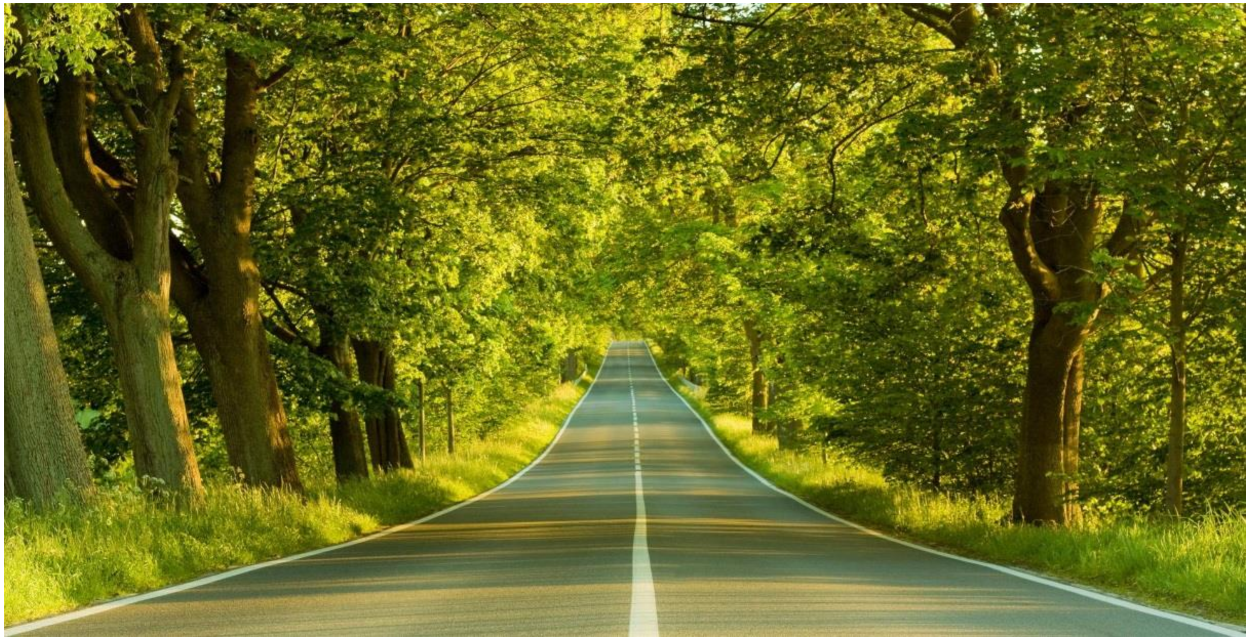
- Automatically identify “adversarial traces” for a P4 system?
  - Example: We’ve manually done this for NetWarden
  - Adversarial traces exist for other P4 systems, too

Worst-case behavior analysis?



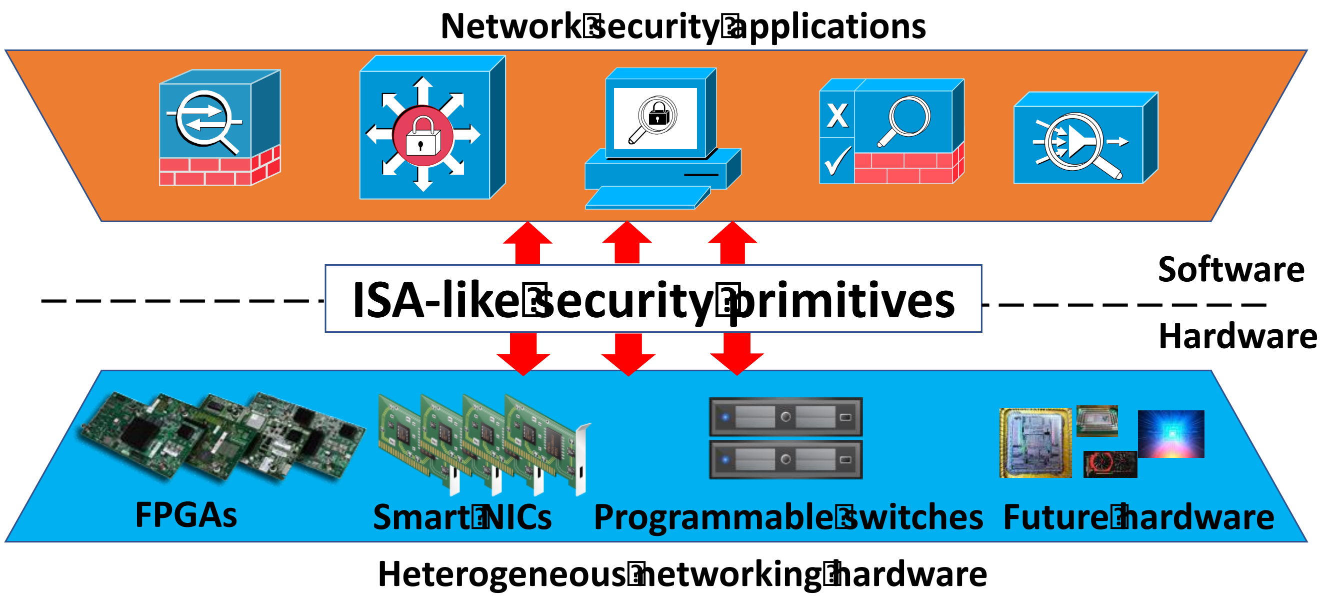
- Intuition: Network programs have complex behaviors, and some are easier to process than others.

Where to go from here?



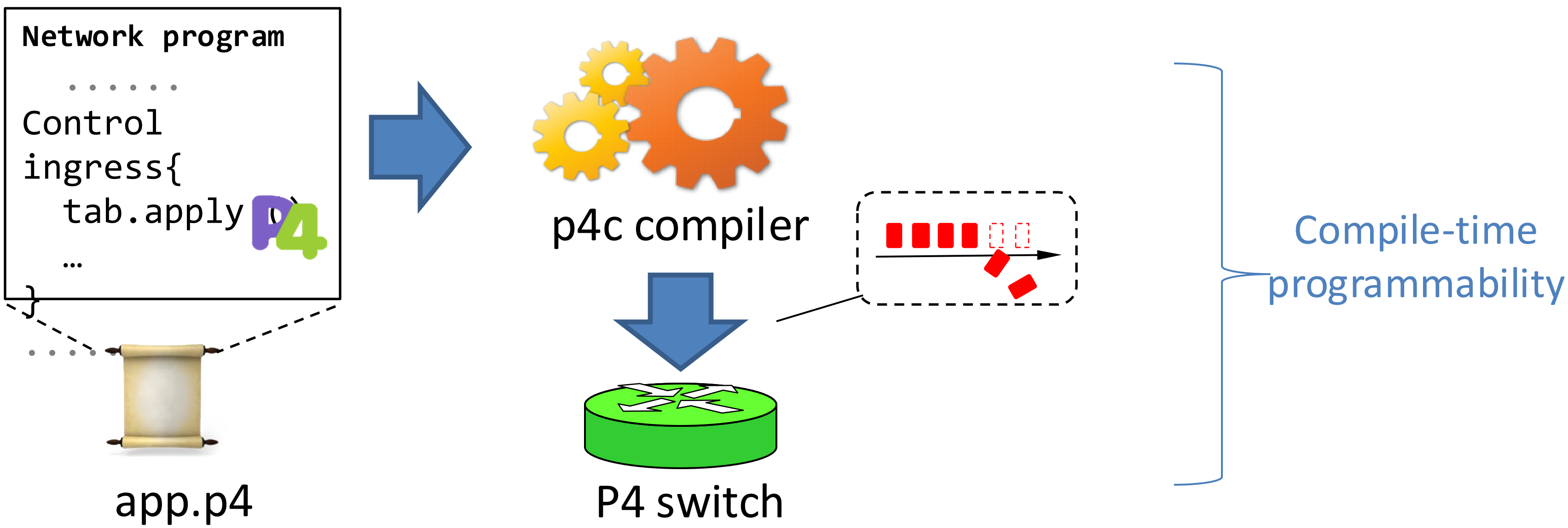
- Programmable In-network Security (Poise)
  - The switch as a defense platform
  - Securing the switch apps
  - The network as a defense fleet

The Poise "network security stack"



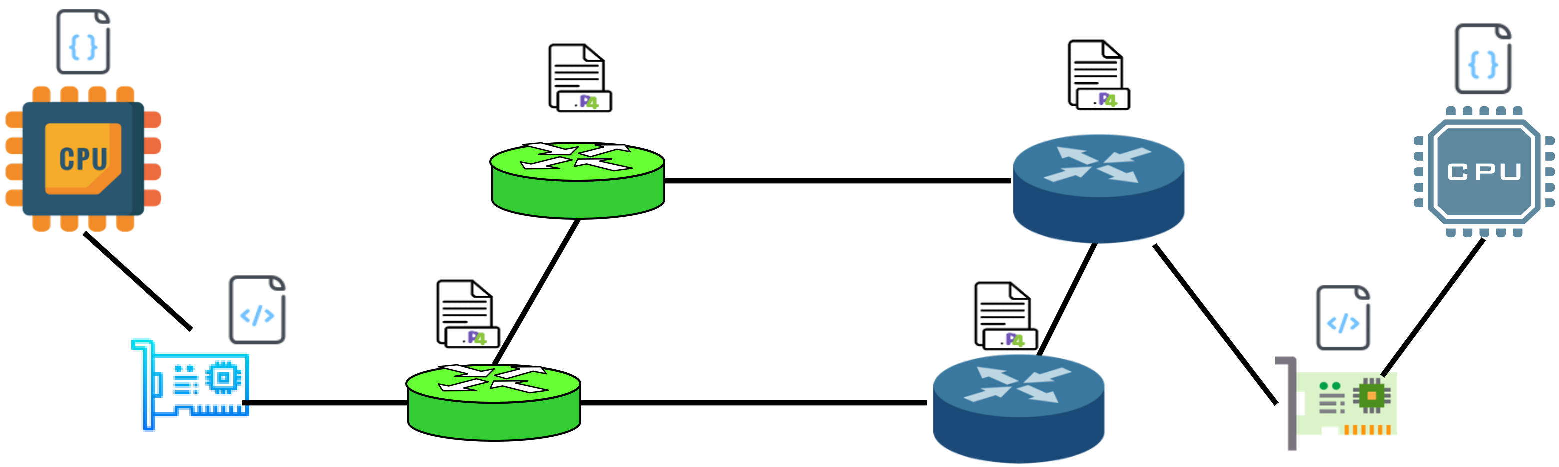
- The "narrow waist" of the network security stack
  - ISA-like primitives for security
  - Small building blocks, verifiable
  - Recomposable for different tasks
  - Dynamically swapped in and out for defense

Need to add runtime flexibility!



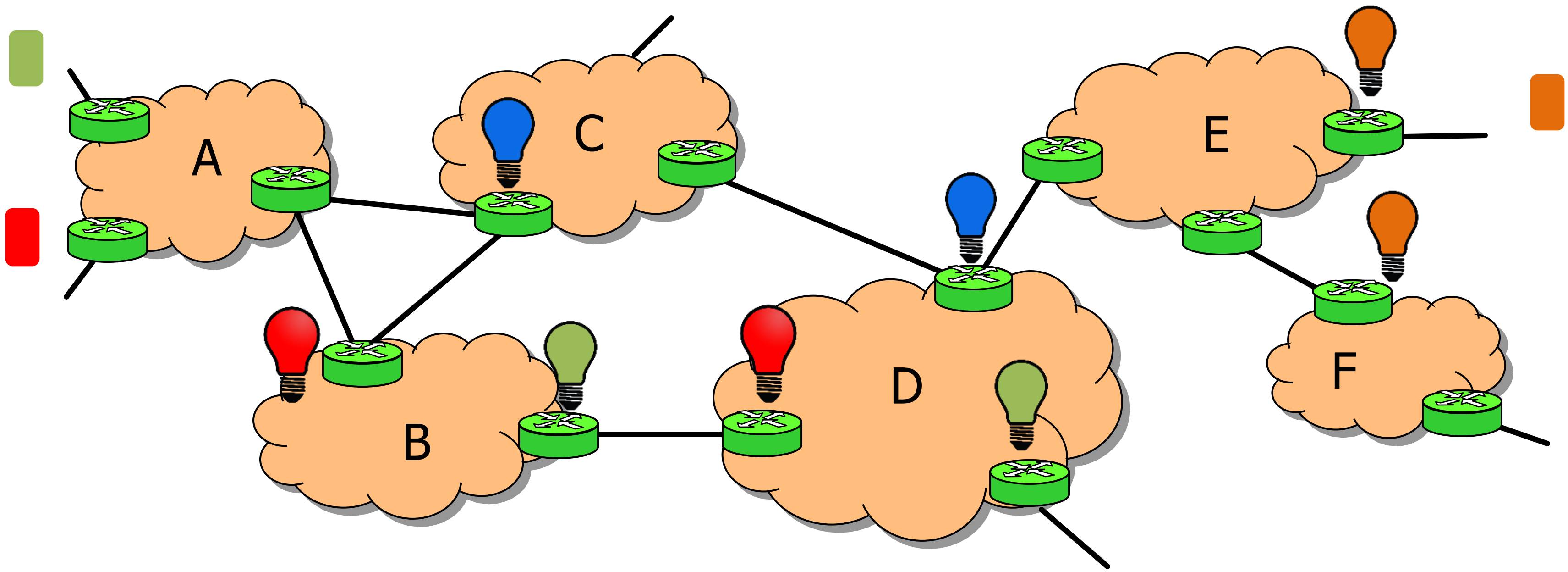
- Operators compile and reflash a new program
  - Causes data plane disruption, so need to provision for that
  - Drain and undrain traffic for any data plane changes
  - Expensive to plan and perform, constrains change velocity

Runtime programmable networks



- Runtime network (re)programming end-to-end
- No downtime, zero packet loss, consistency guarantees
- Users can inject real-time “network extensions”

Why runtime programmable networks?



- Use case: Real-time security defense
  - Network devices swaps defenses in and out
  - Defenses dynamically migrate across the network
  - They shapeshift in real time for changing attacks

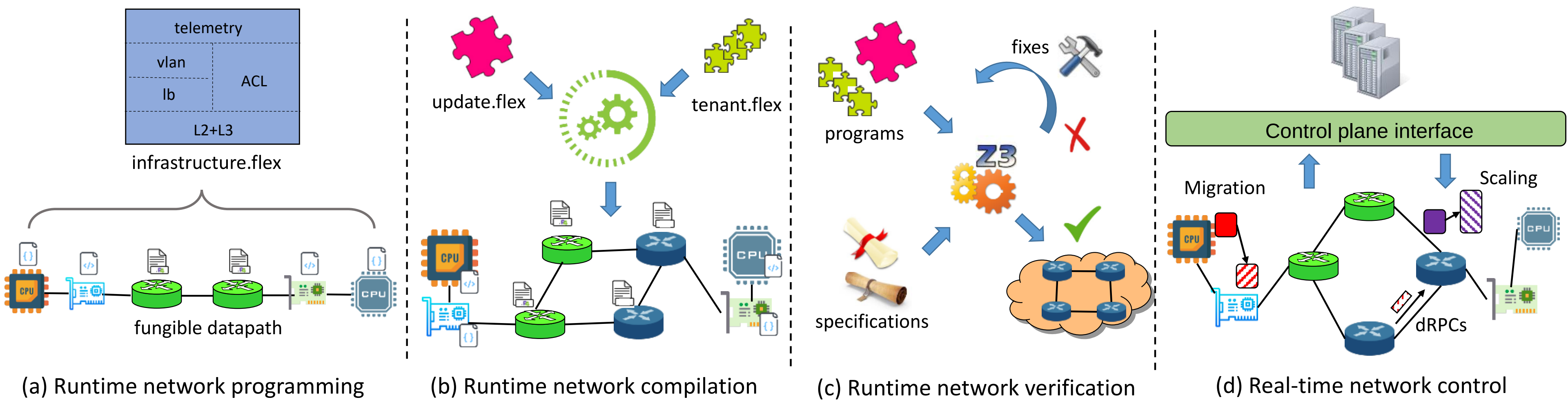
Why runtime programmable networks?

- Just-in-time network optimization
  - Common mode: Basic routing, low latency
  - JIT optimization based on applications and workloads
- Tenant-specific extensions
  - Tenant directly customize network logic
- Incremental infrastructure upgrades
  - Coordinated changes at NICs, switches, and hosts
  - Ex: new congestion control algorithms

Why now?

- Individual targets are becoming runtime programmable
  - Ex: Runtime programmable switches (NSDI’22)
    - Dynamically add/remove/modify tables, control flow, parsers
    - Prototyped on a 12.8Tbps ASIC
  - Ex: Runtime programmable SmartNICs (ongoing)
  - Ex: Host networking based on eBPF

The overall vision



- Brand new challenges and opportunities ahead!

Summary

- Motivation: Next-generation network security
- Approach: Programmable In-network security
  - Architect security back into the network foundation
- We have already made some progress:
  - A switch as a defense platform
  - A network as a defense fleet
  - Securing the defenses
- More opportunities await!

Thank you!