Software Defined Networking (SDN)

PRESENTED BY: LUKE LAMBERT

Outline

- Research Hypothesis
- Overview of SDNs [3, 6]
- OpenFlow [1]
- Programming Protocol-Independent Packet Processors (P4) [2]
- OpenFlow Security Vulnerabilities & other Shortcomings [4,5,6]
- Comparing Virtualization in Legacy Networks and SDN [3]
- Revisiting my Research Hypothesis

Research Hypothesis

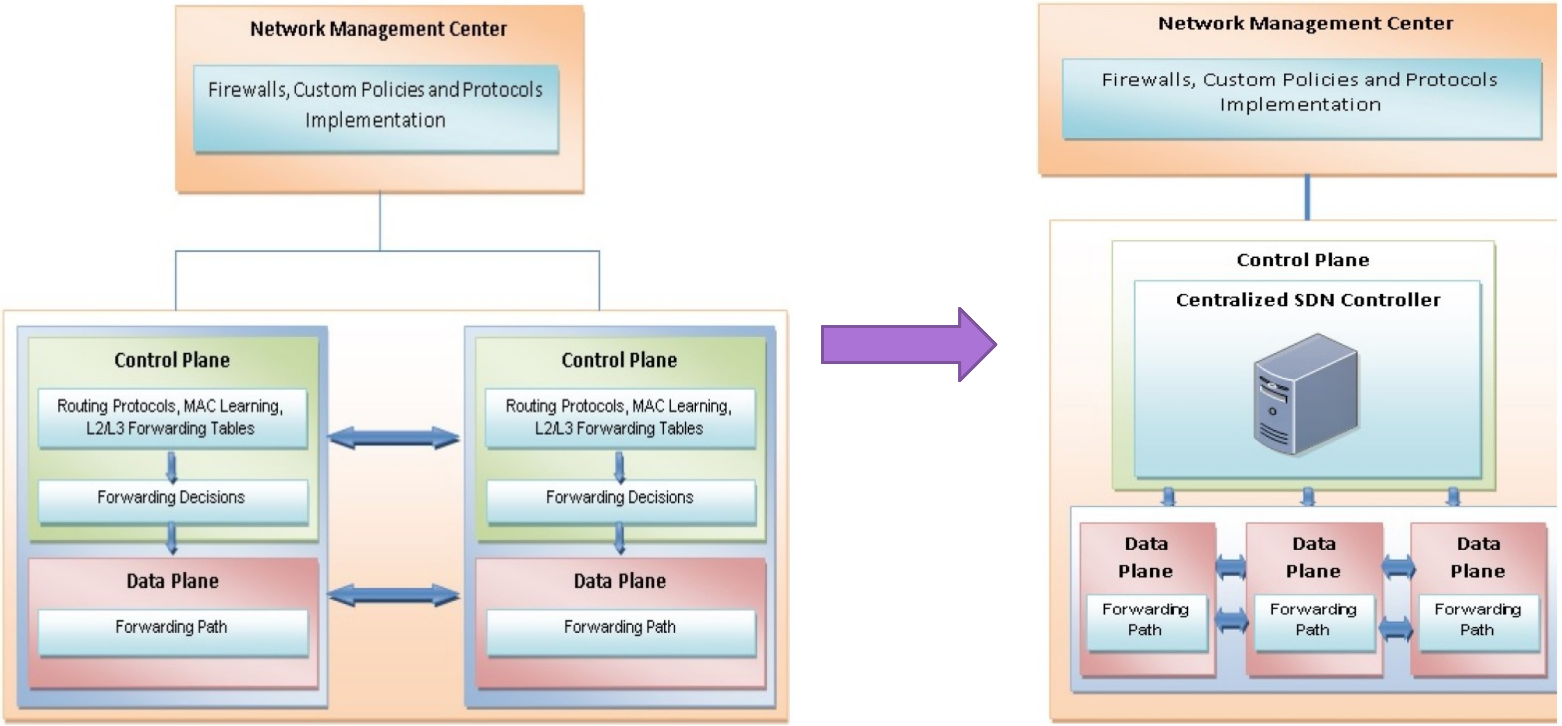
- “SDN gives researchers a practical method of experimentation with new network protocols in realistic settings”

Software Defined Networking (SDN)

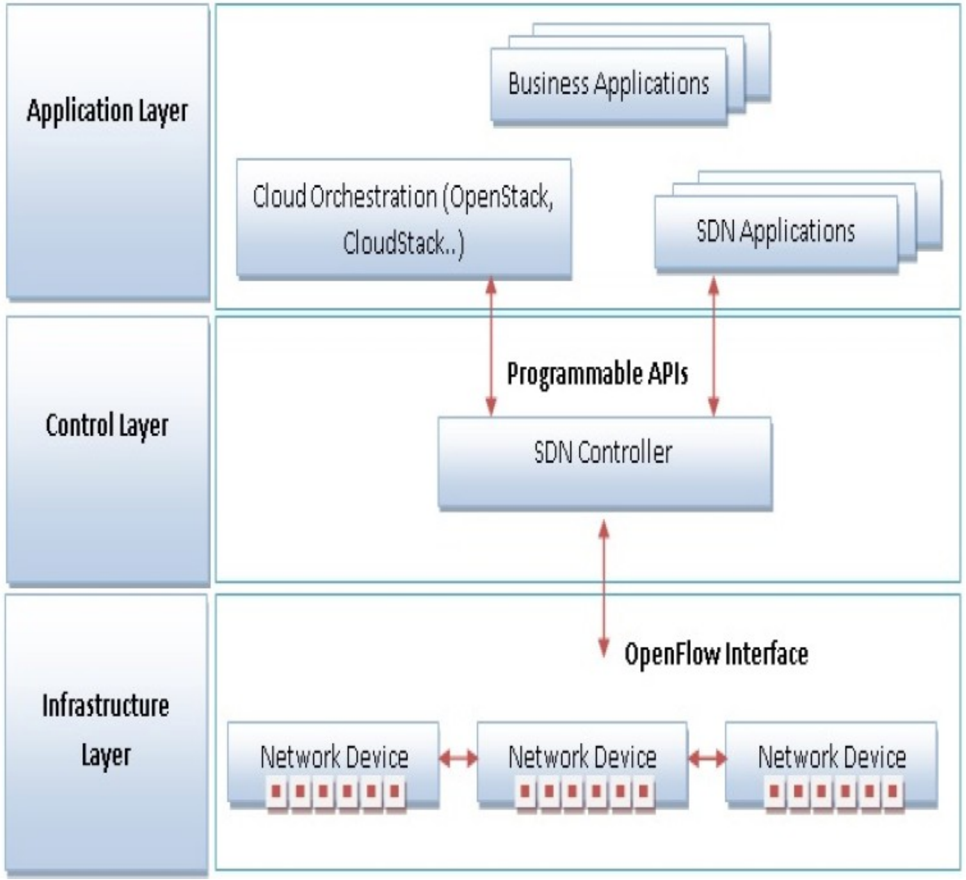
- What is the problem?
  - Legacy network platforms do not have built-in flexibility, automation, programmability, and support to test and implement new networking ideas without interrupting ongoing services
- Why should we care?
  - New ideas go untried & untested
  - Network infrastructure has stagnated
  - High barrier to entry for developers

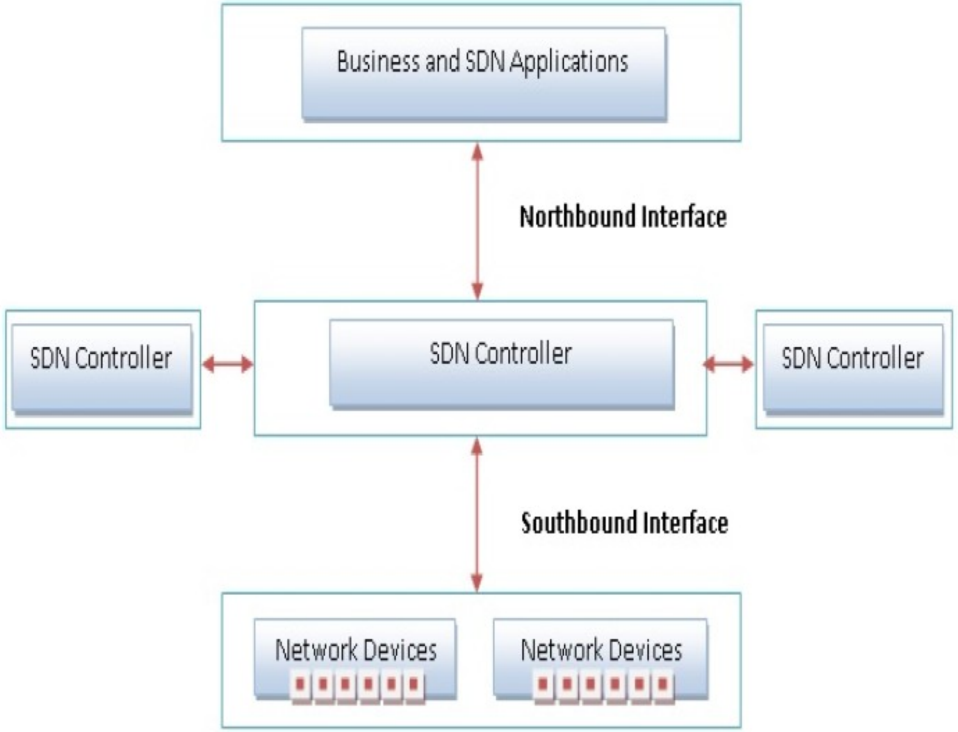
What is an SDN? [3, 6]

- SDN is an emerging network architecture where network control is decoupled from forwarding devices and is directly programmable



SDN Architecture [3, 6]





OpenFlow [1]

- Protocol used for managing the southbound interface of the generalized SDN architecture
- First standard interface defined to facilitate interaction between the control and data planes of the SDN architecture
  - "OpenFlow defines initial concept of SDN and SDN governs future development of OpenFlow" [3]
- Provides software-based access to the flow tables that instruct switches and routers how to direct network traffic
- Provides management tools to control topology changes and packet filtering

OpenFlow – What is a flow?

- A TCP connection
- All packets from a particular MAC address or IP address
- All packets with the same VLAN tag
- All packets arriving from the same switch port

OpenFlow Switches

- Consists of:
  - A flow table
  - A secure channel connecting the switch to the controller
  - The OpenFlow protocol
- Switches can be categorized into:
  - Dedicated OpenFlow Switches
  - OpenFlow-enabled Switches

Dedicated OpenFlow Switches

- Dumb datapath elements that forwards packets between ports according to the controller
- Three basic actions:
  - Forward flow’s packets to a given port
  - Encapsulate and forward flow’s packet to a controller
  - Drop flow’s packet

OpenFlow-enabled Switches

- Commercial switches, routers, and access points
- Support 3 basic actions
- Enhanced with:
  - Flow table
  - Secure channel
  - OpenFlow Protocol
- OpenFlow-enabled switches must isolate experimental traffic from production traffic
  - Two main ways to do this:
    - Add a fourth action: forward a flow's packets through the switch's normal processing pipeline
    - Separate sets of VLANs into experimental and production traffic

Flow Table

- An entry in the flow table has three fields:
  - A packet header
  - The action
  - Statistics
- First generation “Type 0” switch flow header:

In Port	VLAN ID	Ethernet			IP			TCP	
		SA	DA	Type	SA	DA	Proto	Src	Dst

OpenFlow Controller

- Central control point that oversees a variety of OpenFlow-enabled network components
- Adds/removes flow-entries from the flow table
- Two implementations:
  - Static Controller
    - Can be considered a generalization of VLANs
  - Dynamic Controller
    - Add/remove flows as experiment progresses
    - Can support multiple researchers

Simple OpenFlow Example

- I invent a new routing protocol and want to try it in a network of OpenFlow switches without altering end-point software
- My protocol will run on a controller
- Each time a new application flow starts, my protocol picks a route through the OpenFlow switches and adds a new flow entry in each switch along the path
- I want to use my own desktop PC to run my protocol
  - Define one flow to be all traffic entering OpenFlow switch through the switch port my PC is connected to
  - Add flow entry with action: “Encapsulate and forward all packets to a controller”
- Each time a new application flow starts (packets reach the controller), my protocol:
  - Chooses a route through a series of OpenFlow switches
  - Adds a flow-entry in each switch along the path
  - Subsequent packets are processed quickly by the Flow Table

OpenFlow – Other Applications

- Network Access Control
  - Check flow against a set of rules
- VLANs
  - Static flows
- Mobile wireless VOIP clients
- Supporting a non-IP network
  - Flows identified using their Ethernet header
  - A new EtherType value
  - New IP version number
- Processing packets rather than flows
  - Force all packets to flow through a controller
  - Route packets to a programmable switch

OpenFlow – Improving the Match Rule Flexibility

- The first version of OpenFlow matched only 12 fields and grew more complicated from there

Version	Date	Header Fields
OF 1.0	Dec 2009	12 fields (Ethernet, TCP/IPv4)
OF 1.1	Feb 2011	15 fields (MPLS, inter-table metadata)
OF 1.2	Dec 2011	36 fields (ARP, ICMP, IPv6, etc.)
OF 1.3	Jun 2012	40 fields
OF 1.4	Oct 2013	41 fields

Motivation for Programming Protocol-Independent Packet Processors (P4) [2]

- [2] argues that future switches should support flexible, programmable mechanisms for parsing packets and matching header fields (OpenFlow 2.0)
- Recent chips can be flexible but the code is not portable
- There is need for a high-level language for Programming Protocol Independent Packet Processors (P4)
  - Raises network abstraction
  - Can serve as a general interface between a controller and switches
  - Three goals:
    - Reconfigurability
    - Protocol Independence
    - Target Independence

P4 - Abstract Switch Forwarding Model

- Generalizes how packets are processed in different forwarding devices by different technologies
- Controlled by two types of operations or phases:
  - Configure
    - Program the parser
    - Set order of match+action stages
    - Specify header fields
  - Populate
    - Adds/removes entries to the match+action table (flow table)

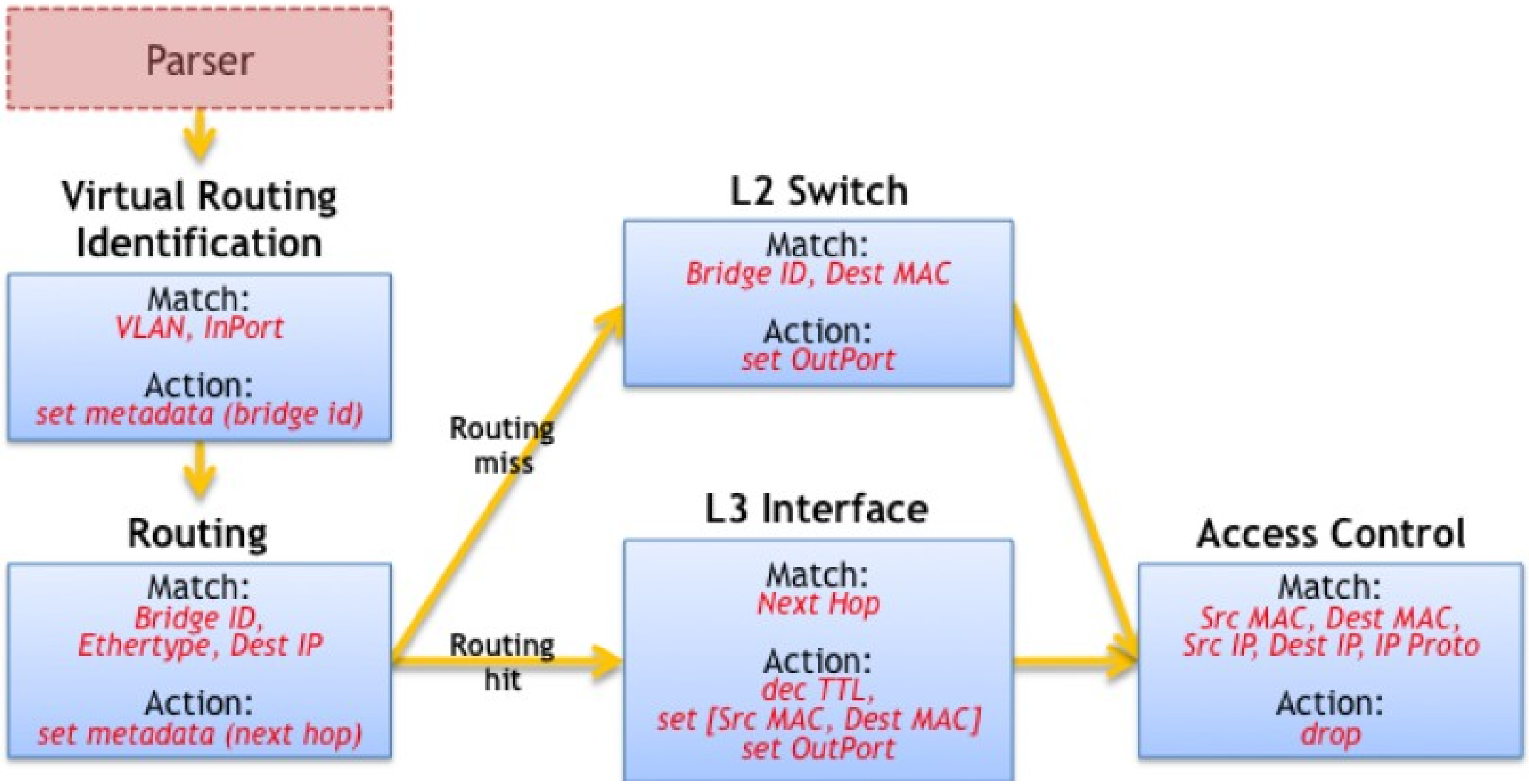
P4 - Abstract Switch Forwarding Model

- Arriving packets handled by parser
  - Parser recognizes and extracts fields from header
  - Extracted header fields passed to match+action tables
- Match+action tables are divided between ingress and egress packets
  - Both may modify packet header
  - Ingress match+action determines the egress port and queue
  - Based on ingress processing, a packet may be forwarded, replicated, or dropped
  - Egress match+action performs per-instance packet header modifications
- Action tables may be associated with a flow to track frame-to-frame state

P4 – Programming Language

- Main goal of paper
- A packet processing language must allow the programmer to express any serial dependencies between header fields
- Dependencies can be identified by analyzing Table Dependency Graphs (TDGs)
  - TDGs describe the field inputs, actions, and control flow between tables
  - TDG nodes map directly to match+action tables
  - Dependency analysis shows where each table may reside
- Programmers express packet processing programs using P4
- Compiler translates P4 representation to TDGs and maps TDGs to specific switch targets

P4 – Example TDG



P4 Concepts

- Headers
  - Describes sequence and structure of a series of fields
  - Specifies field widths and constraints
- Parsers
  - Specifies how to identify headers and valid header sequences within packets
- Tables
  - P4 defines fields on which the table may match and what action it will take
- Actions
  - P4 supports construction of complex actions from simpler protocol-independent primitives
- Control Programs
  - Determines order of match+action tables that are applied to a packet

P4 Example

- Consider an example L2 network deployment with top-of-rack (ToR) switches at the edge connected by a two-tier core
- Assume the number of end-hosts are growing and the core L2 tables are overflowing
- We want to implement a new protocol, mTag

P4 Example – Header Formats

- Headers are specified by declaring an ordered list of field names together with their widths
- Optional field annotations may be added to specify constraints on value ranges or maximum lengths for variable-sized fields

```
header ethernet {
    fields {
        dst_addr : 48; // width in bits
        src_addr : 48;
        ethertype : 16;
    }
}
```

```
header vlan {
    fields {
        pcp : 3;
        cfi : 1;
        vid : 12;
        ethertype : 16;
    }
}
```

P4 Example – Header Formats

- The mTag header can be added without altering existing declarations
- Field names indicate two layers of aggregation
- Switches programmed with rules to examine one of these bytes

```
header mTag {
    fields {
        up1 : 8;
        up2 : 8;
        down1 : 8;
        down2 : 8;
        ethertype : 16;
    }
}
```

P4 Example – The Packet Parser

- P4 assumes the underlying switches are capable of creating a state machine that traverses packet headers from start to finish
- State machine = set of transitions from one header to the next
- Each transition may be triggered by values in the current header
- Parsing starts in the start state and proceeds until:
  - A stop state is reached
  - An unhandled case is encountered
- Upon reaching a state for a new header, the state machine extracts the header and identifies the next transition
- Extracted headers are forwarded to match+action processing

P4 Example – The Packet Parser

```
parser start {
    ethernet;
}

parser ethernet {
    switch(ethertype) {
        case 0x8100: vlan;
        case 0x9100: vlan;
        case  0x800: ipv4;
        // Other cases
    }
}
```

```
parser vlan {
    switch(ethertype) {
        case 0xaaaa: mTag;
        case  0x800: ipv4;
        // Other cases
    }
}

parser mTag {
    switch(ethertype) {
        case 0x800: ipv4;
        // Other cases
    }
}
```

P4 Example – Table Specification

- Next, the programmer must describe how defined headers should be matched and what actions should be performed when a match occurs
- The table specification allows a compiler to decide how much memory it needs and the memory type
- Several attributes are used:
  - reads which fields to match and the match type
  - actions lists the possible actions which can be applied to a packet by the table
  - max_size describes how many entries the table should support

P4 Example – Table Specification

```
table mTag_table {
    reads {
        ethernet.dst_addr : exact;
        vlan.vid : exact;
    }
    actions {
        // At runtime, entries are programmed with params
        // for the mTag action.  See below.
        add_mTag;
    }
    max_size : 20000;
}
```

P4 Example – Table Specification

```
table source_check {
    // Verify mtag only on ports to the core
    reads {
        mtag : valid; // Was mtag parsed?
        metadata.ingress_port : exact;
    }
    actions { // Each table entry specifies *one* action

        // If inappropriate mTag, send to CPU
        fault_to_cpu;

        // If mtag found, strip and record in metadata
        strip_mtag;

        // Otherwise, allow the packet to continue
        pass;
    }
    max_size : 64; // One rule per port
}
```

```
table local_switching {
    // Reads destination and checks if local
    // If miss occurs, goto mtag table.
}

table egress_check {
    // Verify egress is resolved
    // Do not retag packets received with tag
    // Reads egress and whether packet was mTagged
}
```

P4 Example – Action Specification

- P4 defines a set of primitive actions from which more complicated actions can be built
- Each P4 program declares a set of action functions, composed of action primitives
  - Used to simplify table specification and population
- P4 assumes parallel execution of action primitives within an action function

P4 Example – Action Specification

- P4's primitive actions include:
  - set_field: Set a specified field in a header to a value
  - copy_field: Copy one field to another
  - add_header: Set a specific header instance (and all its fields) as valid
  - remove_header: Delete a header (and all its fields) from a packet
  - increment: increment or decrement a value in a field
  - checksum: calculate a checksum over some set of header fields (ex. An IPv4 checksum)

P4 Example – Action Specification

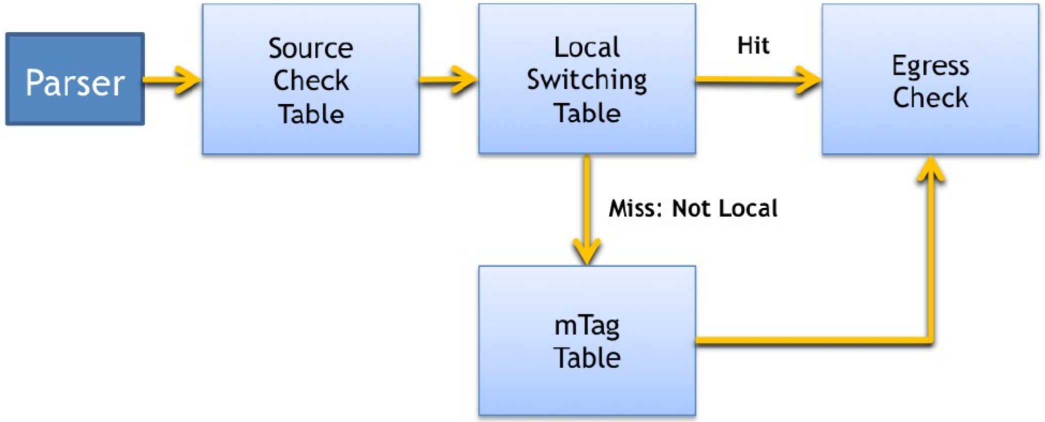
```
action add_mTag(up1, up2, down1, down2, egr_spec) {
    add_header(mTag);
    // Copy VLAN ethertype to mTag
    copy_field(mTag.ethertype, vlan.ethertype);
    // Set VLAN's ethertype to signal mTag
    set_field(vlan.ethertype, 0xaaaa);
    set_field(mTag.up1, up1);
    set_field(mTag.up2, up2);
    set_field(mTag.down1, down1);
    set_field(mTag.down2, down2);

    // Set the destination egress port as well
    set_field(metadata.egress_spec, egr_spec);
}
```

P4 Example – The Control Program

- After tables and actions are defined, the only remaining task is to specify the flow of control from one table to the next
  - Control Flow is specified as a program via a collection of functions, conditionals, and table references

P4 Example – The Control Program




```
control main() {
    // Verify mTag state and port are consistent
    table(source_check);

    // If no error from source_check, continue
    if (!defined(metadata.ingress_error)) {
        // Attempt to switch to end hosts
        table(local_switching);

        if (!defined(metadata.egress_spec)) {
            // Not a known local host; try mtagging
            table(mTag_table);
        }

        // Check for unknown egress state or
        // bad retagging with mTag.
        table(egress_check);
    }
}
```

Compiling a P4 Program

- The control flow is a convenient way to specify the logical forwarding behavior of a switch, but does not explicitly call out dependencies between tables or opportunities for concurrency
- Therefore, P4 employs a compiler that analyzes the control program to identify dependencies and look for opportunities to process header fields in parallel
- The compiler also generates the target configuration for the switch

Compiling a P4 Program

- A two step compilation process is used:
  - First, the P4 control program is converted into a intermediate table dependency graph, which is analyzed to determine dependencies between tables
  - Then, a target-specific back-end maps this graph onto a switch's specific resources

OpenFlow Security Vulnerabilities [4][5]

- The original OpenFlow specification required the control channel between the controller and switches to be protected using Transport Layer Security (TLS)
- Unfortunately, as of v1.3.0, TLS is made optional and many vendors to not follow the recommendation
  - “The switch and controller **may** communicate through a TLS connection”
- The lack of TLS leaves an avenue to infiltrate OpenFlow networks and remain undetected

OpenFlow Security Vulnerabilities [4][5]

- Man-in-the-Middle Attacks
  - Easier to perform in an OpenFlow network
    - An attacker in traditional network must wait for an operator to log into each switch management interface with an insecure protocol
    - However, constant connectivity and lack of authentication in plaintext OpenFlow controller enables:
      - Attacker to seize full control any down-stream switches
      - Fine-grained eavesdropping attacks

OpenFlow Security Vulnerabilities [4, 5]

- Listener Mode
  - Many switches support “Listener Mode”
    - Unauthenticated connections to a configured TCP port accepted from any network source
    - Allows external connections to write rules to switches and read information for debugging
  - Eliminates need for a Man-in-the-middle attack
  - By discovering a switch with a passive listening port, an attacker may
    - Insert rules to hijack downstream switches
    - Capture traffic
    - Configure switch as proxy for future attacks

OpenFlow Security Vulnerabilities [4, 5]

- Switch Authentication
  - Even with TLS, failure to implement switch authentication in the controller allows attackers to perform network reconnaissance
    - Observe how the controller responds to different packets

OpenFlow Security Vulnerabilities [4, 5]

- Flow Table Verification
  - Even with TLS, switches that erroneously alter rules would not be caught
  - Controller cannot keep track of all switch flow-table state changes
  - Mismatch between controller's idea of rule-states and actual rule states
    - Access-control failure
    - Network outage
    - Other unexpected behavior
  - Only way to verify is by dumping and inspecting flow tables for each switch
    - Computationally expensive for both controller and switches

OpenFlow Security Vulnerabilities [4, 5]

- Denial of Service Risks
  - Centralizing the controller creates a new point of failure
  - Mitigated with multiple controllers
  - Without careful rule design, controllers can be exposed to DoS
    - Majority of risks impact networks that use reactive rules
    - Networks with proactive rules still vulnerable from excessive controller flow modifications
      - OpenFlow 1.3 suggests policing packets destined to controller
  - OpenFlow leaves burden of implementing complex security on application developers, who may be unfamiliar with possible attacks

OpenFlow Security Vulnerabilities [4, 5]

- Controller Vulnerabilities
  - OpenFlow applications are capable of deep packet inspection and conversation reconstruction on the controller
  - Application isolation in OpenFlow is an integral part of network security
    - Without it, compromising a single application could lead to adversarial control of the network

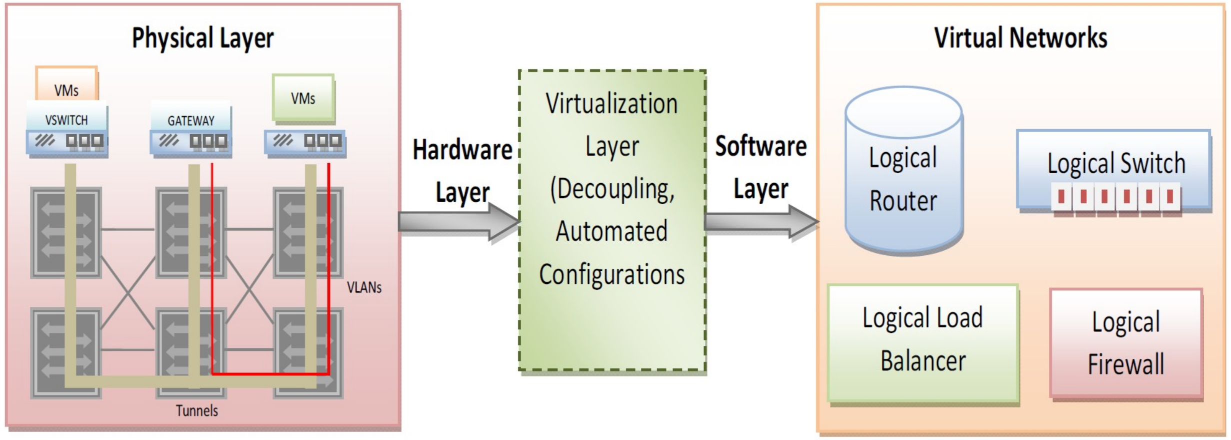
Other Shortcomings with SDN [6]

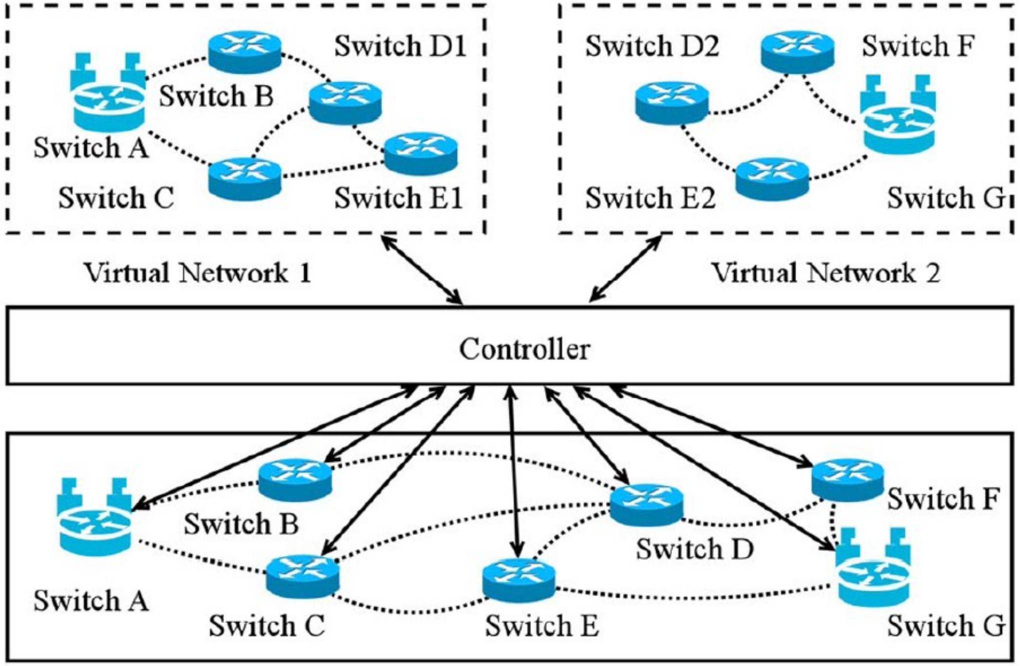
- Reliability
  - Centralized controller = single point of failure
- Scalability
  - NOX – 30,000 requests/s
- Lack of standard APIs
  - No open-source OpenFlow driver
  - No standard north-bound API
  - No standard high level programming language

Comparing Virtualization in Legacy Networks and SDN [6]

- But first, what is the difference between VLAN and Network Virtualization (NV)?
  - While VLAN allows a physical network to be broken into multiple virtual networks, it lacks any fine grained control
  - On the other hand, NV allows the creation of entire networks in software
- So, what are the difference between SDN and NV?
  - SDN = software interacting with hardware, NV = software replicating hardware
  - SDN decouples the control from forwarding devices, decouples and isolates virtual networks from the underlying network hardware
  - SDN requires modifying switches/routers, while NV can reside on the servers of an existing network
  - SDN allows configuration of all virtual networks from a controller, while NVs using virtual tunnels and tags can require tedious configuration
- While there are many differences, both can facilitate virtualization with control

Comparing Virtualization in Legacy Networks and SDN [6]





Revisiting my Research Hypothesis

- "SDN gives researchers a practical method of experimentation with new network protocols in realistic settings"
  - Overall, yes
    - OpenFlow provides the basic infrastructure
    - P4 simplifies OpenFlow programming and improves match rule flexibility
    - OpenFlow has acceptable vulnerabilities **if** TLS implemented properly
  - However,
    - Use-case restricted to campus setting
    - Widespread adoption of strong protocol security is needed to expand OpenFlow use cases

References

[1] N. McKeown, et al. “OpenFlow: enabling innovation in campus networks.” ACM SIGCOMM Computer Communication Review, 2008

[2] Pat Bosshart et. al., P4: programming protocol-independent packet processors. SIGCOMM Computer Communication Reviews, 2014

[3] Xia, Wenfeng, et al. "A survey on software-defined networking." IEEE Communications Surveys & Tutorials 17.1 (2014): 27-51.

[4] Shaghaghi, Arash, et al. "Software-defined network (SDN) data plane security: issues, solutions, and future directions." Handbook of Computer Networks and Cyber Security (2020): 341-387.

[5] Benton, Kevin, L. Jean Camp, and Chris Small. "OpenFlow vulnerability assessment." Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking. 2013.

[6] Jammal, Manar, et al. "Software defined networking: State of the art and research challenges." Computer Networks 72 (2014): 74-98.