Header Space Analysis: Static Checking For Networks

Peyman Kazemian
Stanford University
kazemian@stanford.edu

George Varghese
University of California San Diego
varghese@cs.ucsd.edu

Nick McKeown
Stanford University
nickm@stanford.edu

Abstract

Today's networks typically carry or deploy dozens of protocols and mechanisms simultaneously such as MPLS, NAT, ACLs and route redistribution. Even when individual protocols function correctly, failures can arise from the complex interactions of their aggregate, requiring network administrators to be masters of detail. Our goal is to automatically find an important class of failures, regardless of the protocols running, for both operational and experimental networks.

To this end we developed a general and protocol-agnostic framework, called *Header Space Analysis* (HSA). Our formalism allows us to statically check network specifications and configurations to identify an important class of failures such as *Reachability* Failures, *Forwarding Loops* and *Traffic Isolation and Leakage* problems. In HSA, protocol header fields are not first class entities; instead we look at the entire packet header as a concatenation of bits without any associated meaning. Each packet is a point in the $\{0,1\}^L$ space where L is the maximum length of a packet header, and networking boxes transform packets from one point in the space to another point or set of points (multicast).

We created a library of tools, called Hassel, to implement our framework, and used it to analyze a variety of networks and protocols. Hassel was used to analyze the Stanford University backbone network, and found *all* the forwarding loops in less than 10 minutes, and verified reachability constraints between two subnets in 13 seconds. It also found a large and complex loop in an experimental loose source routing protocol in 4 minutes.

1 Introduction

> "Accidents will occur in the best-regulated families" — *Charles Dickens*

In the beginning, a switch or router was breathtakingly simple. About all the device needed to do was index into a forwarding table using a destination address, and decide where to send the packet next. Over time, forwarding grew more complicated. Middleboxes (e.g., NAT and firewalls) and encapsulation mechanisms (e.g., VLAN and MPLS) appeared to escape from IP's limitations: e.g., NAT bypasses address limits and MPLS allows flexible routing. Further, new protocols for specific domains, such as data centers, WANs and wireless, have greatly increased the complexity of packet forwarding. Today, there are over 6,000 Internet RFCs and it is not unusual for a switch or router to handle ten or more encapsulation formats simultaneously.

This complexity makes it daunting to operate a large network today. Network operators require great sophistication to master the complexity of many interacting protocols and middleboxes. The future is not any more rosy - complexity today makes operators wary of trying new protocols, even if they are available, for fear of breaking their network. Complexity also makes networks fragile, and susceptible to problems where hosts become isolated and unable to communicate. Debugging reachability problems is very time consuming. Even simple questions are hard to answer, such as *"Can Host A talk to Host B?"* or *"Can packets loop in my network?"* or *"Can User A listen to communications between Users B and C?"*. These questions are especially hard to answer in networks carrying multiple encapsulations and containing boxes that filter packets.

Thus, our first goal is to help system administrators statically analyze production networks today. We describe new methods and tools to provide formal answers to these questions, and many other failure conditions, *regardless of the protocols running in the network*.

Our second goal is to make it easier for system administrators to guarantee isolation between sets of hosts, users or traffic. Partitioning networks this way is usually called "slicing"; VLANs are a simple example used today. If configured correctly, we can be confident that traffic in one slice (e.g. a VLAN) cannot leak into another. This is useful for security, and to help answer questions such as *"Can I prevent Host A from talking to Host B?"*. For example, imagine two health-care providers using the same physical network. HIPAA [18] rules require that no information about a patient can be read by other providers. Thus a natural application of slicing is to place each provider in a separate slice and guarantee that no packet from one slice can be controlled by or read by the other slice. We call this *secure slicing*. Secure slicing may also be useful for banks as part of defense-in-depth, and for classified and unclassified users sharing the same

physical network. Our tools can verify that slices have been correctly configured.

Our third goal is to take the notion of isolation further, and enable the static analysis of networks sliced in more general ways. For example, with FlowVisor [5] a slice can be defined by any combination of header fields. A slice consists of a topology of switches and links, the set of headers on each link, and its share of link capacity. Each slice has its own control plane, allowing its owner to decide how packets are routed and processed. While tools such as FlowVisor allow rapid deployment of new protocols, they add to the complexity of the network, pushing the level of detail beyond the comprehension of a human operator. Our tools allow automatic analysis of the network configuration to formally prove that the slicing is operating as intended.

In the face of this need, it is surprising that there are very few existing network management tools to analyze large networks. Further, the tools that exist are protocol dependent and specialized to each task. For example, the pioneering work of Xie, *et al* [3] on static reachability analysis, the analyses of IP connectivity and firewall configuration, e.g. [8, 9, 10, 14], and work on routing failures [11, 12] are all tailored to IP networks. While these papers suggest powerful approaches for reachability in IP networks, they do not easily extend to new protocols and new types of checks.

This paper introduces a general framework, called *Header Space Analysis*, which provides a set of tools and insights to model and check networks for a variety of failure conditions in a protocol-independent way. Key to our approach is a generalization of the geometric approach to packet classification pioneered by Lakshman and Stiliadis [2], in which classification rules over K packet fields are viewed as subspaces in a K dimensional space.

We generalize in three ways. First, we jettison the notion of pre-specified fields in favor of a header space of L bits where each packet is represented by a point in $\{0,1\}^L$ space, where L is the header length. This allows us to work with emerging protocols and arbitrary field formats. Second, we go beyond modeling packet classification in which a header is mapped to a single point in a matching subspace. Instead, we model *all* router and middlebox processing as *box transfer functions* transforming subspaces of the L-dimensional space to other subspaces. For example, in Figure 1, A and B are arbitrary boxes, and T_A and T_B represent their transfer functions. We model how a packet or flow is modified as it travels by composing the transfer functions along the path. Third, we go beyond modeling a single box to modeling a network of boxes using a *network transfer function,* Ψ and a *topology transfer function,* Γ. Ψ combines all individual box functions into one giant function.

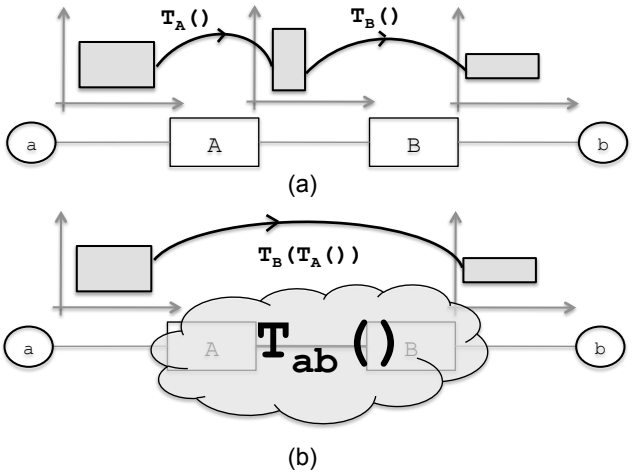

Figure 1: (a) Changes to a flow as it passes through two boxes with transfer function T_A and T_B. (b) Composing transfer functions to model end to end behavior of a network.

Γ models the links that connect ports together. The overall behavior of the network is modeled as a black box by composing Ψ and Γ along all paths.

The contributions of this paper and an outline of the rest of the paper are as follows:

- *Header Space Analysis:* Section 2 describes the geometric model and defines transfer functions. Section 3 shows how transfer functions can be used to model today's networking boxes. Section 4 describes an algebra for working on header space.
- *Use Cases:* Section 5 describes how header space analysis can be used to detect network failures such as reachability failures, routing loops and slice isolation in a protocol independent way.
- *Implementation:* Section 7 describes a library of tools (called *Hassel, or Header Space Library*), based on header space analysis, that can statically analyze networks. We describe five key optimizations that boost Hassel's performance by 5 orders of magnitude relative to a naive implementation.
- *Experiments:* Section 8 reports results of using Hassel to analyze three examples: (1) Stanford University's backbone network, (2) Slice isolation check, and (3) An experimental source routing protocol. We report loops found, and show that even our Python implementation scales to large enterprise networks with our optimizations.

We describe limitations of our approach and related work in Sections 9 and 10. We conclude in Section 11.

2 The Geometric Model

Our *header space* framework is built on a geometric model. We model packets as points *in* a geometric space and network boxes as transfer functions *on* the same ge-

ometric space. Our first task is to define the main geometric spaces.

Header Space, $\mathcal{H}$: We ignore the protocol-specific meanings associated with header bits and view a packet header as a flat sequence of ones and zeros. Formally, a header is a point and a flow is a region in the $\{0,1\}^L$ space, where L is an upper bound on the header length. We call this space *Header Space, $\mathcal{H}$*.

A wildcard expression is the basic building block used to define objects in $\mathcal{H}$. Each wildcard expression is a sequence of L bits where each bit can be either 0, 1 or x. Each wildcard expression corresponds to a hypercube in $\mathcal{H}$. Every region, or flow, in $\mathcal{H}$ is defined as a *union* of wildcard expressions.

$\mathcal{H}$ abstracts away the data portion of a packet because we assume it does not affect packet processing. If it does, as in an intrusion-detection box, then L must be the length of the entire packet. If the fields are fixed, we can define macros for each field in $\mathcal{H}$ to reduce dimensionality. However, the general notion of header space is critical when dealing with different protocols that interpret the same header bits in different ways. Note also that we can model variable length fields such as IP options using a custom parsing function as shown in Section 8.3.

Network Space, $\mathcal{N}$: We model the network as a set of boxes called *switches* with external interfaces called *ports* each of which is modeled as having a unique identifier. We use "switches" to denote routers, bridges, and any possible middlebox.

If we take the cross-product of the switch-port space (the space of all ports in the network, $\mathcal{S}$) with $\mathcal{H}$, we can represent a packet traversing on a link as a point in $\{0,1\}^L \times \{1,...,P\}$ space, where $\{1,...,P\}$ is the list of ports in the network. We call the space of all possible packet headers, localized at all possible input ports in the network, the *Network Space*, $\mathcal{N}$.

Network Transfer Function, $\Psi()$: As a packet traverses the network, it is transformed from one point in Network Space to other point(s) in Network Space. For example, a layer 2 switch, that merely forwards a packet from one port to another, without rewriting headers, transforms packets only along the switch-port axis, $\mathcal{S}$. On the other hand, an IPv4 router that rewrites some fields (e.g. MAC address, TTL, checksum) and then forwards the packet, transforms the packet both in $\mathcal{H}$ and $\mathcal{S}$.

As these examples suggest, all networking boxes can be modeled as *Transformers* with a *Transfer Function* (see Figure 1), that models their protocol dependent functions. More precisely, a node can be modeled using its transfer function, T, that maps header h arriving on port p:

$$T(h,p): \quad (h,p) \rightarrow \{(h_1,p_1),(h_2,p_2),...\}$$

In general, the output of a transfer function is a set of $(header, port)$ pairs to allow multicasting.[1]

A concept we use heavily is the *network transfer function*, $\Psi(.)$. Given that switch ports are numbered uniquely, we combine all the box transfer functions into a composite transfer function describing the overall behavior of the network. Formally, if a network consists of n boxes with transfer functions $T_1(.),...,T_n(.)$, then:

$$\Psi(h,p) = \begin{cases} T_1(h,p) & \text{if } p \in switch_1 \\ ... & ... \\ T_n(h,p) & \text{if } p \in switch_n \end{cases}$$

Topology Transfer Function, $\Gamma()$: We can model the network topology using a *topology transfer function*, $\Gamma()$, defined as:

$$\Gamma(h,p) = \begin{cases} \{(h,p^*)\} & \text{if } p \text{ connected to } p^* \\ \{\} & \text{if } p \text{ is not connected.} \end{cases}$$

Γ models the behavior of links in the network. It accepts a packet at one end of a link and returns the same packet, unchanged, at the other end.

Multihop Packet Traversal: Using the two transfer functions, we can model a packet as it traverses the network by applying $\Phi(.) = \Psi(\Gamma(.))$ at each hop. For example, if a packet with header h enters a network on port p, the header after k hops will be $\Psi(\Gamma(...(\Psi(\Gamma(h,p)...)$, or simply $\Phi^k(h,p)$: each Γ forwards the packet on a link and each Ψ passes the packet through a box.

Slice: A slice, S, can be defined as (*Slice network space*, *Permission*, *Slice Transfer Function*) where *Slice network space* is a subset of the network space controlled by the slice, and Permission is a subset of {read(r), write(w)}[2]. The Slice Transfer Function, $\Psi_s(h,p)$, captures the behavior of all rules installed by the control plane of slice S. For example, a slice that controls packets destined to subnet 192.168.1.0/24 and is restricted to network ports 1, 2 and 3 can be expressed as (($ip\_dst(h)$ = 192.168.1.x , $p \in \{1,2,3\}$), rw , Ψ_s). Here, $ip\_dst(h)$ is a helper function refering to the IP destination bits in the header.

Our concept of a slice combines two notions we normally think of as very different. It describes the *implicit* slicing, when protocols coexist today on the same network using protocol IDs (e.g. TCP and UDP) or networks partitioned using Vlan IDs. It also describes the *explicit* slicing utilized by FlowVisor [5] to create independent experiments in an OpenFlow [4] network, as done in testbed networks such as GENI[3] [17].

[1] It also enables us to model load balancing boxes for which the output port is a psuedo-random function of the header bits.

[2] A real slice may have other attributes such as bandwidth reservations, but our model ignores attributes irrelevant to the analysis.

[3] In GENI parlance, the network space of a slice is called *flow space.*

3 Modeling Networking Boxes

This section is a brief tutorial on transfer functions in order to illustrate their power in modeling different boxes in a unified way. We use helper functions for clarity. We refer to a particular field in a particular protocol using helper function $protocol\_field()$. For example, $ip\_src(h)$ refers to the source IP address bits of header h. Similarly, helper function $\mathcal{R}(h, fields, values)$ is used to rewrite the *fields* in h with *values*. For example, $\mathcal{R}(h, mac\_dst(), d)$ rewrites the MAC destination address to d. Header updates can be represented by a *masking AND* followed by a *rewrite OR*.

We start by modeling an IPv4 router which processes packets as follows: 1) Rewrite source and destination MAC addresses, 2) Decrement TTL, 3) Update checksum, 4) Forward to outgoing port. Thus the transfer function of an IPv4 router concatenates four functions:

$$T_{IPv4}(.) = T_{fwd}(T_{chksum}(T_{ttl}(T_{mac}(.)))).$$

We examine each function in turn. $T_{fwd}(.)$ looks up $ip\_dst(h)$ in a lookup table and returns the output port. If lookup is modeled as $ip\_lookup(.) : ip\_dst \rightarrow port$:

$$T_{fwd}(h, p) = \{(h, ip\_lookup(ip\_dst(h)))\}$$

Similarly, $T_{mac}(.)$ looks up the next hop MAC address and updates source and destination MAC addresses. $T_{ttl}(.)$ drops the packet if $ip\_ttl(h)$ is 0 and otherwise does $\mathcal{R}(h, ip\_ttl(), ip\_ttl(h) - 1)$. $T_{chksum}(.)$ updates the IP checksum. If the focus is on IP routing, we might choose to ignore $T_{mac}(.)$, simplifying the model to $T_{IPv4}(.) = T_{fwd}(T_{ttl}(.))$ or even $T_{IPv4}(.) = T_{fwd}()$. As an example, a simplified transfer function of an IPv4 router that forwards subnet S_1 traffic to port p_1, S_2 traffic to port p_2 and S_3 traffic to port p_3 is:

$$T_r(h, p) = \begin{cases} \{(h, p_1)\} & \text{if } ip\_dst(h) \in S_1 \\ \{(h, p_2)\} & \text{if } ip\_dst(h) \in S_2 \\ \{(h, p_3)\} & \text{if } ip\_dst(h) \in S_3 \\ \{\} & \textit{otherwise.} \end{cases}$$

A firewall blocks access to certain IP address or transport port numbers, based on a set of rules that is usually called a access control list, or ACL. As an example, consider a very simple ACL with two rules: one that denies access to IP address A unless TCP port = Q; and a second rule that denies access to IP address B if source IP address is C. The transform function of the firewall is:

$$T_{acl}(h, p) = \begin{cases} \{\} & \text{if } ip\_dst(h) = A \ \& \\ & tcp\_dst(h)! = Q \\ \{\} & \text{else if } ip\_dst(h) = B \ \& \\ & ip\_src(h) = C \\ \{(h, p)\} & \textit{otherwise} \end{cases}$$

Network Address Translators (NATs) are deployed at the boundary of private networks to share one public IP address among several private hosts. A NAT remembers the source IP address and transport port number of outbound packets and rewrites the transport source port to a unique number which we represent as $nat_{out}(h)$. For the inbound packets, it looks up the destination transport port number in its lookup table using $nat_{in}(hdr)$, and returns the corresponding (ip_dst, tcp_dst) pair to be written back to header.

NAT boxes can be modeled in two levels of detail, depending on whether we are interested in exact mapping state of the NAT box or not.

Exact Transform Function of a NAT Box: Suppose a NAT box has public IP address, IP_{nat} and ports P_i and P_o are connected to private and public sides respectively. Then we can write the exact transfer function of NAT box as follows:

$$T_{nat}(h, p) = \begin{cases} \text{if } p = P_i \ \& \ nat_{out}(h) = V_{tp} : \\ \{(\mathcal{R}(h, tcp\_src, ip\_src, V_{tp}, IP_{nat}), P_o)\} \\ \text{if } p = P_o \ \& \ nat_{in}(h) = (V_{tp}, V_{ip}) : \\ \{(\mathcal{R}(h, tcp\_dst, ip\_dst, V_{tp}, V_{ip}), P_i)\} \end{cases}$$

Coarse Transform Function of a NAT Box: Let's assume that the network behind the NAT has subnet S_{nat}. Also let **X** denote a wildcard on a packet field. Then the overall behavior of the NAT can be modeled using this transform function [4]:

$$T_{nat}(h, p) = \begin{cases} \text{if } p = P_i \ \& \ ip\_src(h) \in S_{nat} : \\ \{(\mathcal{R}(h, tcp\_src, ip\_src, X, IP_{nat}), P_o)\} \\ \text{if } p = p_o \ \& \ ip\_dst(h) = IP_{nat} : \\ \{(\mathcal{R}(h, tcp\_dst, ip\_dst, X, S_{nat}), P_i)\} \end{cases}$$

We can also model a tunneling end point using a shift operator that shifts the payload packet to the right and a rewrite operator that rewrites the beginning of the header.

While modeling is trivial but tedious, we have written tools that parse router configuration files and forwarding tables to automate the process.

4 Header Space Algebra

Algorithms that compute reachability or determine if two slices can interact must determine how different spaces overlap. We therefore need to define basic set operations on $\mathcal{H}$: *intersection, union, complementation* and *difference*. We also define the *Domain, Range* and *Range Inverse* for transfer functions. The next section shows how this algebra is used.

[4]Note that by rewriting ip_dst to S_{nat}, which is a set of IP addresses and not one IP address, we mean that the output packet can be any of the packets whose IP address $\in S_{nat}$

4.1 Set Operations on $\mathcal{H}$

While set operations on bit vectors are well-known, we need set operations on *wildcard* expressions. Since all objects in header space can be represented as a union of wildcard expressions, defining set operation on wildcard expressions allows these operations to carry over to header space objects. For the rest of this paper, we overload the term *header* to refer to both packet headers (points in $\mathcal{H}$) and wildcard expressions (hyper-cubes in $\mathcal{H}$).

Intersection: For two headers to have a non-empty intersection, both headers *must* have the same bit value at every position that is not a wildcard. If two headers differ in bit b_i, then the two headers will be in different hyper-planes defined by $b_i = 0$ and $b_i = 1$. On the other hand, if one header has an x in a position while the other header has a 1 or 0, the intersection is non-empty. Thus, the *single-bit intersection* rule for $b_i \cap b_i'$ is defined as:

b_i \ b_i'	0	1	x
0	0	z	0
1	z	1	1
x	0	1	x

In the table, z means the bitwise intersection is empty. The intersection of two headers is found by applying the single-bit intersection rule, bit-by-bit, to the headers. z is an "annihilator": if any bit returns z, the intersection of all bits is empty. As an example, 11000xxx ∩ xx00010x = 1100010x and 1100xxxx ∩ 111001xx = 11z001xx = ϕ. A simple trick allows efficient software implementation. Encode each bit in the header using *two* bits: $0 \rightarrow 01$, $1 \rightarrow 10$, x $\rightarrow$ 11 and z $\rightarrow$ 00. Intersection, then, is simply an AND operation on the encoded headers.

Union: In general, a union of wildcard expressions cannot be simplified. For example, no single header can represent the union of 1111xxxx and 0000xxxx. This is why a header space object is defined as a union of wildcard expressions. In some cases, we can simplify the union (e.g., 1100xxxx ∪ 1000xxxx simplifies to 1x00xxxx) by simplifying an equivalent boolean expression. For example, 10xx ∪ 011x is equivalent to $b_4\overline{b_3} \oplus \overline{b_4}b_3b_2$. This allows the use of Karnaugh Maps and Quine-McCluskey [16] algorithms for logic minimization.

Complementation: The complement of header h — the union of all headers that do *not* intersect with h — is computed as follows:

$h' \leftarrow \phi$
for bit b_i in h **do**
  if $b_i \neq x$ **then**
    $h' \leftarrow h'$ ∪ x...x$\overline{b_i}$x...x
  end if
end for
return h'

The algorithm finds all non-intersecting headers by replacing each 0 or 1 in the header with its complement. This follows because just one non-intersecting bit (or z) in a term results in a disjoint header. For example, (100xxxxx)′ = 0xxxxxxx ∪ x1xxxxxx ∪ xx1xxxxx.

Difference: The difference (or minus) operation can be calculated using intersection and complementation. $A - B = A \cap B'$. For example:

100xxxxx − 10011xxx =
100xxxxx ∩ (0xxxxxxx ∪ x1xxxxxx ∪ xx1xxxxx
∪xxx0xxxx ∪ xxxx0xxx)
= $\phi \cup \phi \cup \phi \cup$ 1000xxxx ∪ 100x0xxx
= 1000xxxx ∪ 100x0xxx.

The difference operation can be used to check if one header is a subset of another: $A \subseteq B \Longleftrightarrow A - B = \phi$.

4.2 Domain, Range and Range Inverse

To capture the destiny of packets through a box or set of boxes, we define the *domain*, *range* and *range inverse* as follows:

Domain: The domain of a transfer function is the set of all possible *(header, port)* pairs that the transfer function accepts. Even headers for which the output is empty (i.e., dropped packets) belong to the domain.

Range: The range of a transfer function is the set of all possible *(header, port)* pairs that the transfer function can output after applying all possible inputs on every port.

Range Inverse: Reachability and loop detection computation requires working backwards from a range to determine what input *(header, port)* pairs could have produced it. If $S = \{(h_1, p_1), ..., (h_j, p_2)\}$, then the range inverse of S under transfer function $T(.)$ is $X = \{(h_i, p_i)\}]_1^n$ such that $T(X) = S$. Equivalently, $X = T^{-1}(S)$. The inverse of a transfer function is well-defined: A transfer function maps each (h, p) pair to a set of other pairs. By following the mapping backward, we can invert a transfer function.

5 Using Header Space Analysis

In this section we show how the header space analysis – developed in the last three sections – can be used for solving several classical networking problems in a protocol-agnostic way.

5.1 Reachability Analysis

Xie, et. al. [3] analyze reachability by tracing which of all possible packet headers at a source can reach a destination. We follow a similar approach, but generalize

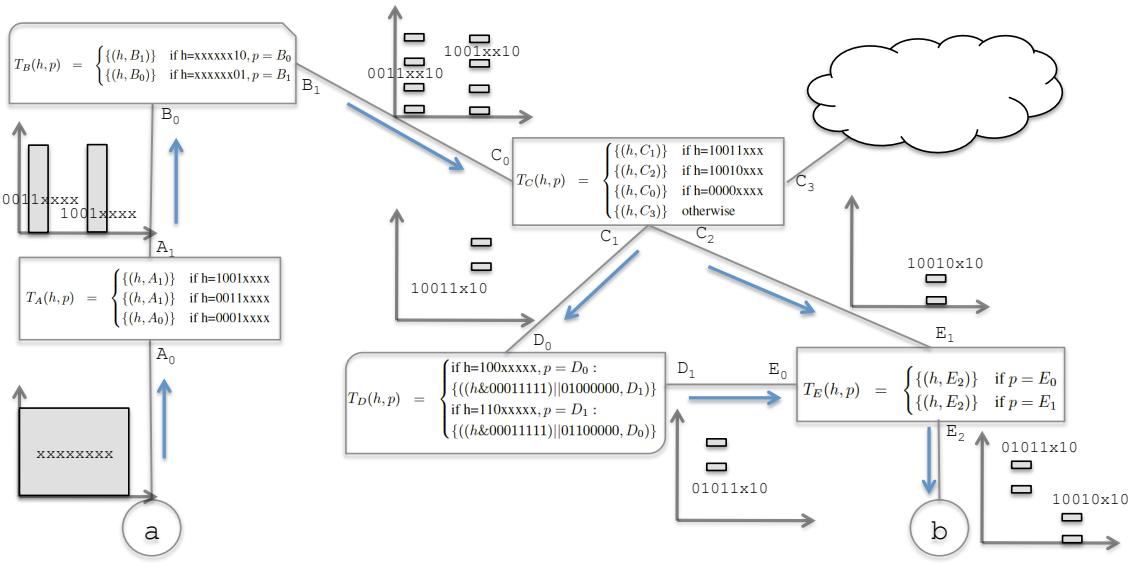

Figure 2: Example for computing reachability function from a to b. For simplicity, we assume a header length of 8 and show the first 4 bits on the x-axis and the last 4 bits on the y-axis. We show the range (output) of each transfer function composition along the paths that connect a to b. At the end, the packet headers that b will see from a are 01011x10 $\cup$ 10010x10.

to arbitrary protocols. Using header space analysis, we consider the space of all headers leaving the source, then track this space as it is transformed by each successive networking box along the path (or paths) to the destination. At the destination, if no header space remains, the two hosts cannot communicate. Otherwise, we trace the remained header spaces backwards (using the range inverse at each step) to find the set of headers the source can send to reach the destination.

Consider, for example, the question: *Can packets from host a reach host b?*. Define the reachability function R between a and b as:

$$R_{a\to b} = \bigcup_{a\to b \text{ paths}} \{T_n(\Gamma(T_{n-1}(.....(\Gamma(T_1(h,p)...))\}$$

where for each path between a and b, $\{T_1, ..., T_{n-1}, T_n\}$ are the transfer functions along the path. The switches in each path are denoted by:

$$a \to S_1 \to ... \to S_{n-1} \to S_n \to b.$$

The *Range* of $R_{a\to b}$ is the set of headers that can reach b from a. Notice that these headers are *seen at* b, and not necessarily headers transmitted by a, since headers may change in transit. We can find which packet headers can leave a and reach b by computing the range inverse. If header $h \subset \mathcal{H}$ reached b along the $a \to S_1 \to ... \to S_{n-1} \to S_n \to b$ path, then the original header sent by a is:

$$h_a = T_1^{-1}(\Gamma(...(T_{n-1}^{-1}(\Gamma(T_n^{-1}((h,b))...)),$$

using the fact that $\Gamma = \Gamma^{-1}$.

To provide intuition, we do reachability analysis for the small example network in Figure 2. Each box in Figure 2 contains its transfer function. To keep things simple, we only use 8-bit headers; since we cannot easily depict eight dimensions, we represent the first 4 bits of the header on the x-axis and the last 4 bits on the y-axis. Note that in this example, A and C are miniature models of IP routers, B is a firewall, D is a simplified NAT box and E behaves like an Ethernet switch.

Figure 2 shows how the network boxes transform header space along each path. By repeatedly applying the output of each transfer function to the input of the next transfer function in each path, the reachability function from a to b becomes:

$$R_{a\to b}(h,p) = \begin{cases} \text{if h=10010x10}, p=A_0: \\ \{(h,E_2)\} \\ \text{if h=10011x10}, p=A_0: \\ \{((h\&00011111)||01000000, E_2)\} \end{cases}$$

The range of $R_{a\to b}$, which is the final output set in Figure 2, is 10010x10 $\cup$ 01011x10. This is the set of headers that can reach b from a. To find the set of headers that a can send to b, we compute the range inverse of $R_{a\to b}$ which is 10010x10 $\cup$ 10011x10.

In section 6, we show that under the *Linear Fragmentation* assumption, which holds for most real networks, the running time of this algorithm is $O(dR^2)$, where d is the network diameter and R is the maximum number of forwarding rules in a router.

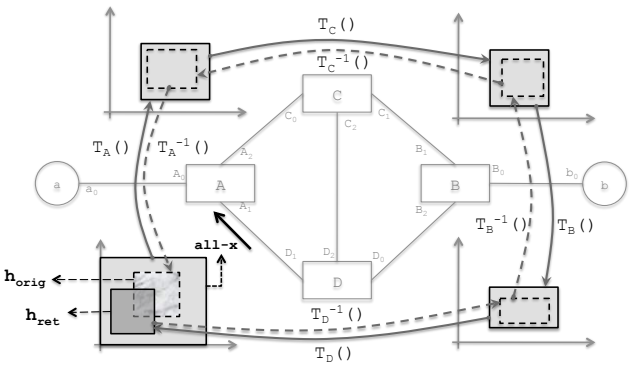

Figure 3: An example network for running the loop detection algorithm. The solid lines show the changes in the all-x test packet injected from A_1 till it returns to the injection port as h_{ret}. The dashed lines show the process of detecting infinite loop, where h_{ret} is traced back to find h_{orig}, the part of all-x packet that caused h_{ret}.

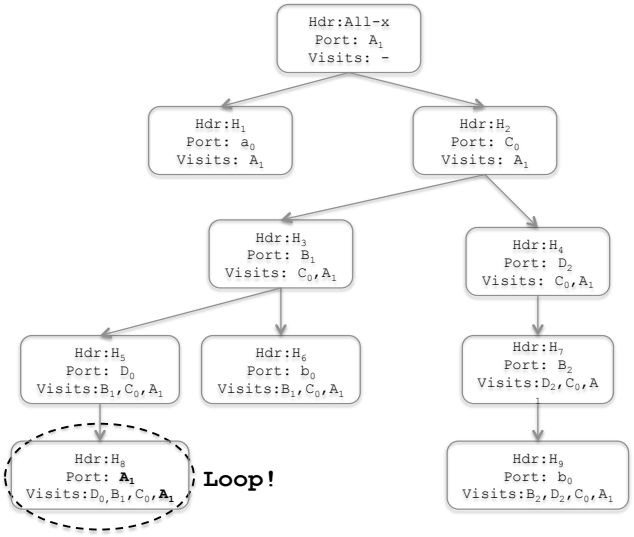

Figure 4: Example of the propagation graph for a test packet injected from port A_1 in network of figure 3.

5.2 Loop Detection

A loop occurs when a packet returns to a port it has visited earlier. Header space analysis can determine all packet headers that loop. We first describe how to detect *generic* loops and then show how to detect *infinite* loops, a subset of generic loops where packets loop indefinitely. An example of a generic, but finite loop, is an IP packet that loops until the TTL decrements to zero.

Generic Loops: Given a network transfer function, we detect all loops by injecting an *all-x* test packet header (i.e., a packet header, all of whose bits are wild-carded) from *each port* in the network and track the packet until:

- (Case 1) It leaves the network;
- (Case 2) It returns to a *port* already visited (P_{ret}); or
- (Case 3) It returns to the port[5] it was injected from (P_{inj}).

Only in Case 3 – i.e. the packet comes back to its injection port — do we report a loop. Since we repeat the same procedure starting at every port, we will detect the same loop when we inject a test packet from P_{ret}. Ignoring Case 2 avoids reporting the same loop twice.

We find loops using *breadth first search* on the *propagation graph*. For example, in Figure 3 we inject the all-x test packet into port A_1. Figure 4 is the corresponding propagation graph. Each node in the propagation graph shows the *set* of packet headers, `Hdr`, that reached a `Port` and the set of ports visited previously in their path: `Visits`. For example, in Figure 4:

$$\{(H_3, B_1), (H_4, D_2)\} = \Phi(H_2, C_0).$$

We detect a loop when `Port` is the first element of `Visits`.[6]

Finding Single Infinite Loops: Not all generic loops are infinite. For example, in the loop "$A \rightarrow C \rightarrow B \rightarrow D \rightarrow A$" in Figure 3, the header space changes as the test packet traverses the loop. Let h_{ret} denote the part of header space that returns to A_1. Then h_{orig}, defined as

$$h_{orig} = \Phi^{-1}(\Phi^{-1}(\Phi^{-1}(\Phi^{-1}(h_{ret}, A_1))))$$

is the original header space that produces h_{ret}. Figure 3 also depicts the process of finding h_{orig}.

Now, h_{ret} and h_{orig} relate in one of three ways:

1. $h_{ret} \cap h_{orig} = \phi$: In this case, the loop is surely finite. The header space that caused the loop, i.e. h_{orig}, does not intersect with the returned header space, and the loop will terminate.

2. $h_{ret} \subseteq h_{orig}$: in this case the loop is certainly infinite. Every packet header in h_{orig} is mapped by the transfer function of the loop to a point in h_{ret}. Since h_{ret} is completely within h_{orig}, the process will repeat in the next round, and the loop will continue indefinitely.

3. Neither of the above: In this case, we need to iterate again. First, note that $h_{ret} - h_{orig}$ completely satisfies Case 1's condition, and therefore cannot loop again. But we must examine $h_{ret} \cap h_{orig}$. So we redefine $h_{ret} := h_{ret} \cap h_{orig}$ and calculate the new h_{orig}. We repeat until one of the first two cases happens. The process must terminate, because at each step the newly defined h_{ret} shrinks. Hence, eventually case 1 or 2 will happen, or

[5] While we could define a loop as a packet returning to a *node* visited earlier, using ports helps detect infinite loops.

[6] If `Port` appears anywhere else in `Visits`, we terminate the branch.

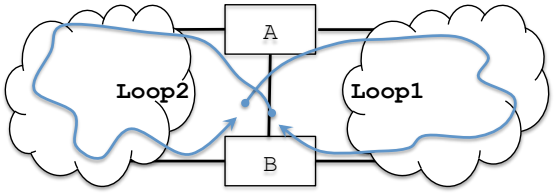

Figure 5: Example of an infinite loop consisting of multiple loops. In these cases we should consider the effect of all the loops together.

h_{ret} will be empty.[7]

Finding Infinite Multi-Loops: Infinite loops can take tortuous paths, but still be detected. For example, in figure 5, a packet can first go through loop 1, return to the injection port, and then in the next round, due to header rewrites, go through loop 2, return to the injection port, but with a header that causes the process to repeat itself indefinitely. To detect infinite multi-loops "$L_1 \rightarrow L_2 \rightarrow ... \rightarrow L_n \rightarrow L_1$", we can generalize the single infinite loop detection approach as follows: First detect all the loops originated from a single port, and for each, find h_{ret_i} and h_{orig_i}. Define:

$$h_{ret} := \bigcup_{i=1}^{n} h_{ret_i}\ ,\ h_{orig} := \bigcup_{i=1}^{n} h_{orig_i}.$$

Then perform the same check on h_{ret} and h_{orig} to decide whether the combination of loops is infinite or not.

In section 6 we show that the running time of loop detection test, under Linear Fragmentation assumption is $O(dPR^2)$ where d is the diameter of the network, R is the maximum number of rules in a router, and P the number of ports.

5.3 Slice Isolation

Network operators often wish to control which groups of hosts (or users) can communicate with each other. They might define the traffic belonging to a slice using VLANs, MPLS, FlowVisor, or – as far as we are concerned – any set of headers. A common requirement is that traffic stay within its slice, and not leak to another slice. Leakage might cause a malfunction or lead to a security breach.

Header space analysis can (1) Help create new slices that are guaranteed to be isolated, and can (2) Detect when slices are leaking traffic. We consider each use case in turn.

Creating New Slices: Creating a new slice requires identification of a region of network space that does not overlap with regions belonging to existing slices. Consider an example of two slices, a and b with regions of network space $N_a, N_b \in \mathcal{N}$,

$$N_a = \{(\alpha_i, p_i)\}_{p_i \in \mathcal{S}} \quad , \quad N_b = \{(\beta_i, p_i)\}_{p_i \in \mathcal{S}}$$

where α are headers in N_a and β are headers in N_b, and $p_i \in \mathcal{S}$ are individual ports in each slice.

If the two slices do not overlap, they have no header space in common on any common port, i.e., $\alpha_i \cap \beta_i = \phi$, for all i. If they intersect, we can determine precisely where (which links) and how (which headers) by finding their intersection:

$$N_a \cap N_b = \{(\alpha_i \cap \beta_i, p_i)\}_{p_i \in N_a \& p_i \in N_b}.$$

Set intersection could, for example, be used to statically verify that communication is allowed at one layer, or with one protocol but not with another. A simple check for overlap is extremely useful in any slicing environment (e.g. VLANs or FlowVisor) to check for run-time violations. The test can flag violations, or could be be used to create one slice to monitor another.

Detecting Leakage. Even if two slices do not overlap anywhere, packets can still leak from one slice to another when headers are rewritten. We can use a (more involved) algorithm to check whether packets can leak. If there is leakage, the algorithm finds the set of offending $(header, port)$ pairs.

Assume that slice a has reserved network space N_a = $\{(\alpha_i, p_i)\}_{p_i \in \mathcal{S}}\}$, and the network transfer function of slice a is $\Psi_a(h, p)$. Slice a is only allowed to control packets belonging to its slice using Ψ_a. Leakage occurs when a packet in slice a at any switch-port can be rewritten to fall into the network space of another slice. If packets cannot leak at any switch-port, then they cannot leak anywhere. Therefore on each switch-port, we apply the network transfer function of slice a to its header space reservation, to generate all possible packet headers from slice a. Call this the *output header set*. If the output header space of slice a at any switch port, overlaps with any other slice, then there is the potential for leaks. Figure 6 graphically represents this check.

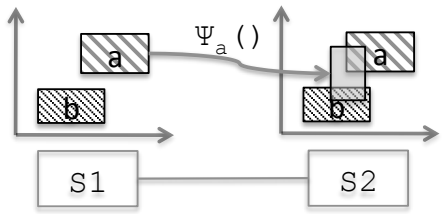

Figure 6: Detecting slice leakage. Although slice a and b have disjoint slice reservation on S_1 and S_2, but slice a's reservation on S_1 can leak to slice b's reservation os S_2 after it is rewritten by slice a's transfer function rules.

[7] IP TTL is an example of Case 3. For a loop of length n, $h_{ret} = ttl$ for $0 < ttl < 256 - n$ and $h_{orig} = ttl$ for $n < ttl < 256$. In the next round $h_{ret} := h_{ret} \cap h_{orig} = ttl$ for $n < ttl < 256 - n$. In subsequent rounds h_{ret} shrinks by $2n$ until it is empty.

In section 6 we show that the complexity of both tests is $O(W^2N)$, where W is the maximum number of wildcard expressions used to describe any slice's reservation and N is the total number of slices in the network.

6 Complexity Analysis

6.1 Reachability and Loop Detection

We analyze the complexitities of both loop detection and reachability in a uniform way because in both cases an all-x packet is pushed along a path such that the output of one transfer function is applied to the input the next transfer function. The main difference is that in loop detection we do this procedure repeatedly, once for each input port.

We start by determining the complexity of operations done at a single node. At each node, we apply the input headerspace to the transfer function of the node. If the headerspace consists of union of R_1 wildcard expressions and the transfer function has R_2 rules, then the output can be a headerspace with $O(R_1R_2)$ wildcard expressions. For example, if the first router has 1000 destination prefixes, then the output of the first switch can be a header space fragmented into 1000 rules. If this is applied to a second switch with 100 ACLs to reject traffic from certain sources, then the output of the second switch can have $1000 * 100 = 100,000$ rules. However, this is a worst case scenario and happens only if the match expressions of the transfer function are orthogonal to the input wildcard expressions as in the example.

In a real network whose purpose is to provide connectivity, as the header space propagates to the core of the network, the match patterns of forwarding rules will become less specific, therefore the space will not be divided into too many pieces. Most of the flow division happens as a result of rules that are filtering out some part of input flow (e.g. ACL rules). As a result, each of the input wildcard expressions will match only a few rules in the transfer function and generate at most cR (and not R^2) output wildcard expressions, where $c << R$ is a small constant called fragmentation factor, and R is the maximum number of wildcard expressions in any node's transfer function. We call this, the *Linear Fragmentation* assumption.

To check the validity of this assumption in real networks, we ran a test on Stanford University's backbone network. (see section 8.1 for more details) We started with an all-x flow on one of the input ports at the edge, and tracked the flow as it goes through the network. We then counted the total number of wildcard expressions that is produced at each step on all paths. Figure 7 shows the total number of wildcard expressions produced vs. the hop count for 12 different input ports. The figure suggests that after the first hop the increase rate of wildcard count is a small constant ($c < 10$), and that constant will be less than 1 after the first few hops as packets start reaching their destination.

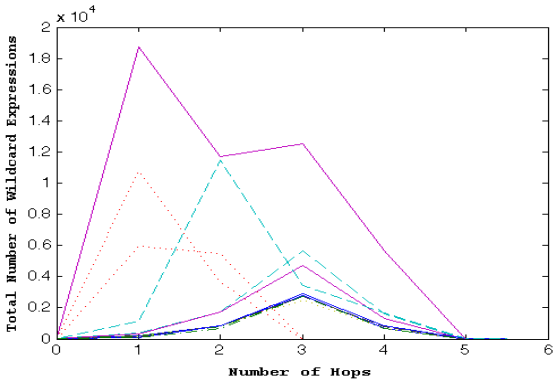

Figure 7: Total number of wildcard expressions vs. propagation hop count for 12 all-x flows injected from 12 different ports in stanford backbone network.

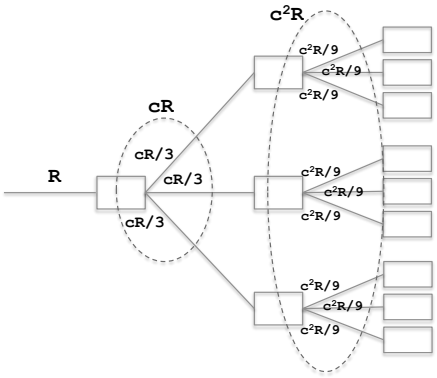

Figure 8: Fragmentation of header space as the header passes through the network.

Therefore when pushing the header along all the paths in the network, at step t, we will have at most $O(c^tR)$ wildcard expressions overall, as figure 8 suggests. To generate the headers to be used in the next step, we should apply these $O(c^tR)$ headers to $O(R)$ rules. For example in figure 8, at step 2, each of the $c^2R/9$ wildcard expressions will pass through a transfer function with R rules. Therefore the total computation at step t will be $O(c^tR^2)$. If the diameter of network, i.e. the maximum length of a path in the network, is d, then we will have at most d steps. We repeat this computation at each step and t is bounded by d. Therefore the overall complexity is less than $O(c^dR^2)$. Note that the constant c^d is very overestimated, because we used the maximum fragmentation factor, c, for all hops, while in reality this factor will be less than 1 for later hops.

Therefore the complexity of the reachability algorithm is $O(c^dR^2)$. For loop detection, we repeat the pro-

cess once for each port, therefore the complexity will be $O(c^d dR^2 P)$.

6.2 Complexity Analysis of Slice Isolation Tests

To check for isolation of two reserved slices, we need to find the intersection of one slice's network space reservation with all the other one's. If there are N slices in the network, each described by $O(W)$ wildcard expressions on each of the P ports belonging to the slice topology, then finding the intersection has $O(NW^2)$ complexity, as we need to find intersection of W wildcard expressions with NW other wildcard expressions. If the reservation of slice is different on each port belonging to the slice, then we need to repeat the computation once for each port, and hence the complexity will be $O(NW^2P)$.

To check if a new rule added to slice s will leak packets to other slices, we need to apply that rule to the header space reservation of slice s. This has complexity $O(W)$ as there are W wildcard expressions in description of slice s each of which need to be transformed by that rule. Then we need to intersect the result of the transformation with all the existing reservations. This is similar to slice isolation check, and will require $O(NW \times W) = O(NW^2)$ work.

7 Implementation

We created a set of tools written in Python 2.6 - called Header Space Library or *Hassel* - that implement the techniques described above. The source code is available here [1]. Hassel's basic building block is a header space object that is a union of wildcard expressions which implements basic set operations. In Hassel, *transfer function objects* that implement network transfer functions are configured by a set of rules; when given a header space object and port, a transfer function generates a list of output header space objects and ports. Transfer functions can be built from standard rules (i.e. by matching on an input port and wildcard expression), or from custom rules supplied by the programmer. Hassel allows the computation of the inverse of a transfer function. We also wrote a Cisco IOS parser that parse router configurations and command outputs and generates a transfer function object that models the static behavior of the router. The resulting automation was essential in analyzing Stanford's network.

Figure 9 is a block diagram of Hassel. For Cisco routers, we first use Cisco IOS commands to show the MAC-address table, the ARP Table, the Spanning Tree, the IP forwarding table, and the router configuration. The result is passed to the parser which builds transfer function objects which are then used by applications such as Loop Detection.

Disabled Optimization	T.F. Generation	Reach. Test	Loop Test
None	160s	12s	11s
(1) IP Table Compression	10.5x	15x	19x
(2) Lazy Subtraction	1x	>400x	>400x
(3) Dead Object Deletion	1x	8x	11x
(4) Lookup Based Search	0.9x	2x	2x
(5) Lazy T.F. evaluation	1x	1.2x	1.2x

Table 1: Impact of optimization techniques on the runtime of the reachability and loop detection algorithms.

Our implementation employs five key optimizations marked with superscript indices in Figure 9 that are keyed to the rows in Table 1. We briefly describe all optimizations, defering details to [**?**]. Table 1 reports the impact of disabling each optimization in turn when analyzing Stanford's backbone network. For example, loop detection for a single port with all optimizations enabled took 11 seconds: however, disabling IP compression increased running time by 19x and disabling lazy subtraction inflated running time by 400x. Since the optimizations are orthogonal, the overall effect of all optimizations is around 10,000X, transforming Hassel from a toy to a tool.

IP Table Compression: We used IP forwarding table compression techniques in [6] to reduce the number of transfer function rules.

Lazy Subtraction: The simple geometric model, which assumes the rules in the transfer function are disjoint, can dramatically increase the number of rules. For example, if a router has two destination IP addresses: 10.1.1.x and 10.1.x.x and uses longest prefix match, our original definition requires the second entry to be converted to 8 disjoint rules. To avoid this, we extended the notion of a header space object to accept a union of wildcard expressions *minus a union of wildcard expressions*: $\cup\{w_i\} - \cup\{w_j\}$. Then, when we want to apply 10.1.x.x rule to the input header space, we simply *subtract* from the final result, the output generated by the first rule. Lazy subtraction allows delaying the expansion of terms during intermediate steps, only doing so at the end. As the table suggests, performance is dramatically improved.

Dead Object Deletion: At intermediate steps, header space objects often evaluate to empty, and should be removed. Lazy subtraction masks such empty objects; so we added a quick test to detect empty header space objects without explicit subtraction.

Lookup Based Search: To pass an input header space through a transfer function object, we must find which transfer function rules match the input header space. We avoid inefficient linear search via a lookup table that returns all wildcard rules that may intersect with the (possi-

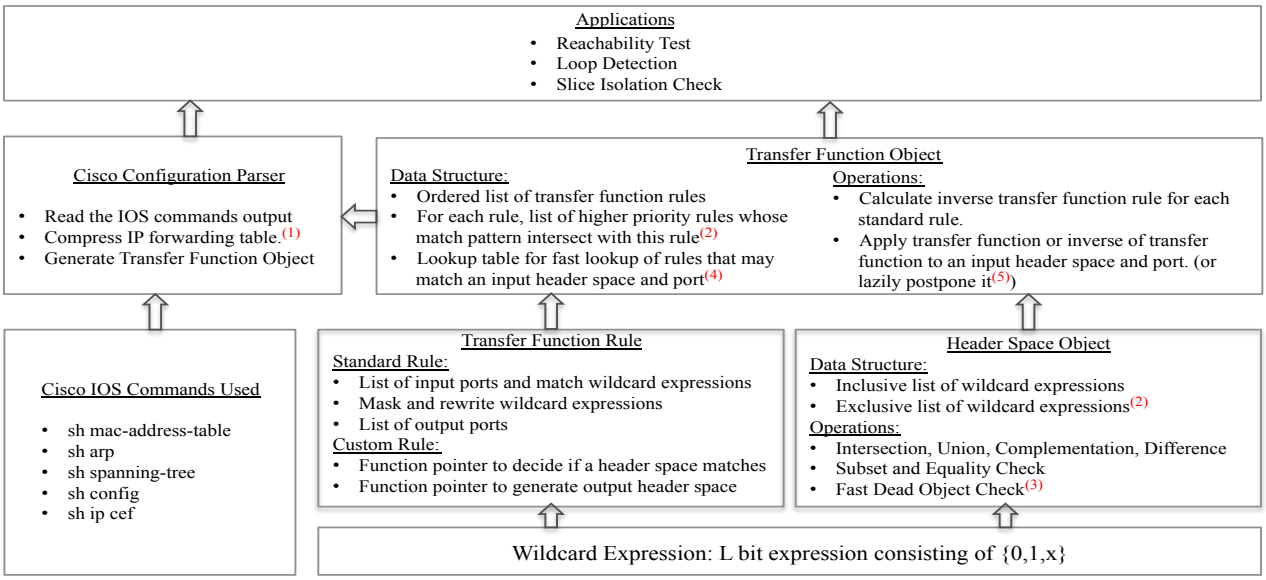

Figure 9: Header Space Library (Hassel) block diagram.

bly wildcarded) search key. This look up table is created such that we can index a wildcard expression in the table and get back all rules that match the input key. To do that, the match wildcard expression for each rule appears multiple times in the table for all possible wildcard expressions that can match it: each bit that is 0 or 1, will match x as well, and each bit that is x will match 0, 1 and x. Because the size of such table will be very large, we build the key only from two most informative nibbles in the header. Also we store a pointer to the rule instead of the rule itself.

Lazy Evaluation of Transfer Function Rules: It is possible for the header space to grow as the cross-product of the rules. For example, if some boxes forward based on D destination address, while others filter based on S source IP address, the network transfer function can have $D \times S$ fragments. If two transfer functions are orthogonal, HSL uses commutativity of transfer functions to delay computation of one set of rules until the end.

Bookmarking Applied Transfer Function Rules: Both reachability and loop detection tests, require tracing backwards using the inverse transfer function. HSL "bookmarks" or memoizes the specific transfer rules applied to a header space object along the forward path. HSL saves time during the reverse path computation by only inverting the bookmarked rules.

8 Evaluation

In this section, we first demonstrate the functionality of *Hassel* on Stanford University's backbone network and report performance results of our reachability and loop detection algorithms on an enterprise network. Then we benchmark the performance of our slice isolation test on random slices created on Stanford backbone network which are similar to the existing VLAN slices. Finally, we showcase the applicability of our approach to new protocols. All of our tests are run on a Macbook Pro, with Intel core i7, 2.66Ghz quad core CPU and 4GB of RAM. Only two of the cores were in use during the tests.

8.1 Verification Of An Enterprise Network

We ran Hassel on Stanford University's backbone network. With a population of over 15,000 students, 2,000 faculty, and five /16 IPv4 subnets, Stanford is a relatively large enterprise network. Figure 10 shows the network that connects departments and student dorms to the outside word. There are 14 operational zone (OZ) routers at the bottom connected via 10 switches to 2 backbone routers which connect Stanford to the outside world. Overall, the network has more than 757,000 forwarding entries and 1,500 ACL rules. We do not provide exact IP addresses or ports to meet privacy concerns.

We had two experimental goals: we wished to demonstrate the utility of running Hassel checks, and we wished to measure Hassel's performance in a production network. When we generated box transfer functions, we chose not to include learned MAC address of end hosts. This allowed us to unearth problems that can be masked by learned MAC addresses but may surface when learned entries expire.

Checking for loops: We ran the loop detection test on the entire backbone network by injecting test packets from 30 ports. It took 151 seconds to compress the forwarding table and generate transfer functions, and 560 seconds to run loop detection test for all 30 ports. IP table compression reduced the forwarding entries to around

4,200.[8]

The loop detection test *found 12 infinite loop paths* (ignoring TTL), such as path L1 in Figure 10, for packets destined to 10 different IP addresses. These loops are caused by interaction between spanning tree protocols of two VLANs: a packet broadcast on VLAN 1 can reach the leaves of the spanning tree of VLAN 1, where IP forwarding on a leaf nodes forwards it to VLAN 2. Then, the packet is broadcasted on VLAN 2 and is forwarded at the leaf of VLAN 2 back to the original VLAN where the process can continue.

Although IP TTL will terminate this process, if the TTL is 32 and the normal path length is 3, this consumes 10 times the normal resources during looping periods. More importantly, it shows how protocol interactions can lead to subtle problems. Each VLAN has a separate spanning tree that prevents loops but VLANs are often defined manually. More generally, individual protocols often contain automated mechanisms that guarantee correctness for the protocol by itself, but the interaction between protocols is often done manually. Such manual configuration often leads to errors which Hassel can check for; route redistribution [11] provides another example of how manual connection of different routing protocols can lead to errors.

We also found 4 other loop paths, similar to L1, L2 and L3 in Figure 10 for packets destined to 16 subnets. However, these loops were single-round loops, because when the packets return to the injection port, they are assigned to a VLAN not defined on that box, and hence will be dropped. Table 2 summarizes performance results. Note that we can trivially speed up the loop detection test by running each per-port test on a separate core.

Time to generate Network and Topology Transfer Function	151 s
Runtime of loop detection test (30 ports)	560 s
Average per port runtime	18.6 s
Max per port runtime	135 s
Min per port runtime	8 s
Average runtime of reachability test	13 s

Table 2: Runtime of loop detection and reachability tests on Stanford backbone network

Detecting possible configuration mistakes: As a second example, we considered a configuration mistake that could cause packets to loop between backbone router 1 and the Internet. Stanford owns the IP subnet 171.64.0.0/14. But not all of these IP addresses are currently in use. The backbone routers have an entry to route those IP addresses that are in use, to the correct OZ router. Also, the default route in the backbone routers (0.0.0.0/0) is to send packets to the internet. To avoid sending packets destined to the unused Stanford IP addresses to the outside world, the backbone routers have a manually installed null rule that drops all packets destined to 171.64.0.0/14, if they don't match any other rule.

[8]There were 4 routers which together had 733,000 forwarding entries because no default BGP route was received. As a result they kept one entry for every Internet subnet they knew of, but all with the same output port. IP compression reduced their table size by 3 orders of magnitude.

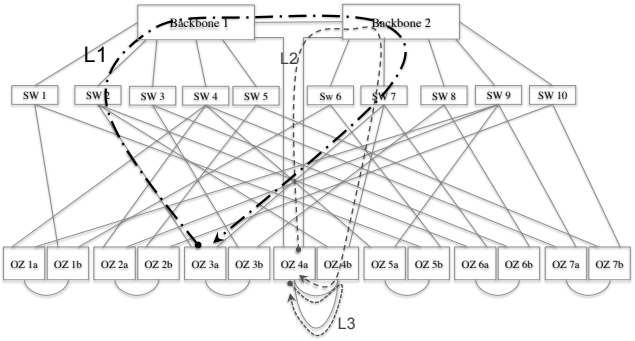

Figure 10: Topology of Stanford University's backbone network and 3 types of loops detected using Hassel. Overall, we found 26 loops on 14 loop paths. 10 of these loops, caused by packets destined to 10 IP addresses, are infinite loops masked by bridge learning. 16 other loops are single round loops.

Suppose that by mistake the null rule is set to drop 171.64.0.0/16 IP addresses (i.e., the /14 is fat-fingered to a /16). Assume that the ISP's router does not filter incoming traffic traffic from Stanford destined to Stanford. Then packets sent to unused addresses in 17.64.0.0/14 that are not in 171.64.0.0/16 will loop between the backbone routers and the ISP's router. We simulated this scenario, and the test successfully detected the loop in less than 10 minutes (as in Table 2). More importantly, the tool allowed the loop to be traced to the line in the configuration file that caused the error. In particular, the tool output shows that packets in the loop match the default 0.0.0.0/0 forwarding rule and not the 171.64.0.0/16 rule in the backbone router.

Verifying reachability to an OZ router: As a third example, we calculated the reachability function from the OZ router connected to the student dorms to the OZ router connected to the CS department. We verified that all the intended security restrictions, as commented by the admin in the config file were met. These restrictions included ports and IP addresses that were closed to outside users. Table 2 shows the run time for this test. We have heard from managers that many restrictions and ACLs were inserted by earlier managers and are still preserved because current managers are afraid to remove them. Hassel allows managers to do "What if" analysis to see the effect of deleting an ACL.

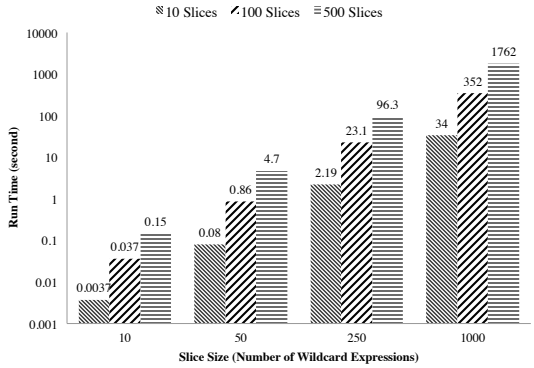

Figure 11: The time it takes to check if a new slice is isolated from other slices at reservation time.

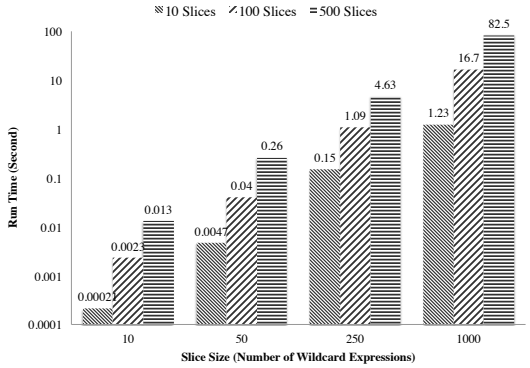

Figure 12: The time it takes to determine whether a new rewrite action will cause packets to leak between slices.

8.2 Checking Slice Isolation

Suppose we want to replace VLANs in the Stanford backbone network with the more flexible slices made possible by FlowVisor. Stanford's VLANs mostly carry traffic belonging to a particular subnet — e.g. VLAN 74 carries subnet 171.64.74.0/24. VLAN 74 is equivalent to a FlowVisor slice across the same routers with header space: $ip\_dst(h) = 171.64.74.0/24$ or $ip\_src(h) = 171.64.74.0/24$.

We would like to understand how quickly we can create new and flexible slices on-demand in the Stanford network. Recall from Section 5.3, we need to perform two checks:

1. When creating a slice, we need to make sure its header space does not overlap with an existing slice.
2. Whenever a rewrite action is added to a slice, we need to check that it cannot cause packet leakage.

We generated random slices with a topology similar to the existing VLANs, as follows: for each slice, we randomly pick two operational zones in Stanford together with all router ports and switches that connect them. Then we add random pieces of header space to each slice by picking X source or destination subnets of random prefix length (or random TCP ports) to union together. X, the number of wildcard expressions used to describe a slice, denotes the slice's *complexity*. While all existing VLAN slices in Stanford require fewer than 10 wildcard expressions[9], we explored the limits of performance by varying X from 10 to $1,000$. We ran experiments to create new slices of varying complexity, X, while there were 10, 100 or 500 existing slices.

In the first experiment, we create a new slice, and the checker verifies isolation by looking for intersection with all the existing slices. This test is done every time a new slice is created, and we hope it will complete in a few minutes or less. In the second experiment, we emulate the behavior of adding a new rewrite action. We generate random rewrite actions, and the checker checks to see if it could possibly cause a packet leak to another slice. This test has to be run every time a new rule is added, so it needs to be really fast.

Figure 11 and 12 shows the run time of our tests. As the figures suggest, if the slices are not very complex (can be explained with fewer than 50 wildcard expressions)[10], then the tests run almost instantly. Surprisingly, the tests are very fast even when there are 500 slices: more than adequate for existing networks. If there are only 10 or 100 slices, the checks can be done on very complex slices. The experimental run times matches the expected complexity in Section 5.3, which is *quadratic* in the number of wildcard expressions per slice and *linear* in the number of slices.

8.3 Debugging a Protocol Design

This section describes a plausible scenario in which a loop is caused by a protocol design mistake. The scenario allows the loop size to be parameterized to examine how detection time varies with loop size. It also showcases how HSL can model IP options and other variable length fields using custom transfer function rules.

In our scenario, Alice – a networking researcher – invents a new loose source routing protocol, IP*. IP* allows a source to specify the sequence of middle boxes that a packet must pass through. Alice's protocol has the header format shown in Figure 13.a. IP* works exactly like normal IP, except that it updates the header at the first router where a packet enters the IP* network. Figure 13.b shows an example of header update for a stack size of three. The header update operation sets the current source address to the "sender IP address" field, rewrites the destination IP address to the address at the top of the stack, and rotates all the IP addresses in the stack. After

[9] In most cases, two expressions suffice.

[10] This includes wildcard expressions that are included in, or excluded from, the definition of a slice.

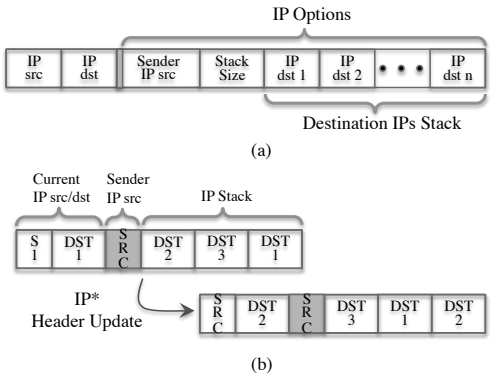

Figure 13: (a) Header format for IP* protocol. (b) IP* stack rotation: 1. The packet's IP source is replaced by the Sender IP source. 2. The packet's IP destination is replaced by the top of the stack. 3. The stack is rotated.

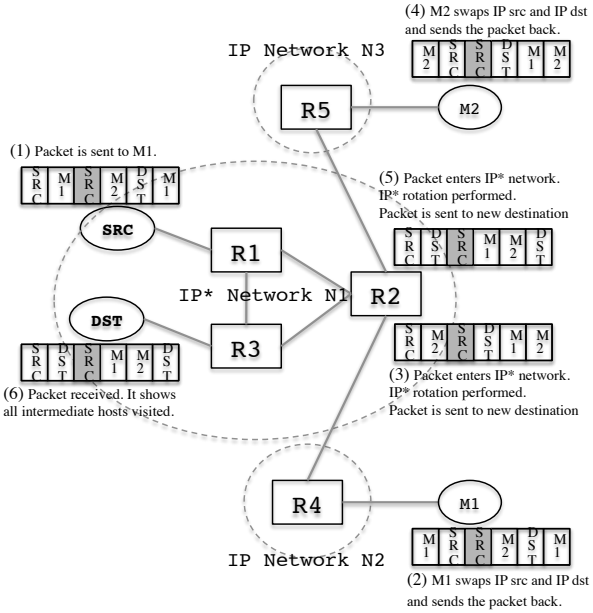

Figure 14: Example of an IP* network where a packet sent from SRC to DST visits middleboxes $M1$ and $M2$. The figure labels 6 steps of packet processing along with the transformed header at each step. $R2$ is the entry point to the IP* network which performs IP* header updates.

processing, a destination middlebox swaps the destination and source IP addresses and resends the packet to a router. Alice designs IP* to allow tunneling across existing IP networks.

Alice tries IP* in the network topology of Figure 14 and verifies that packets are successfully routed via middle boxes $M1$ and $M2$. Figure 14 also shows how the packet header changes as it passes through $M1$ and $M2$ to final destination DST in 6 steps. At DST, the stack contains the IP address of all middleboxes visited.

To continue her verification of IP*, Alice tries the more complex network in Figure 15 where the destination is attached to a different IP* network than the source. Instead of deploying a real network, she uses the loop detection algorithm from Section 5.2 and finds several loops. The most interesting one is an infinite loop consisting of the two strange loops below:

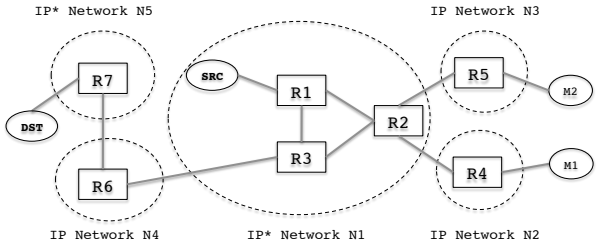

Figure 15: Alice's second network topology can cause infinite loops.

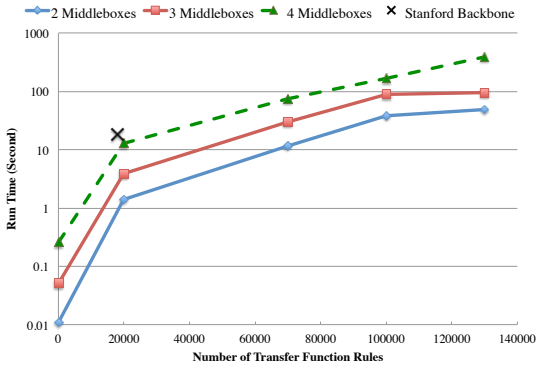

Figure 16: Running time of loop detection algorithm on Ip* network

1) $R_2 \rightarrow R_5 \rightarrow M_2 \rightarrow R_5 \rightarrow R_2 \rightarrow R_4 \rightarrow M_1 \rightarrow R_4 \rightarrow R_2 \rightarrow R_3 \rightarrow R_6 \rightarrow R_3 \rightarrow R_2$
2) $R_2 \rightarrow R_4 \rightarrow M_1 \rightarrow R_4 \rightarrow R_2 \rightarrow R_5 \rightarrow M_2 \rightarrow R_5 \rightarrow R_2 \rightarrow R_3 \rightarrow R_6 \rightarrow R_3 \rightarrow R_2$

Alice now realizes that having a second IP* network in the path can cause loops. She removes the concept of a "first" router, instead adding a pointer that describes the middlebox to visit next. We should be clear that by no means are we proposing IP* as a viable protocol. Instead, we hope this example suggests that Hassel could be a useful tool for protocol designers as well as network managers. While this particular loop could be caught by a simulation, if there were many sources and destinations and the loop was caused by a more obscure precondition, then a simulation may not uncover the loop. By contrast, static checking using Hassel will find all loops.

Figure 16 shows the *per-port* performance of our infinite loop detection algorithm for the ports participating in the loops. We varied the number of middle boxes connected to R_2 (e.g., $M1$ and $M2$) and the number of router forwarding entries. For four middle boxes, the

```
def detect_loop(NTF, TTF, ports, test_packet):
    loops = []
    for port in ports:
        propagation = []
        p_node = {}
        p_node["hdr"] = test_packet
        p_node["port"] = port
        p_node["visits"] = []
        p_node["hs_history"] = []
        propagation.append(p_node)

        while len(propagation)>0:
            tmp_propag = []
            for p_node in propagation:
                next_hp = NTF.T(p_node["hdr"],p_node["port"])
                for (next_h,next_ps) in next_hp:
                    for next_p in next_ps:
                        linked = TTF.T(next_h,next_p)
                        for (linked_h,linked_ports) in linked:
                            for linked_p in linked_ports:
                                new_p_node = {}
                                new_p_node["hdr"] = linked_h
                                new_p_node["port"] = linked_p
                                new_p_node["visits"] = list(p_node["visits"])
                                new_p_node["visits"].append(p_node["port"])
                                new_p_node["hs_history"] = list(p_node["hs_history"])
                                new_p_node["hs_history"].append(p_node["hdr"])
                                if len(new_p_node["visits"]) > 0 \
                                    and new_p_node["visits"][0] == linked_p:
                                        loops.append(new_p_node)
                                        print "loop detected"
                                elif linked_p not in new_p_node["visits"]:
                                    tmp_propag.append(new_p_node)
            propagation = tmp_propag

    return loops
```

Figure 17: Loop Detection Code

loop has a length of 72 nodes! Finding a large and complex loop in less than four minutes — for a network with 100,000 forwarding rules and *custom* actions — using less than 50 lines[11] of python code (figure 17) demonstrates the power of the Header Space framework.

9 Limitations

Header space analysis is designed for static analysis, to detect forwarding and configuration errors. It is no panacea, serving as one tool among many needed by protocol designers, software developers, and network operators. For example, while header space analysis might tell us that a routing algorithm is broken because routing tables are inconsistent, it does not tell us why. Even if the routing tables are consistent, header space analysis offers no clues as to whether routing is efficient or meets the objectives of the designer. Despite this, header space analysis could play a similar role in networks as post-layout verification tools do in chip design, or static analysis checkers do in compilation. It checks the low level output against a set of universal invariants, without understanding the intent or aspirations of the protocol designer. To analyze protocol correctness, other approaches, such as [13] should be used.

Similarly, while our approach can pinpoint the specific entry in the forwarding table or line in the configuration file that causes a problem, it does not tell us how or why those entries were inserted. This is because our model captures the current state of the system, and not how the state was generated. Finally, like all static checkers, our formalism and tools cannot deal well with churn in the network, except to periodically run it based on snapshots: thus it can only detect problems that persist longer than the sampling period.

[11] Not counting the underlying Hassel implementation

10 Related Work

The notion of a transfer function in our work is similar to ASE mapping defined in axiomatic routing [13], where the authors develop tools to analyze a variety of protocols. Header space analysis makes no attempt to analyze protocols; instead, it tackles the problem of independently checking if their output creates conflicts. Roscoe's predicate routing [7] introduces the notion of pushing a test packet (as used in our reachability and loop detection algorithms) when designing routing mechanisms, rather than for static checking as we do. Xie's reachability analysis [3] uses test packets to determine reachability (not loop detection) for the special case of TCP/IP networks. The static analysis tools described in [8, 9, 10] are designed specifically for TCP/IP firewalls, and Feamster's work in [14] finds reachability failures in BGP routers.

Header space analysis is broader in two ways. First, it is a framework that can identify a range of network configuration problems. Second, the algorithms developed in this framework are independent of protocols. Finally, note that model checking, SAT solvers, and Theorem Provers are other commonly used frameworks for verification. However, when these frameworks detect a violation of a specification (e.g., reachability) they are limited to providing a *single counterexample* and not the *full set* of failed packet headers that header space analysis provides.

11 Conclusions

Our paper introduces *Header Space Analysis*: a general framework for reasoning about arbitrary protocols, and for finding common failures, or accidents. By parsing routing and configuration tables automatically, we show that header space analysis can be used in existing networks where protocol interactions are increasingly complex. As we saw in the Stanford backbone and noted by [11], while individual protocols use automated mechanisms to prevent internal problems, managing many protocols simultaneously is a manual and error-prone business. Header Space Analysis can also be used in emerging networks, where new protocols can be added dynamically. It can give network operators the confidence to adopt new protocols, or new slicing mechanisms — the framework can be used to create comprehensive checkers that can be used by network operators to pro-actively avoid (or retroactively investigate) accidents.

Our personal story is that we set out to create a geometric model to better understand slicing. Along the way, we discovered how simple it is to analyze networks using Network Transfer Functions (Ψ). We found that checking for a given violation was surprisingly easy when expressed using this high level abstraction, allowing elegant expression and simple implementation as our

code snippets (see Figure 17) suggest.

We have work to do to improve the performance of our prototype Hassel implementation. A first round of optimization reduced running time by five orders of magnitude, making Hassel perform well for production networks with a few dozen routers, adequate for most enterprises and campuses. With simple fixes (e.g. exploiting 64-bit arithmetic, using compiled languages rather than Python, and harnessing the parallelism of multicore chips) we expect another 2-3 orders of magnitude performance gain. We expect running time of a complete loop test for a campus backbone to be reduced from about 1,000 seconds to less than 10 seconds. Even more optimizations are apparent, such as using a Karnaugh-Map to reduce the size of header space after each transformation. There is also scope for checking updates incrementally, by analyzing loops and reachability once, and then seeing how a new rule (or slice) changes the result.

We have other work ahead of us: we would like to create tools that create test packets to dynamically sample header space to detect faults in an operational network. We also hope to explore the notion of how secure, or how fault-tolerant, a network is by finding the "distance" between the current status of the network, and different failure conditions — analogous to Hamming distance.

Finally, preliminary experiments show that HSA can be effectively combined with model checking [15]. Header space, and the errors we are looking for, can be expressed as a set of constraints in a SAT solver, and a Theorem Prover can prove or provide a counterexample for it. While fast, the approach is limited to producing a single counterexample, rather than providing the full set of offending packet headers. But the two techniques can be combined: initially, errors can be found quickly using the SAT solver, then we can use header space analysis to reveal the full scope of the problem.

Accidents will happen in the best regulated of networks; but the judicious use of checkers such as ours can reduce their probability.

References

[1] Header Space Library (Hassel) `http:/stanford.edu/~kazemian/hassel.tar.gz`

[2] T. V. Lakshman and D. Stiliadis, *High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching*, Proc. ACM SIGCOMM, pp. 191–202, Sept. 1998.

[3] G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford, *On Static Reachability Analysis of IP Networks*, Proc. IEEE INFOCOM Conference, 2005.

[4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, *OpenFlow: Enabling Innovation in Campus Networks*, ACM SIGCOMM Computer Communication Review, Volume 38, Number 2, April 2008.

[5] R. Sherwood, G. Gibb, K.K Yap, G. Appenzeller, M. Casado, N. McKeown, G. Parulkar, *Can the Production Network Be the Test-bed?*, USENIX Symposium on Operating Systems Design & Implementation (OSDI) 2010.

[6] R. Draves, C. King, S. Venkatachary, B. Zill, *C*onstructing optimal IP routing tables, 1999 Proc. IEEE INFOCOM, March 1999.

[7] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, P. Jardetzky *Predicate Routing: Enabling Controlled Networking* Proc. of 1st Workshop on Hot Topics in Networks, 2002

[8] L. Yuan, J. Mai, Z. Su, H. Chen, C-N. Chuah, and P. Mohapatra, *FIREMAN: A Toolkit for Firewall Modeling and Analysis*, 2006 IEEE Symposium on Security and Privacy, pp. 213-228

[9] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool. *Firmato:A novel firewall management toolkit*, Proc. 20th IEEE Symposium on Security and Privacy, 1999.

[10] A. Mayer, A. Wool, and E. Ziskind, *Fang: A firewall analysis engine*, Proc. IEEE Symposium on Security and Privacy, 2000.

[11] F. Le, G. Xie, D. Pei, J. Wang, and H. Zhang, *Shedding Light on the Glue Logic of the Internet Routing Architecture*, Proc. ACM SIGCOMM Conference, August 2008

[12] F. Le, G. Xie, and H. Zhang, *Understanding Route Redistribution*, Proc. IEEE ICNP 2007 Conference, Beijing, China, October 2007.

[13] M. Karsten, S. Keshav , S. Prasad , M. Beg *An Axiomatic Basis for Communication* Proc. ACM SIGCOMM Conference, August 2007

[14] N. Feamster, H. Balakrishnan, *Detecting BGP configuration faults with static analysis*, Proc. of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2, NSDI'05,

[15] E. M. Clarke, O. Grumberg, D. A. Peled, *Model Checking*, MIT Press, 1999.

[16] S. Brown, Z. Vranesic, *Fundamentals of Digital Logic with Verilog Design*, McGraw-Hill, 2003.

[17] Global Environment for Network Innovations (GENI), `http://www.geni.org`

[18] The Health Insurance Portability and Accountability Act (HIPAA), `http://www.hhs.gov/ocr/privacy/`