FACILITATING DYNAMIC NETWORK CONTROL WITH SOFTWARE-DEFINED NETWORKING

A Thesis
Presented to
The Academic Faculty

by

Hyojoon Kim

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2015

FACILITATING DYNAMIC NETWORK CONTROL WITH SOFTWARE-DEFINED NETWORKING

Approved by:

Dr. Nick Feamster, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Nate Foster
Department of Computer Science
Cornell University

Dr. Ellen Zegura
School of Computer Science
Georgia Institute of Technology

Dr. Yoshio Turner
Founder
Banyan

Dr. Mostafa Ammar
School of Computer Science
Georgia Institute of Technology

Date Approved: 1 May 2015

To my family, who made everything possible.

ACKNOWLEDGEMENTS

This dissertation would have not been possible without all the support, guidance, and encouragement I have received along the way.

I want to deeply thank my advisor, Professor Nick Feamster, for guiding and supporting me since the start of my second year as a master student at Georgia Tech, and being patient and sticking with me until I finally got my Ph.D. degree. For six years, he never (explicitly) pushed me to finish up my study, and he never made me worry about funding. I will never forget the moment when he took that leap of faith and told me that he will fund me as a graduate research assistant (GRA) when I was just a master student. I know how many sleepless nights he spends to get grant money to support all his students. He once spent an entire day writing a proposal right next to a coffee machine in Klaus so that he can save the time on getting coffee (plus to get an endless supply of it). And he took short naps beside that coffee machine, on the floor. I am truly grateful.

Nick has newly defined what dedication and enthusiasm really means to me. The way he dives into topics with details, how he leads discussions, and the time and effort he puts into research and making things happen is still mind blowing to me. The focus he shows when he really means it and the lightening ideas that come out during meetings always amaze me. I want to thank Nick for being an exceptional role model as a scientist, researcher, professor, lecturer, presenter, writer, advisor, fund raiser, and a person with great sense of humor.

I want to thank Dr. Russ Clark, who gave me tremendous help. He actively got involved in our research projects and gave me great ideas and feedback based on unimaginable amount of experience he has in managing real networks. The data and resource he fetched for me and for my research were incredibly useful. I have no doubt that I would have not

finished my studies without him. I also want to thank the whole OIT team at Georgia Tech for supporting my research, fetching data, backing up data, and managing our machines.

I want to thank Professor Aditya Akella and Theo Benson (now an assistant professor at Duke University) for the great collaboration we had with our awesome IMC paper. I also want to thank Professor Marshini Chetty, now at University of Maryland - College Park (and Nick's wife!), for the collaboration we had with the uCap project and our great CHI 2015 paper. Josh Reich was like an angel from Princeton University who decided to come and collaborate with me so that we can finally make our Resonance/Kinetic work truly awesome, and eventually publish a full paper at NSDI 2015. Josh is an exceptional programmer with a brilliant mind and a great sense of humor. I also want to thank Professor Jennifer Rexford from Princeton University for constantly giving me invaluable feedback on our project, and accepting me as a Ph.D student at Princeton University in 2010. Although I decided to remain at Georgia Tech to continue to work with Nick then, it's all good now because all three of us are currently at Princeton University. I want to thank Jen again for giving me the opportunity to work at Princeton University after the completion of my Ph.D. study. It's an honor to be at Princeton University and to work with such a great mind (and super nice!).

I want to thank all the folks at HP Labs - Palo Alto for having me as their research intern multiple times to work on interesting problems and projects. Coronet is a big part of my thesis, and I hope it will see light and be applied on real networks in the future. I want to specially thank Dr. Yoshio Turner and Dr. Renato Santos, who were my great mentors at HP Labs. I wish them luck, truly from my heart, on their new path as in startups.

I want to thank my thesis committee members, Professor Mostafa Ammar, Professor Ellen Zegura, and Professor Nate Foster (along with Nick and Yoshio) for giving me great comments and feedback on my work, guiding me through my thesis, and attending the proposal and defense of my dissertation, and of course, approving it.

I had a great time at Georgia Tech because of all my great friends and colleagues. I want

to thank all the NTG and non-NTG lab mates: Abhinav, Ankur, Arpit, Ben, Bilal, Binh, Boris, Chaitrali, Demetris, Giuseppe, Illias, Maria, Miseon, Murtaza, Robert, Saamer, Said, Sam, Samantha, Sarthak, Sathya, Sean, Shahbaz, Shuang, Srikanth, Steve, Swati, Valas, Walter, and Yogesh. I will never forget the moments we shared of lives together, having regular lunch breaks, playing soccer, running, bicycling (once, I think), having special and normal dinners, drinking, and partying. It's always great to be friends with smart, sharp, nice, and incredibly fun people from all over the world.

I especially want to say hi and pray for our friend, Saamer Akhshabi, who suddenly passed away and left us in March, 2014. Dr. Saamer Akhshabi, yes, I cheated when we had that drinking match years ago. You actually won. You have proven to be smarter and also a better drinker than me. I will see you later up there, bro.

I also want to thank my Korean friends (a.k.a., the Korean Mafia, as Valas used to say): Byoungyoung, Changhyun, Chayoung, Dongchan, Heejoong, Hwajung, Ilho, Jaegul, Jaewook, James, Janghaeng, Jaeyeon Jayoung, Joonseok, Jungju, Minsung, Moonkyung, Sangmin Park, Sangmin Shim, Seungyeon, Sungwon, Wonwoo, and Younggyun. Because of these smartest young people from Korea, I was able stay sane as they constantly reminded me of my warm home country.

I thank my all my best friends in Korea, Hangsang, Soonhyuck, Sunghwan, and Wooyong for keeping in touch with me even I was hard to reach, always contacting me first and asking how I was doing. Love you guys. Wes Anderson was my first roommate here in Atlanta, and we stayed in touch time to time. I thank him and his whole family for caring for me, inviting me on holidays, feeding me, and giving me presents. I have to somehow find out a way to pay back their love. I thank all my friends in church who prayed for me, and God who never gave up on me.

I want to thank my father and mother, who supported me through all my studies, actually through all my life. Now it's time to pay back the love and support they gave me. I am quite sure I will never be able to fully pay back what I have received so far, but I will try.

Thanks to Hyoshin, my older siser, for reaching out to me time to time, never questioning my ability, and buying me stuff or just giving me allowance few times. I also thank her for allowing me to drive her car whenever I visited Korea, and forgiving me when I once trashed her car. Thank you.

I want to thank both my maternal and paternal grandmothers for their love and support. They always screamed with joy when I called them time to time, always saying that nothing is more important than my health and wellbeing, even above my studies. I want thank my grandfathers up there too, who will be jumping with joy together, congratulating the completion of my doctoral degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Network management is complex and error-prone; this complexity and brittleness has several causes. First, much of today's network management process remains low-level and manual: operators must update router and switch configuration files to change the network's forwarding behavior (*e.g.*, quarantining, garden-walling, or rate limiting a host) while configuration language is unintuitive and hard to understand. Second, the configuration is distributed all over the network instead of residing in one location, which makes it harder to manage. Third, network conditions are dynamic. Traffic patterns change, hosts arrive and depart, topologies change, intrusions occur, and so forth, thus the network's configuration needs continuous updates by operators. Yet, we understand very little about the nature of network configuration and its changes. We also lack alternate solutions that can help operators to express today's complex network policies with less misconfigurations.

This dissertation analyzes traditional network management methods to better understand the problem in network configuration, and presents better network management solutions that help operators to configure and program their network in a concise, intuitive, and less error-prone way. First, we analyze over five years of historical network configuration files from two big campus networks and show that a network can experience a lot of changes; number of lines that change in all configuration files ranges from 200,000 to 800,000 lines per year. Based on our findings, this dissertation presents two distinct solutions that are both based on Software-Defined Networking (SDN). Kinetic is a domain specific language and network control platform that enables operators to write programs that can automatically reconfigure the network in face on arbitrary network events. Kinetic also automatically verifies the correctness of these control programs with respect to user-specified temporal properties. Coronet is an SDN-based service that provides automated

recovery in face of an unexpected but common network event: data-plane failures such as switch and link failures. Coronet provides other SDN control applications with the abstraction of a reliable virtual topology and achieves fast failover by updating small number of table entries upon failure detection rather than the much larger number of individual flows that are assigned to those table entries.

Our user study of Kinetic with several hundred network operators demonstrates that Kinetic is intuitive and usable, and our performance evaluation shows that realistic Kinetic programs scale well as the number of policies and the size of the network increase. Evaluation of Coronet against four Mininet-emulated network topologies demonstrates that Coronet can provide fast failover (maximum less than two seconds), and encoding paths as trees achieves network load balancing before and after failure close to the performance of end-to-end routing paths, but with lower failover time and much lighter use of OpenFlow table resources at switches.

CHAPTER I

INTRODUCTION

Today, computer networks are large and complex. For example, our study shows that a large campus network has around 1,000 to 1,700 network devices (*e.g.*, routers, firewalls, switches), and experiences 1,000 to 18,000 changes per month [46]. Despite of the growth of size and complexity, however, the method and tools used for configuring a network have not changed much. Network operators still perform complex network tasks by typing in low-level commands in configuration files in individual network devices, and they make frequent mistakes when making changes to network configuration [22, 67].

There are several reasons why network configuration is so hard and error-prone. First, network configuration process remains low-level and mostly manual. Operators still use command line interface (CLI) for interacting with network devices, and use low-level, unintuitive commands to configure routers, firewalls, and switches. Second, configuration files reside in network devices, thus are distributed all over the network, making it hard to manage. For example, Feamster *et al.* has concluded that distributed configuration across routers is the cause of the majority of the configuration faults in ASes [22]. Third, network conditions are dynamic, thus frequent and continuous network configuration updates are inevitable. Traffic patterns and volume change, host devices arrive and depart, new departments appear and move, and so forth. Such planned or unplanned arbitrary network events likely cause big or small updates to the network. Operators generally rely on manual updates to configuration files, but also often use ad hoc scripts or tools to automate frequent changes to the configuration. However, such scripts or tools do not prevent misconfiguration, but in fact, can even introduce more misconfigurations [22].

To shed light on understanding how network is configured, and to explore better ways

to configure networks, this dissertation focuses on analyzing current configuration practices and designing systems that make network configuration easier, more intuitive, and less error-prone. First, we study how network is configured and how it *changes* over time through static and longitudinal analysis. Such analytical study helps us better understand current practices and identify problems in configuration methods. With lessons learned from our analysis, this dissertation presents a radically different way of configuring the network using SDN. This dissertation presents the design and implementation of systems that operators can use to build network control programs that perform automatic adjustment of network behavior based on planned and unplanned network events, and most importantly, in a more concise, intuitive, and less error-prone way.

1.1 Challenges in improving network configuration methods

It is fundamentally hard to introduce better configuration solutions as the market is dominated by few vendors, who predominantly constraint operators to use methods that require typing in low-level commands directly in network devices. The API is also not open source, thus has limited programmability. Such vertically integrated and closed "blackbox" devices make it hard to test and introduce new functions or methods. The advent of Software-Defined Networking (SDN) opens up a lot of possibilities. SDN separates network control and data planes; a logically centralized controller makes traffic forwarding decisions and programs switches to carry out traffic forwarding in the data plane. SDN enables global network visibility, programmability, and extensibility. However, this approach introduces another set of challenges.

SDN is an open-world without guidance. SDN is a powerful approach to managing computer networks [23]. However, it opens up new challenges, especially in searching for the right abstraction. While there are much work on standardizing the interface between the data plane and control plane ("southbound" API, like OpenFlow [84]), not much research has been done in coming up with the right abstraction for the language for building SDN

control applications, the controller platform itself, and the communication between SDN control applications and the platform ("northbound" API). The openness of SDN has encouraged plethora of controller platforms to emerge, such as NOX [79], POX [86], Floodlight [26], HP VAN [36], Open Daylight [83], Ryu [92], Pyretic [75], and so forth. Yet current SDN lacks *guidance on how to implement an efficient and correct control program*. Such lack of guidance fails to prevent inefficient or even buggy control programs to appear, which can introduce unexpected and undesired behavior in the network.

Generally hard to verify and guarantee correct network behavior. As the consequence of misconfiguration or faulty control logic can be serious, it is important to have certain sense of guarantee that the control program correctly encodes the intended network policy. However, it is generally hard, if not impossible, to automatically verify the correctness of the control program that operator implemented with a general-purpose programming language. Manual inspection is too slow, inefficient, and bound to human error. There are related work that performs verification on snapshots of network's data-plane state ([43,45,68]), but there are not much work on verifying the logic of the control program that generates the data-plane state in the first place.

Scaling to large networks. The size of a typical enterprise network continues to grow, and specialized networks such as data center networks usually house hundreds of switches with thousands of links. The challenge is to provide a control platform and service that allow operators or developers to build SDN control programs that scale to increasing number of links, switches, hosts, and network events. Specifically, SDN control programs should react to arbitrary network events and rapidly recompile and produce new forwarding rules that are installed in the data plane. The number of necessary updates to the data plane should also be minimized to ensure fast network reconfiguration in face of planned and unplanned events. The language and control platform should bare such responsibility, rather than relying on the operator to optimize her program to be efficient.

1.2 Towards better network management

Thesis statement: This dissertation identifies that lots of network changes are caused by various types of network events, and posits that network events and dynamic reactions to them should be encoded in the network policy in the first place by the operator, or should be handled transparently by a dedicated module for certain common network events. We use software-defined networking to present a language and systems that help operators to write dynamic network control programs in a concise, intuitive, and less error-prone way compared to legacy methods. We bring guarantees and correctness verifications to network control programs through software.

1.3 Contribution

This dissertation makes following contributions in defense of the thesis statement:

The first extensive longitudinal analysis study on network configuration changes. This dissertation presents the first extensive longitudinal analysis on how network configuration changes over time. By inspecting and analyzing over five years of configuration file change logs from network devices from two large campus networks, we uncover and confirm that network configuration changes a lot, which was only based on anecdotes from network operators. We have found that a campus network may experience anywhere from 1,000 to 18,000 changes per month, and a variety of network-wide tasks contribute to changes in devices, including provisioning, management, traffic engineering, security, and QoS.

Encoding network events and dynamic changes in forwarding behavior in network policy. This dissertation focuses on a variety of network events and posits that such events and corresponding reactions should be modeled into the network policy in the first place. Instead of thinking a network-wide network policy as a combination of multiple static network policies with network events that trigger transitions between them, we present a domain specific language and platform that lets operators to encode network events into

the network policy. Through an extensive user study and quantitative evaluation, we show the benefit of such approach.

Making network configuration easier with abstractions, composition, and reusable modules. This dissertation presents high-level abstractions to operators to make network configuration easier. Our system allows to compose multiple network control programs that have distinctive network tasks together to express more complex network policies. This dissertation presents modular services and network control programs that can be reused by operators, promoting code reusability. Coronet is an SDN service module that specifically handles unplanned data-plane failures, and guarantees to swiftly and efficiently recover from any single link or switch failure. This brings the developers and operators together to create different SDN control "apps" and services that can be reused and shared among the community.

Bringing verification and guarantees in network through software. We explore ways to provide verification and guarantee in the network through software. Kinetic not only enables operators to write network policies in a concise and intuitive way, but also provides verification capabilities that can be used to verify whether the created software program correctly encodes the event and reaction control logic using model checking in temporal logic. Coronet uses graph theory to create a set of primary and backup paths that guarantee the network to survive any single data-plane failure in the network, and uses linear programming to guarantee to find the optimal traffic failover placement on backup paths to minimize maximum load on links.

1.4 Outline

Chapter 3 presents our analytical study on five years of network configuration files from two big campus networks. Chapter 4 presents Kinetic, a domain specific language and controller platform for programming verifiable dynamic network policies. Chapter 5 presents

Coronet, a failure recovery service module for SDN control applications. Chapter 7 concludes this dissertation with summary of contributions, lessons learned, and future work.

1.5 Bibliographic notes

The analysis study of campus network configuration appeared as a full paper at IMC 2014 [46]. Kinetic was published as a full paper in NSDI 2015 [47]. A poster version of Coronet appeared in ICNP 2012 [48], and is currently under submission in SOSR 2015.

CHAPTER II

BACKGROUND AND RELATED WORK

2.1 Network configuration in legacy switches

In traditional network, network operators type in a combination of low-level commands into a configuration file that resides in individual network devices such as routers, firewalls, and switches. Figure 1 shows an example of how a configuration files looks like. It is extremely hard to understand and interpret the meaning of such list of commands without appropriate knowledge, and remains challenging even for network operators who normally obtain certifications to understand and implement such configurations.

Operators often use tools to make configuration easier to manage and also to *rollback* to previous configuration when an error is detected. RANCID [89] is a widely used tracking and monitoring tool for network device configurations. RANCID creates a catalog of devices, pull the current configurations, and store them in a designated storage system. In addition to configuration files, RANCID also pulls hardware information (*e.g.*, model number, serial number). When combined with a version control system, such as CVS, it is possible to keep track how configurations in network devices have changed over time. The version control system also allows operators to manually check in changes they made in network devices, in addition to what RANCID provides automatically. The changelogs are stored in a Revision Control System (RCS) file format, which stores "diffs" between consecutive versions of a configuration file, and it is possible to reconstruct the snapshot of each revision.

Static and dynamic analysis on network configuration. Static analysis of network configuration focuses on capturing "snapshots" of configuration files from network devices and analyzing them to uncover otherwise hidden characteristics. Previous work has performed

```
...
config-register 0x2102
upgrade fpd auto
version 12.2
no service pad
service tcp-keepalives-in
service tcp-keepalives-out
service timestamps log datetime localtime
!
hostname Rich2-rtr no ip bootp server
!
ip vrf gtsav-door
 description Rich2 VRF for door controller networks
 rd 2637:999
 route-target export 2637:999
...
```

Figure 1: Example of content of configuration file

static analysis of network configurations to study network properties and to help network operators diagnose misconfigurations in their networks. Caldwell *et al.* propose a system to automatically discover configuration templates and rules [8]. Maltz *et al.* performed a study of many enterprise network configurations to gain insights about the design and use of network configuration in enterprise networks [69]. The *rcc* tool performs network-wide static analysis of router configurations to detect configuration errors in BGP [22]. There is much other previous work on identifying and detecting misconfiguration [55, 67], but these tools focus on static analysis. Other work has used static analysis of router configuration to understand how operators configure specific functions (*e.g.*, route redistribution [60] and access control [104]), as well as to identify sources of configuration complexity [6]. The NetDB system pulled router configurations from the AT&T backbone network into a database for analysis of configuration snapshots [25].

Other projects have explored the changes in configuration over time, albeit for smaller sets of network configuration, or over shorter periods of time. Sung *et al.* examined changes in stanzas over time for five enterprises VPN customers. They study which sets of stanzas change most often [98]. Chen *et al.* also examine the dependencies between configuration

stanzas by analyzing changes [11].

Automating configuration generation. Gottlieb *et al.* used NetDB to perform static analysis of network configuration to discover and construct configuration templates to help network operators automatically configure new sessions for ISP customers [31]; a follow-up system, Presto, constructs network configurations based on configuration templates [20]. These systems rely on operators to manually specify network configuration templates; ultimately, the type of analysis we perform in this study might help automatically identify common tasks and configuration idioms that could be automated. Le *et al.* [59] aim to automatically generate rules to determine commands that a given stanza should contain. This approach requires pre-processing and domain knowledge to determine how to represent the commands as attributes of stanza; it also does not explore configuration changes over time.

Mining software repositories. The field of software repository mining, a sub-area of software engineering, mines version control systems to discover artifacts that are produced and archived during the evolution of a software program. Several studies have explored the longitudinal evolution of software. Lehman *et al.* studied the evolution of source code in several IBM products between 1969 and 2001 [61]. Eick *et al.* studied the phenomenon of code decay, whereby changes to a software system become increasingly difficult over its lifetime [18]. Those studying changes to software repositories focus on both changes to high-level *properties*, such as software complexity or maintainability; and on changes to *artifacts*, which is typically done by analyzing differences between versions in the version control system itself. A specific approach to mining software repositories called *source code differencing*, pioneered by Raghavan *et al.*, examines difference data in software repositories to look for additions, deletions, and modifications to source code over time [88]. This approach is similar to that which we take in this paper, where we explore additions, deletions, and modifications to network configuration files to understand changes to software artifacts. Kagdi *et al.* provide an excellent survey and taxonomy of the

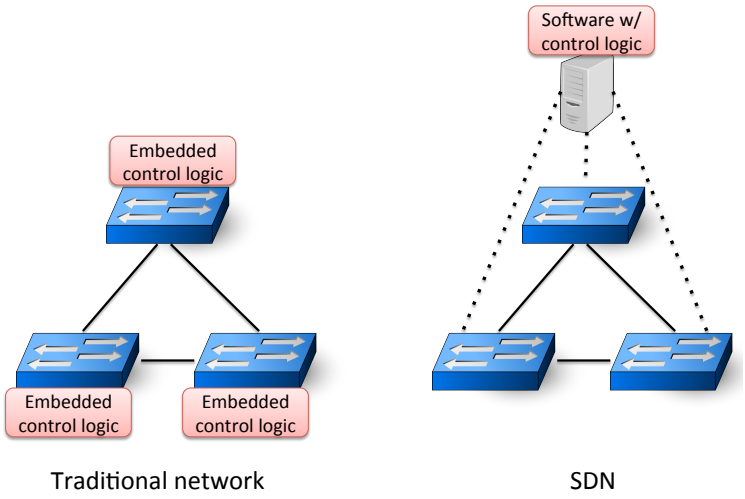


Figure 2: Traditional network vs. SDN

large body of work in mining software repositories [41].

2.2 *Software-defined networking*

Software-defined networking (SDN) decouples the system that makes the traffic forwarding decision (control plane) from the system that actually forwards the traffic (data plane). Switches in the data plane become "dumb" forwarding devices, while the "brain" and control logic resides in the logically centralized controller as a software program. Figure 2 describes this separation of control plane from the data plane, compared to the traditional network. Such decoupling enables to bring innovation easier than traditional switching mechanisms that has every logic built into the hardware. By modifying the control logic in the control plane, it is easier to make changes in forwarding behavior with much more flexibly. SDN can also simplify networking and management by having a logically centralized controller that makes forwarding decisions.

Although SDN has gain significant attention and traction only recently, there has been previous work and study that steadily lead to SDN, advocating a *programmable network*. We omit the detailed description of SDN and its history in this dissertation as there are

abundant previous studies that summarize this. Feamster *et al.* presents an intellectual history of programmable networks with technology push and use pull of techniques and protocols along the way, as well as myths and misconceptions [23]. Nunes *et al.* gives summary of past, present, and future of programmable networks [80], and Kreutz *et al.* gives a comprehensive survey on OpenFlow and SDN [54].

2.2.1 SDN, OpenFlow, and northbound and southbound API

It is important to note the difference between SDN and OpenFlow [71], and be aware of the concept of northbound and southbound API in SDN. SDN is a networking concept where the control plane is separated from the data plane, and SDN uses a well-defined interface, or Application Programming Interface (API) to interact with data plane, which is called the southbound API, as shown in Figure 3. OpenFlow is just one of the most prominent example of such southbound APIs, which defines and specifies on how to interact with forwarding devices (*e.g.*, routers, switches) in the data plane. On the other hand, northbound API is the interface between the control platform and SDN control applications. Northbound API bridges the control applications to control platform, and commands from the control application will be passed to to the data plane through the southbound API.

OpenFlow dominates the southbound API field, and new specifications are discussed and standardized by Open Networking Foundation (ONF) [81]. While there are interesting research and works in the southbound API, this dissertation focuses on the northbound API and control platforms, where research in finding the right high-level abstraction is on the way.

2.2.2 Related work in SDN

Language and abstractions. FlowLog [78] provides a database-like programming model that unifies the control-plane logic with data-plane state and controller state. Aspects of FlowLog programs can be verified, but it cannot verify arbitrary temporal relationships,

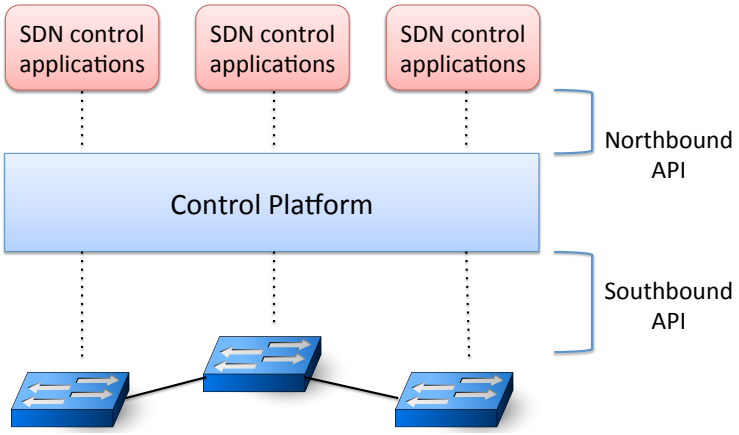


Figure 3: Northbound and southbound API in SDN

such as those that can be verified with CTL in Kinetic. FlowLog uses Alloy to perform bounded verification, so its analysis is not complete, and certain aspects of verification are manual. FlowLog has not been evaluated for realistic network policies or for large networks. It requires storing multiple database entries for each network state variable and handles certain aspects of control logic by sending data packets to the controller, so it is unlikely to scale. VeriCon [4] verifies that a program written in its language (CSDN) is correct for all topology and packet events (*e.g.*, packet arrivals, switch joins). It does not handle arbitrary network events, and there is no OpenFlow-based implementation, so its practicality is unclear.

Recent work in network verification has focused on verifying static properties of the data-plane state [45, 91, 95]. Especially, consistent updates to the data plane is extremely important for correct behavior [91]. Kinetic should leverage such system to ensure correct network behavior in face of frequent data-plane reconfiguration. Anteater [68] and HSA [43, 44] can verify properties of a static snapshot of a network's data-plane state. These systems can determine whether a static snapshot of data-plane state violates some invariant, but they do not verify the logic of the control program that generated the state in

the first place, making it difficult to identify which aspect of the network's control-plane logic caused the incorrect data-plane state. NICE [9] can test control-plane properties that might result from arbitrary sequences of standard OpenFlow events; it is not a controller, but rather a test harness for control programs written in existing low-level controllers (*e.g.*, NOX) and hence does not permit reasoning about arbitrary events.

Many languages raise the level of abstraction for writing network control programs, yet these languages do not offer constructs for concisely encoding policies that capture network dynamics, nor do they incorporate formal verification of control-plane behavior. FML [33] allows network operators to write and maintain policies in a declarative style. Nettle [100] is a domain specific language in Haskell. Procera [101] applies functional reactive programming to help operators express policies. Frenetic [27] is a family of languages that share fundamental constructs and techniques for efficient compilation to OpenFlow switches.

Fault tolerance. OpenVirtex [3] "network hypervisor" presents virtual topologies to SDN controllers and applications while assuming responsibility for ensuring physical connectivity when network components fail, but lacks efficient mechanisms for fast failover and congestion avoidance, and performs only shortest path route planning. FatTire [90] proposes an SDN policy language that allows operators to specify flow-specific resilience requirements, among other properties. Coronet could be customized to provide an efficient, scalable implementation of such policies. Keep Forwarding [105] and Data Driven Connectivity [65] propose new data-plane protocols for extremely fast restoration of data-plane connectivity after failures. While these protocols could be used with an SDN control plane that optimizes routes over longer time scales, they require switch hardware modification, unlike Coronet which works with unmodified OpenFlow switches.

SWAN [34] and B4 [40] use SDN to manage traffic routing for expensive WAN links. zUpdate [64] produces schedules for congestion-free re-routing of traffic flows for planned network maintenance events. Forward Fault Correction (FFC) [63] also uses an LP solver

after unplanned failures, but reserves spare bandwidth in advance of unplanned failures to avoid congestion. R3 [102] provides congestion-free recovery in IP networks using legacy OSPF routing and modified MPLS. In contrast to previous work, Coronet provides best-effort congestion avoidance with weighted fast failover to multiple backup routes, and more suited to SDN environment and amenable to providing transparent failover service to SDN-based control applications.

Finally, prior research has proposed improvements for controller consistency and reliability [53], and reliable switch-controller communication [5]. Aditya *et al.* suggest improvements to mitigate unreliability of switch-controller and controller-controller communication [1].

CHAPTER III

THE EVOLUTION OF NETWORK CONFIGURATION

3.1 Introduction

The behavior of a communications network depends in part on the configuration of thousands of constituent network devices, each of which is configured independently. In this sense, network configuration is a large, distributed program. Despite its importance in dictating the overall behavior of the network, we understand very little about the nature of configuration. Today, network operators implement high-level network tasks with low-level configuration commands; operators frequently make mistakes when making changes to network configuration [22, 67]. Although some studies have examined properties of configuration "snapshots" (*e.g.*, [69]), we have little understanding of how network configuration *evolves* over time.

Studying the evolution of network configuration over time can offer unique insights that cannot be learned from a single static snapshot. First, such a study can shed light on how network functions evolve over time: for example, we can learn more about how network configuration evolves, and which network tasks contribute to the growth in complexity. Second, because configuration changes are the cause of many errors, understanding the nature of how configuration changes over time—and what tasks configuration changes are associated with—can better inform configuration testing by helping create targeted test cases. Third, knowledge about configuration changes also offers valuable information about the parts of network configuration where operators spend time, which may help designers of configuration languages and configuration management systems design a better environment to make these tasks easier.

Towards these goals, this paper presents the first long-term longitudinal study of the

evolution of network configuration, for two large campus networks: Georgia Tech and the University of Wisconsin. Comparing the evolution of network configuration across two different large campuses allows us to identify trends that are common across campuses, as well as practices that are specific to a particular network. We study the changes that operators make to the configurations of all routers, switches, and firewalls in these networks over a five-year period. We study the following questions:

- How does network configuration size evolve over time? How many of the changes are additions, modifications, or deletions? Do changes tend to occur at a specific time of day or day of the week?
- Which network-wide factors contribute most to configuration evolution?
- What parts of the configuration change most frequently, and why?
- Are there dependencies and correlations in changes to network configuration, and why?

We perform our analysis on two campus networks. Although the results may not generalize beyond these two campuses, they represent a case study and a first attempt to perform extensive longitudinal analysis on campus networks. Our study also complements previous studies, which have focused on enterprise and backbone networks [25, 98].

We find that configurations continue to grow over time, and a variety of factors—infrequent ones, such as, infrastructure expansion and policy changes, and more frequent ones, such as customer addition/modification—contribute in interesting ways to configuration evolution. We find that routers experience configuration changes more frequently, and that changes to routers involve a broader range of network configuration tasks than the changes that operators make to switch and firewall configurations. Given that there are far fewer routers than firewalls or switches in both networks, the relative frequency of configuration changes to routers highlights that many crucial day-to-day network operations tasks

center around router configurations. Third, although there are many similarities in the configurations between the two campuses—both in the amount of configuration devoted to each configuration task and to the nature of the configuration changes—the network operators for each campus do have distinct practices (*e.g.*, the use of static ARP entries) that tend to appear on more general purpose devices, such as core routers. In contrast, devices that are more special-purpose such as firewalls tend to exhibit more common configuration change patterns. Finally, we note that configuration changes to switches and routers are sometimes correlated, suggesting the possibility for better configuration tools that can assist operators by automatically recognizing these dependencies.

Our findings suggest possible areas for improvement in configuration management and testing. For example, our findings concerning correlated changes across configurations can suggest improvements to the configuration process: a configuration management system could observe correlations in configuration changes and suggest changes that operators might make to low-level configuration constructs based on observations of past correlated changes for different high-level tasks. The management system can similarly help with debugging erroneous changes to configuration. Our observations regarding the high frequency of router configuration updates—which are often manual and hence error-prone—suggest the need for automated network-wide configuration management systems targeted specifically toward routers.

The rest of the paper is organized as follows. Section 4.2 presents background on network configuration, and on the configuration management systems used to make changes to network configuration. Section 3.3 describes the datasets that we use for this study and our approach. Section 3.4 describes our results for routers, switches, and firewalls. In each case, we observe characteristics for a snapshot of the network configuration; we also perform a longitudinal analysis of configuration changes over the five years of our study. Section 3.5 discusses possible future research directions for our results, including suggestions for how they might be used to improve configuration management and testing.

Section 5.6 concludes.

3.2 Background and Related Work

We use an archived history of network configuration files to analyze the nature and causes of configuration changes to switches, routers, and firewalls in the Georgia Tech and University of Wisconsin campus networks. In this section, we describe the common configuration management systems used by the networks studied.

3.2.1 Configuration Management

Both campus networks use a configuration management tool called RANCID [89] for tracking and monitoring network device configurations. RANCID's main function is to construct a catalog of devices, pull the current configurations, and store them in a version control system, which can allow network operators to track changes in the configurations to network devices. In addition to device configurations, RANCID also pulls hardware information (*e.g.*, cards, serial numbers). RANCID operates with devices from many vendors, including Cisco, Juniper, Foundry, and HP. More details can be found on the RANCID Web site [89].

In addition to accepting input from RANCID, the version control system also allows network operators to manually checkpoint and commit changes to the repository. Just as programmers use revision control to track changes to source code, network operators have adopted revision control to manage network devices by tracking changes to device configuration files. The CVS repository tracks changes by giving each revision of the configuration file a unique revision number and by storing metadata along with this revision, such as the time of change, comment, author of the change, and the sets of lines changed.

CVS stores all the metadata along with revisions for each configuration file in a single Revision Control System (RCS) file. Figure 4 shows an excerpt from an RCS file. The current revision number is on the top (line 2), followed by *log* (lines 3–5), which contain comments provided by the operator or RANCID explaining the nature and cause of the

```
1  ...
2  1.51
3  log
4  @Fri Feb  5 15:04:28 EST 2010
5  @
6  text
7  @a141 1
8   port-object range bootps bootpc
9  a160 4
10 object-group service 12-123-12-13-any-udp udp
11  port-object range bootps bootpc
12 object-group service 12-123-12-14-any-udp udp
13  port-object range bootps bootpc
14 d173 16
15 a188 9
16 object-group service 13-14-15-16-any-udp udp
17  port-object range bootps bootpc
18 object-group service 14-15-16-17-any-udp udp
19 ...
20 @
```

Figure 4: RCS file showing changes to network configuration.

change (the revision date in this case). The *text* (lines 6–20) lines document the location, number and nature of the change; these lines allow us to track additions and deletions to the network configuration. We interpret an addition immediately followed by a deletion as a modification if the sum of the line number and the number of line changes of deletion is equal to the line number of addition added by one. For example, in Figure 4, lines 14–15 shows a modification (173+16 = 188+1).

3.2.2 Related Work

We provide a brief overview of related work. We first discuss various prior work on static analysis of network configuration to better understand network properties and to characterize or diagnose network configurations. Although the networking community has seen only limited work on analyzing configuration *changes*, the software engineering community has performed extensive studies of changes to source code that relate to our study.

Static analysis. Previous work has performed static analysis of network configurations to study network properties and to help network operators diagnose misconfigurations in their networks. Caldwell *et al.* propose a system to automatically discover configuration

templates and rules [8]. Maltz *et al.* performed a study of many enterprise network configurations to gain insights about the design and use of network configuration in enterprise networks [69]. The *rcc* tool performs network-wide static analysis of router configurations to detect configuration errors in BGP [22]. There is much other previous work on identifying and detecting misconfiguration [55, 67], but these tools focus on static analysis. Other work has used static analysis of router configuration to understand how operators configure specific functions (*e.g.*, route redistribution [60] and access control [104]), as well as to identify sources of configuration complexity [6]. The NetDB system pulled router configurations from the AT&T backbone network into a database for analysis of configuration snapshots [25]. Gottlieb *et al.* used NetDB to perform static analysis of network configuration to discover and construct configuration templates to help network operators automatically configure new sessions for ISP customers [31]; a follow-up system, Presto, constructs network configurations based on configuration templates [20]. These systems rely on operators to manually specify network configuration templates; ultimately, the type of analysis we perform in this study might help automatically identify common tasks and configuration idioms that could be automated.

Dynamic analysis. Other projects have explored the changes in configuration over time, albeit for smaller sets of network configuration, or over shorter periods of time. Sung *et al.* examined changes in stanzas over time for five enterprises VPN customers. They study which sets of stanzas change most often [98]. Chen *et al.* also examine the dependencies between configuration stanzas by analyzing changes [11]. Le *et al.* [59] aim to automatically generate rules to determine commands that a given stanza should contain. This approach requires pre-processing and domain knowledge to determine how to represent the commands as attributes of stanza; it also does not explore configuration changes

over time.

Mining software repositories. The field of software repository mining, a sub-area of software engineering, mines version control systems to discover artifacts that are produced and archived during the evolution of a software program. Several studies have explored the longitudinal evolution of software. Lehman *et al.* studied the evolution of source code in several IBM products between 1969 and 2001 [61]. Eick *et al.* studied the phenomenon of code decay, whereby changes to a software system become increasingly difficult over its lifetime [18]. Those studying changes to software repositories focus on both changes to high-level *properties*, such as software complexity or maintainability; and on changes to *artifacts*, which is typically done by analyzing differences between versions in the version control system itself. A specific approach to mining software repositories called *source code differencing*, pioneered by Raghavan *et al.*, examines difference data in software repositories to look for additions, deletions, and modifications to source code over time [88]. This approach is similar to that which we take in this paper, where we explore additions, deletions, and modifications to network configuration files to understand changes to software artifacts. Kagdi *et al.* provide an excellent survey and taxonomy of the large body of work in mining software repositories [41].

3.3 Data and Analysis

We describe the configuration data that we use in our study and our approach for postprocessing and analyzing the data.

3.3.1 Data

Both Georgia Tech (*GT*) and University of Wisconsin (*UW*) manage their network configurations with RANCID and CVS, the tools described in the previous section. We have collected five years of archived configuration files from all network devices (*i.e.*, routers, firewalls, and switches) in the two networks. Table 1 shows the number of network devices

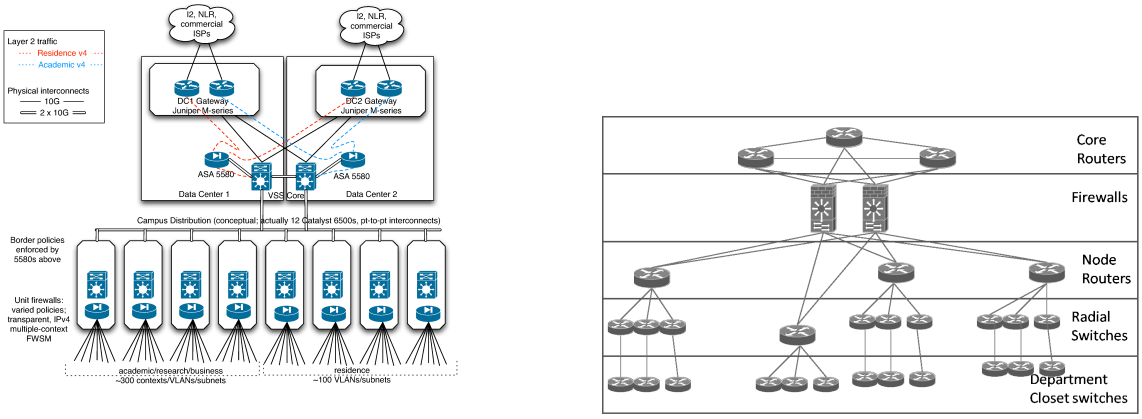
Table 1: Number of devices for each device type.

	Routers	*Firewalls*	*Switches*	*Total*
Georgia Tech	16	365	716	1097
Wisconsin	53	325	1246	1624

currently deployed in both networks for each device type. We note that both networks comprise mostly Cisco devices; configurations are stored in the RCS format. Our analysis tools thus focus on parsing Cisco's configuration language and the RCS format. The border routers in both networks are Juniper routers. Because the configuration language syntax differs significantly from Cisco's configuration language, we omit the border routers from our analysis for both networks. We now describe the design of each network and how they use RANCID and CVS in more detail.

Georgia Tech. Figure 5a shows the topology of the Georgia Tech network. The Internet connection originates from two multi-homed border routers; two physical firewalls are deployed close to the border. Departments, research groups, and residence halls are divided into separate subnets and assigned unique virtual LANs (VLANs). Each subnet has different access control policies, enforced with a separate virtual firewall. The network has hundreds of virtual firewall instances that are not shown in the figure. Hosts are connected to the network via switches at the edge. RANCID pulls the latest configuration files every three hours and saves the snapshot of the files. A CVS "commit" only occurs when the system detects a change in the latest configuration. In this network, the operators rarely make manual backups of the configurations; as such, most revisions in the repository are created by RANCID.

University of Wisconsin. The *UW* network, shown in Figure 5b, is similar to *GT* in many aspects; like the *GT* network, the *UW* network topology is hierarchical, with Internet traffic entering through two border routers and redirected through two major border firewalls for access control. Border routers are distinct from core routers, and they reside one level



(a) Georgia Tech (b) University of Wisconsin

Figure 5: GT and UW campus networks.

higher than the core routers. Figure 5b does not show the border routers. The *UW* network contains four tiers: the core and node layers, which primarily comprises routers; and the radial and access layers, primarily comprising switches. The access switches connect computers in the different departments to the network. Unlike *GT*, the RANCID setup at *UW* pulls snapshots for only firewall configurations. Similarly, commits to firewall configuration files only happen when the system detects a change to the latest configuration. Operational policies and tools at *UW* force operators to manually snapshot and commit all other configuration changes before they can be pushed to the devices. As such, most revisions are created by operators. This different approach does not affect our results, as it simply reflects a difference in the mechanism on who pulls snapshots, detects changes and commits to the repository.

3.3.2 Analysis

We analyze the evolution of network configuration changes by analyzing the RCS files for changes and extracting information about how each configuration file changes over time. This section describes our analysis, which has two major parts: (1) abstracting higher-level operator tasks from low-level configuration changes; (2) performing change analysis on the RCS files.

3.3.2.1 Developing abstractions for functionality

The first task in analyzing the network configuration is to derive abstractions of high-level functions and tasks from low-level configuration commands. This helps us more readily identify the types of tasks that network operators most commonly perform when making changes to the network configuration and the amount of configuration devoted to each task. We do so by manually associating each command in the configuration language with a unique function. Table 2 shows the list of the functions we use in our analysis, with several example commands for each of the categories. Given this categorization, we construct a database that maps configuration commands to one of these functions. Due to the abstraction we apply to the configurations, we do lose fine-grain information that can be valuable to our study, but the goal of this work is to offer a high-level analysis of network configuration and its evolution.

Although our categorization is extensive, the mapping does not cover all configuration commands: some commands are difficult to classify or do not map to any particular function (*e.g.*, commit checksum values, comments, banners, start/end markers, and version info). Although not complete, we believe our functionality map is extensive enough to cover 93% of all configuration commands and 91% of all configuration changes in University of Wisconsin, and 98% of all commands and changes in Georgia Tech, thus complete enough to shed light on how operators devote time to various high-level tasks.

To abstract configuration changes that involve multiple lines of configuration, we group configuration into *stanzas*, each of which is the largest contiguous block of commands that encapsulate a piece of the network functionality. Cisco's configuration language provides special symbols ("!") for separating stanzas. We analyze configuration changes according to how stanzas change in total.

Table 2: Functionality map. We map each command to a specific functionality to effectively analyze configuration files.

	Meaning	Examples
mgt	device management settings	username, password, telnet, ssh, logging, aaa, clock, console, radius-server
l1	interface/port settings	interface definition, bandwidth, switchport, description, duplex
l2	layer 2 settings	arp x.x.x.x, mac-address, ip proxy-arp, arp timeout, spanning-tree
vlan	VLAN settings	vlan definition, switchport mode trunk, switchport mode access vlan, set vtp, set vmps
l3	layer 3 related	ip address x.x.x.x , ip gateway x.x.x.x, ip route x.x.x.x, nat, router bgp/ospf/rip, router-id, neighbor
acl	access control	object define, access-list, permit, deny
sec	security related	vpn, ipsec, crypto, webvpn, anyconnect, ssl, tunnel-group, flood-guard
cflt	control filtering	prefix-list
qos	QoS	policy-map, class-map, service-policy, port-channel load-balance, set qos

3.3.2.2 Change analysis

After developing abstractions for the configuration tasks and grouping configuration commands by stanza, we perform three types of analysis on the configuration changes to better understand how configuration evolves over time:

- *Snapshot and change analysis.* As a baseline for understanding the composition of network configuration in the two networks, we analyze a snapshot of the network configuration for each network and examine the number of lines of configuration devoted to each function. To understand where operators devote their efforts when modifying network configuration, we then study the extent to which network configuration changes are associated with different functions that the configuration performs. We also explore how many of the changes corresponded to additions, deletions, and modifications. We use a Python library called *difflib* to infer what constitutes an addition, deletion, or modification, by comparing two consecutive revisions and producing a "diff" in a context format. We evaluate the size of the diff in terms of

number of lines. We assume the library is able to correctly identify additions, deletions, and modifications. We also correlate the line number difference from the first version to the last revision to verify that the numbers of line additions, deletions, and modifications derived are consistent.

- *Longitudinal analysis.* To understand how the network configuration has evolved over time, and the primary factors underlying different evolution patterns, we perform a longitudinal analysis of configuration changes, using five years of RCS files from each network.
- *Correlation analysis.* To understand whether certain aspects of the network configuration frequently change together, we perform a correlation analysis to determine network configuration tasks that operators commonly perform together.

Our toolkit parses each RCS file and aggregates information for the snapshot and basic change analysis, such as how many lines of configuration are devoted to each function, how many changes occur and what types of changes (*i.e.*, addition, deletion, modification) occur, when and where these changes happen. It also parses the data and author information from the RCS files determine the frequency of changes made on certain days, times, and by certain authors. The toolkit then inserts this information into a database that performs the longitudinal and stanza analysis. Correlation analysis requires additional software functionality, beyond gathering first-order statistics. To perform this analysis, the toolkit parses the RCS files and extracts every version of the configuration file. The configuration files are parsed, and stanzas from each configuration file are stored in a database, using which we can study correlated changes.

3.4 The Evolution of Configuration

We now present results on network configuration evolution in the two campus networks. In the process, we identify change characteristics that the campuses share in common and those where they differ. We augment our analysis with operator discussions to shed light on

Table 3: Number of add/del/mod for all devices and average per device for each device type in *GT*.

Georgia Tech	*add*	*del*	*mod*	Total
Routers (16)	31,178	27,064	262,216	326,458
Firewalls (365)	249,595	118,571	171,005	539,171
Switches (716)	216,958	20,185	116,277	353,420
Rtr avg. per device	2,324	1,692	16,389	20,404
FW avg. per device	684	325	469	1,477
Swt avg. per device	303	28	162	494

Table 4: Number of add/del/mod for all devices and average per device for each device type in *UW*.

UW-Madison	*add*	*del*	*mod*	Total
Routers (53)	79,202	38,288	154,407	271,897
Firewalls (325)	193,499	73,827	161,863	429,189
Switches (1246)	1608,512	213,910	1,768,384	3,590,806
Rtr avg. per device	1,494	722	2,913	5,130
FW avg. per device	595	227	498	1,321
Swt avg. per device	1,291	172	1,419	2,882

the impact of operational practices on configuration evolution in the respective campuses.

3.4.1 Overview

What types of changes occur? We begin by measuring the number of configuration commands changed for each device type and normalize the results by the number of devices, as shown in Tables 3 and 4.

We make the following high-level observations. First, the networks have a similar number of firewalls, but *UW* has nearly four times as many routers and almost twice as many switches. Second, although most changes happen in firewalls and switches (with changes on switches dominating), routers, which are far fewer in number, tend to have many more configuration changes per device (on average) in both the *GT* and *UW* networks. We delve into router configuration changes in Section 3.4.2. Third, we find interesting differences between the two networks: in *UW*, switches clearly undergo a large number of modifications relative both to other devices in the same network, and to switches in the

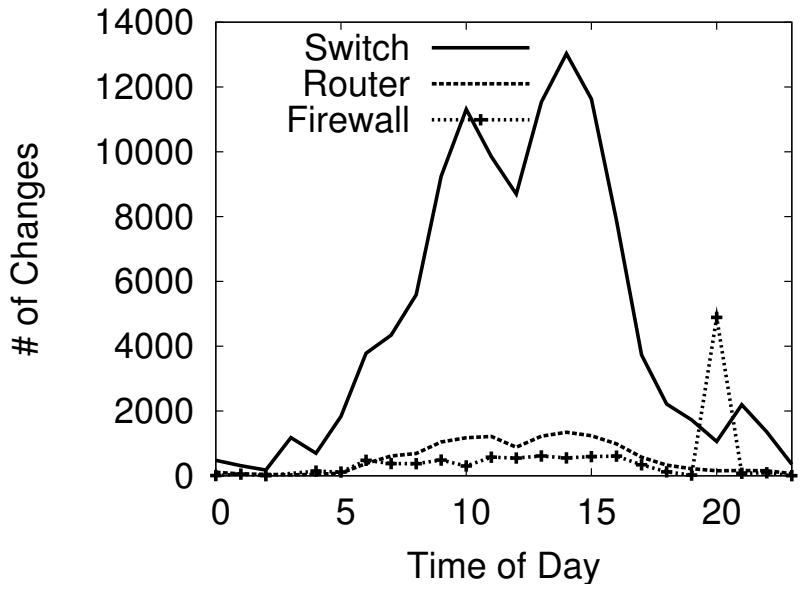
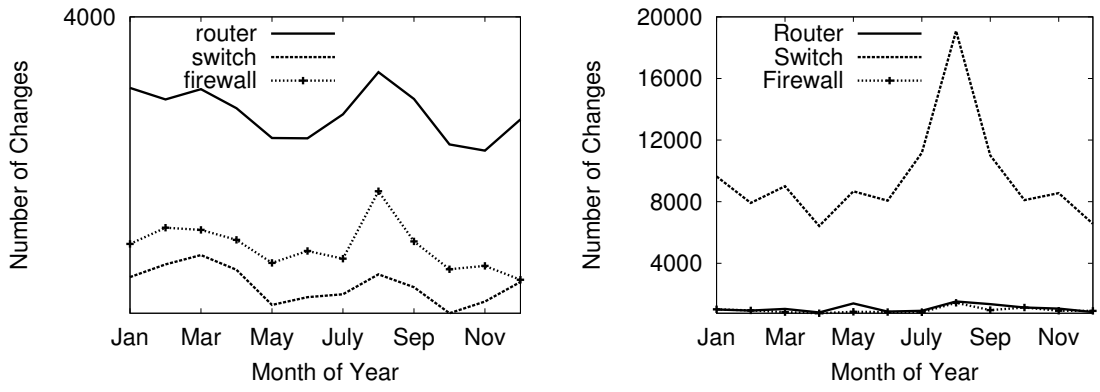


Figure 6: Changes to the *UW* network configuration over a day.

GT network. *GT* has relatively more changes to its firewall devices than *UW*. We delve into firewall and switch configuration evolutions across the two campuses in Sections 3.4.3 and 3.4.4, respectively.

When do changes occur? Figure 6 shows how configuration changes are distributed across hours of the day for *UW*. The pattern shows that most changes to switches and routers are made during the work hours. However, many of the configuration changes to firewalls happen later, as shown by a large increase in commits around 8 p.m. In contrast to routers and switches, the firewall configurations are often managed by a department-specific system administrator and then transferred to a member of the campus IT staff, who checks them for consistency and eventually uploads the files to the firewalls and commits them toward, or after the end of, working hours. Figures 7a and 7b show how changes to network configuration are distributed across the months of the year in the two networks. We observe that the number of configuration changes peaks during August. Our conversations with operators revealed that, in both networks, large network-wide changes are put off until the summer when many of the users are away and disruptions might have the least detrimental



(a) GT (b) UW

Figure 7: Changes to the *GT* and *UW* configurations over one year. Changes to the configuration increase dramatically before the start of the academic year.

effect on users of the network.

Table 5 summarizes the main findings from our analysis of these two campus networks. We note several interesting characteristics that may ultimately help understand network configuration and its evolution better, and ultimately devise better methods or tools for device configuration and management. The rest of this section discusses these findings and others in more detail.

3.4.2 Routers

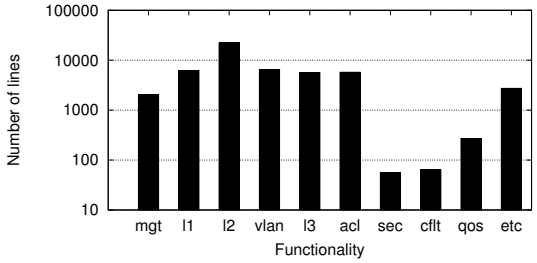
We first focus our analysis on routers. The main function of routers in the network is to support layer-3 routing, but, interestingly, many of the configuration changes on routers are associated with tasks other than layer-3 routing.

3.4.2.1 Snapshot and change analysis

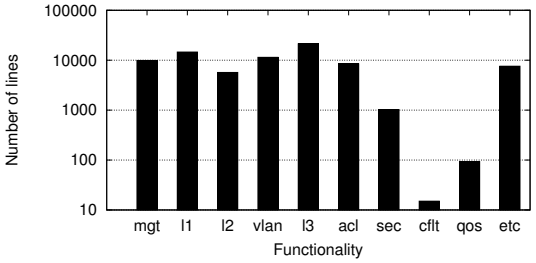
Commands other than those involving routing change frequently over time. Figures 8a and 8b present the results from analyzing a snapshot of recent configuration files from all routers from each campus. Figures 8c and 8d present the changes occurring in both networks over five years for the routers in each campus. In both campuses, the *l1* and *l3* commands are added, deleted, or modified most frequently. In the *GT* network, *l2* changes

Table 5: Main results, and where they are described.

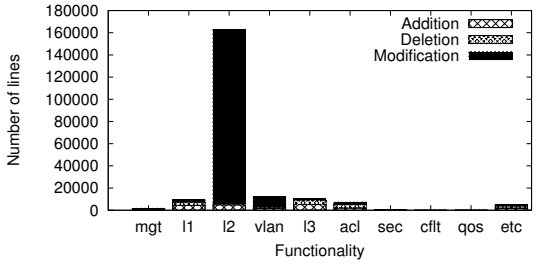
Many changes in routers are security-related. Bulk changes in routers come from security-related tasks, and different campuses use different mechanisms, thus showing dissimilar patterns of changes.	§4.2.1, §4.2.2, Figure 8, Figure 9
Deployment location matters. Where the device is placed within the network , *e.g.*, border vs edge, have significant influence on the size of change and what changes. Intuitively, devices at the border undergo a lot more changes than edge devices. In addition, the changes are more restrictive at borders and are more closely monitored.	§4.2.1: Figure 9, §4.3.1: Figure 12
Variety of network-wide tasks contribute to changes in devices. This includes network expansion, change of policy, addition of new features or modification of existing, and provisioning new customers or deleting subnetworks.	§4.2, §4.3, §4.4
Specialized devices are similar across different networks. Highly specialized network appliances, like firewalls, tend to have very similar traits when it comes to configuration and its evolution.	§4.3.1, §4.3.2
Correlated changes in firewall devices occur within the *acl* command. In firewalls, commands are highly concentrated on *acl* stanzas. However, we found highly correlated behavior within the *acl* stanza, namely object definition and actual access lists.	§4.3.3
Switch configuration is highly port-centered. Switches are configured port-by-port, and batch changes on interfaces is frequent. *l1* has a variety of subcommands, hence large correlation occur among commands in switches, although they differ by network and tasks assigned.	§4.4, §4.4.3



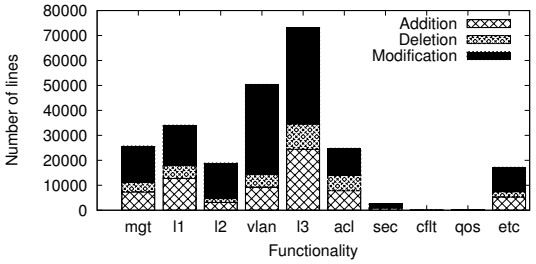
(a) Static analysis of latest snapshot (logscale) - *GT*



(b) Static analysis of latest snapshot (logscale) - *UW*

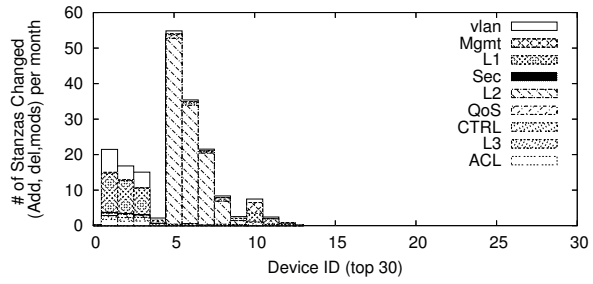


(c) Change characteristic over five years - *GT*

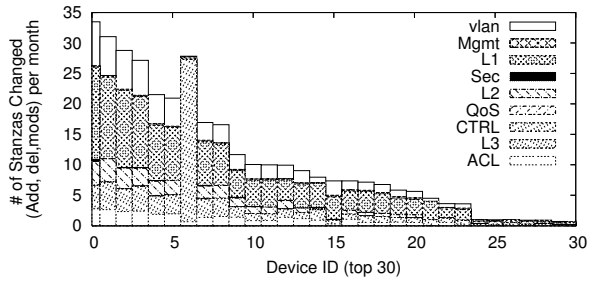


(d) Change characteristic over five years - *UW*

Figure 8: Static analysis and change analysis for routers. (a) and (b) shows a static analysis of the latest router configurations all combined for each campus. (c) and (d) shows the number of additions, deletions and modifications of each command in a row-stacked fashion, all devices combined over 5 years.



(a) *GT*



(b) *UW*

Figure 9: Number of changes to each router per month for the *GT* and *UW* networks.

are extremely frequent, due to changes to static ARP entries, which we will explain in more detail later. There are many more modifications to the *UW* network in general, many of which result from modifications to the existing *l1– l3* and *vlan* commands. Further investigation shows that VLAN interface definitions in *UW* router configurations are constantly changing due to addition, update and removal of "customers" of the *UW* network (departments or groups of users with specific needs, each group with its own VLAN). This explains the large amount of the *l1*, *l2* and *vlan* configuration changes. As we show later, the changes in *l3* are due to security related changes to a particular router.

Distinct security-related practices are found. Notably, many configuration changes to routers involve security-related tasks, although the mechanisms that each network uses differ. Each campus has security practices that use specific router configuration commands. The significant amount of the *l2* command changes in Figure 8c shows one example. Further inspection reveals that this is due to *static ARP* entries in routers deployed in residence halls in the *GT* network. The *GT* network maps each end host machine's MAC address to a static IP address in residence halls. The main goal is to provide better security and more fine-grained monitoring. This part of the configuration constantly undergoes changes as residents come and go. We examine changes to the stanzas in the routers for the different networks in Figures 9a and 9b. Our analysis shows that, on average, most routers in the *UW* network have configuration changes that reflect a consistent breakdown across functions, regardless of the router. The exception to this pattern is device 6, which has significantly more layer-3 changes than other devices. Upon further examination, we discovered that router is a *blackhole* device. Blackhole filtering, or null routing, provides a firewalling service with minimal impact on performance, by routing certain traffic to nowhere, and not providing any information regarding what happened to the traffic sources (no "no route to destination" reported back; hence the term blackhole). This practice uses "static routes" to manipulate the blackhole behavior; thus, we see constant changes to *l3* stanzas in this router device. This is another example of using specific router configuration commands to

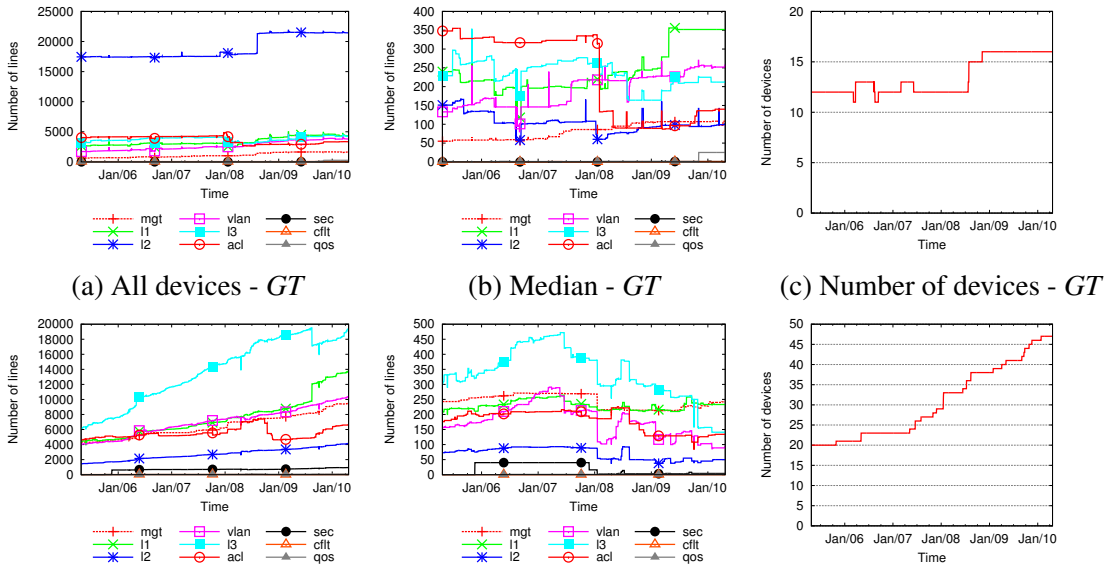
achieve certain security goals. The *GT* network also has routers with changes that are distinct from one another. Devices 5, 6, 7, and 8 are "resnet" routers in the campus dormitories with static ARPs.

Number of changes in core routers are noticeably larger than non-core routers. In Figure 9a, devices 1, 2, and 3 in the *GT* network have many *l1* and *vlan* changes, which are distinct from other routers. These three routers are core routers through which many VLANs are trunked. Core routers are responsible for routing traffic between the upper border gateway and subnets at the edge. Many VLANs for edge subnets are aggregated (or trunked) at these routers, so changes in each subnet can result in substantial changes at the core router. This characteristic is apparent in the *UW* network as well, as shown in Figure 9b. Devices aligned on the left of the figure (device 0-5) are the core routers, which are identified by their device names. The number of changes per month for these routers are significantly larger than other non-core routers. The changes are concentrated on *l1* and *vlan* stanzas.

This analysis shows that, depending on the function of the router and its placement in the network, the changes to its configuration may differ significantly. Different design and operational practices in two networks result in distinct change patterns in routers.

3.4.2.2 Longitudinal analysis

Network expansion causes steady increase in configuration size. Figures 10a and 10d show how the number of lines changes, for each router function. The figures account for all routers that were deployed in the network at a single time epoch. There is a gradual increase in overall number of lines in *UW* routers while it is quite steady for the *GT* routers. This growth appears to be caused by devices being added to the network, as shown in Figures 10c and 10f. The number of *GT* routers has grown relatively slowly over the past five years, while the *UW* network has added many routers over the time period. In 2007, there are 7 cores, 10 nodes, and 6 devices playing miscellaneous roles. By 2010, however, there are 9



(a) All devices - *GT* (b) Median - *GT* (c) Number of devices - *GT*

(d) All devices - *UW* (e) Median - *UW* (f) Number of devices - *UW*

Figure 10: Longitudinal analysis of routers over five years. The graphs show the total number of lines of configuration for all devices, the median number of lines of configuration per device, and the total number of devices over time.

cores, 12 nodes, 4 radial and hot spares, 1 VPN router, and several access routers. There is one considerable period of growth in routers in the *GT* campus network around July 2008 (Figure 10c), which caused a significant increase in *l2*, as shown in Figure 10a. This growth in routers corresponded to the expansion of the "resnet" network in *GT*.

Adding features, modifying existing commands and provisioning new customers using VLANs cause a lot of changes. We now examine the evolution of a typical router configuration. We calculate the median number of lines of each function among all routers at every time epoch. Results are shown in Figure 10b and 10e. For the *GT* network, *l1* is increasing over time, while *l3* stays relatively steady. The sudden increase of *l1* seems to be caused by new routers added to the network, again around July 2008. This is understandable, as new router devices would have a bulk of ports defined and opened when first deployed. A sudden drop in *acl* around January 2008 was caused by a major configuration clean-up: several core routers purged hundreds lines of access control lists, and disabled *AppleTalk* routing and related instances. Several subnets were removed from the network

Table 6: Correlated stanza changes for routers.

UW		GT	
Correlated Stanzas	%	Correlated stanzas	%
l1, vlan	35%	l1, vlan	26%
l1, l2, l3	13%	l1, l2	18%
l1, l3	14%	acl, l1	10%
acl, l1, l2, l3	7%	l1, l3	9 %
acl, l1	5%	l1, l3, acl	5%

at this point, causing hundreds of *acl* to be obsolete. For the *UW* network, the number of *vlan* and *l3* lines for the median device generally decreases over time as new devices are added, mainly because the newly added devices have far fewer lines with these functionalities. The exception to this is the period from from Jan 2008 to Jun 2008, where there is growth in the configuration constructs for these two functions. There are two reasons for this: first, the median device remains the same over this period; as more VLANs are added, there is a steady growth in *vlan* statements; and second, the introduction of multicast related configuration and new filters *acl* result in a sharp jump in *l3* and *acl* configuration lines.

On the whole, we find that a variety of network-wide factors contribute to configuration changes: network expansion, addition of new features or modification of existing ones, and provisioning of new customers. No one factor appears to be the dominant reason for configuration changes, although the impact of the first two factors, and the configuration changes that result from them, appear less often.

3.4.2.3 Correlation analysis

We first examine the extent to which changes to certain configuration constructs are correlated with others. To perform a certain high-level operational task, an operator has to change (add, delete, or modify) many different stanzas simultaneously. Indeed, we found that a fraction of changes—13% and 10% of all change events in *UW* and *GT*, respectively—are indeed performed in a correlated fashion, involving two or more stanza

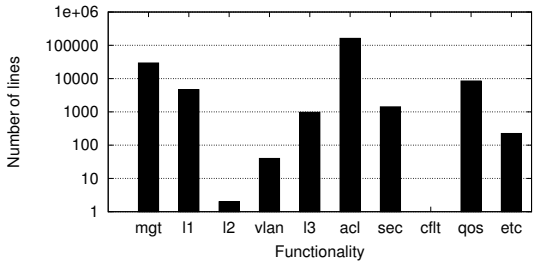
types. Such correlation has implications for the design of configuration management and debugging tools, as we discuss in Section 3.5.

Table 6 enumerates the five most common types of correlated stanza changes. The two campuses agree in three of the five: (1) *l1* and *vlan*: Although routers provide routing capabilities, they also have to control reachability (i.e., offer basic connectivity between, as well as ensure separation of, multiple subnets), which is often achieved through the use of VLANs. Thus, a considerable amount of change to *l1* and *vlan* occurs; (2) *l1* and *acl*: These change together because adding, deleting, or modifying networks (an *l1* change) often causes access control lists corresponding to the networks to be added, deleted, or modified to ensure consistency; and (3) *l1* and *l3*: These correlated changes occur because modifying interfaces and subnets in the routing stanzas permits redistribution of reachability information.

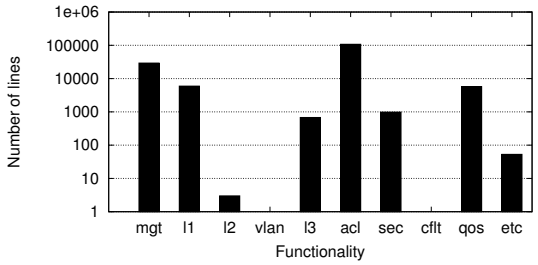
Correlated *l1* and *l2* changes (the second most common in the *GT* network) occur when defining interfaces and setting a spanning-tree protocol for that interface for which each port has a variety of options. Similarly, correlated *l1*, *l2* and *l3* changes in *UW* occur when interfaces are added, the spanning tree is modified for the interfaces, and the routing protocol is changed to include the subnet or to setup routing protocol/peering adjacency over the interface.

3.4.3 Firewalls

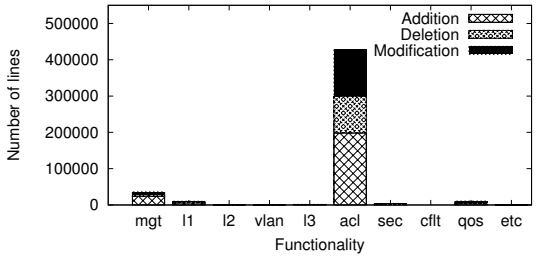
A firewall's main function is to perform access control. As expected, we find that configuration changes are concentrated on *acl* statements (Figures 11c and 11d). Therefore, we focus most of our analysis instead on how firewalls are used in the respective campuses and the similarities and differences in how they are used. We find that, despite the highly specialized nature of firewalls, their usage across the two campuses is similar (see Figures 11 and13).



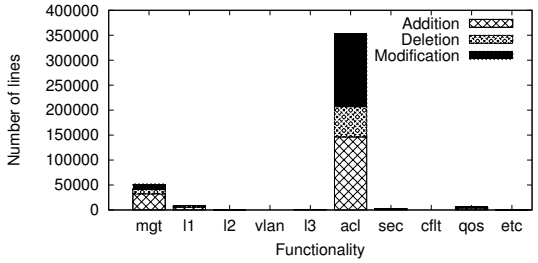
(a) Static analysis of latest snapshot (logscale) - *GT*



(b) Static analysis of latest snapshot (logscale) - *UW*

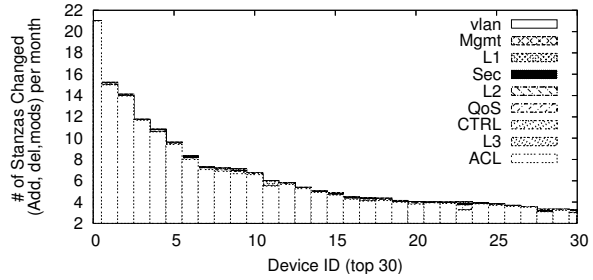


(c) Change characteristic over five years - *GT*

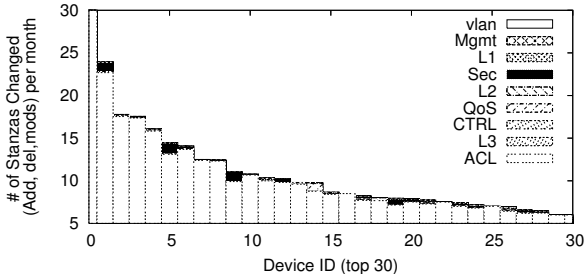


(d) Change characteristic over five years - *UW*

Figure 11: Static analysis and change analysis for firewalls. (a) and (b) shows a static analysis of the latest firewall configurations all combined for each campus. (c) and (d) shows the number of additions, deletions and modifications of each functionalities in a row-stacked fashion, all devices combined over 5 years.



(a) *GT*



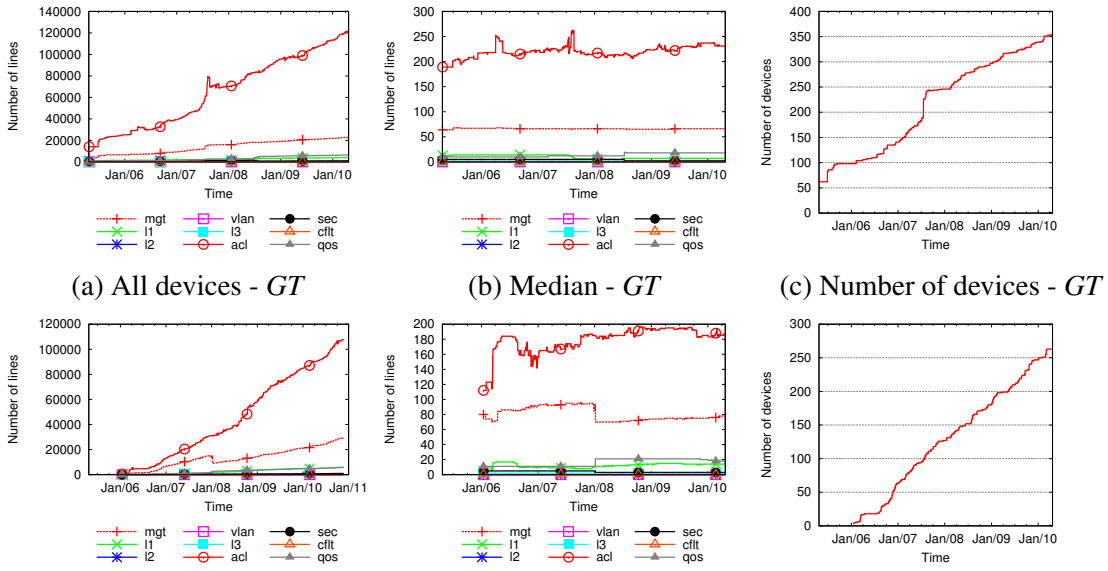
(b) *UW*

Figure 12: Number of changes to each firewall device per month for the *GT* and *UW* networks.

3.4.3.1 Snapshot and change analysis

The change characteristics are similar between two campuses. Figures 11a and 11b show the static analysis of the latest firewall configuration snapshot. As expected, most of the configuration is dedicated to access control (note that graphs are in log scale). This characteristic holds for changes as well, as shown in Figure 11c and 11d. Most changes are dedicated to access control while other types are mostly insignificant. High similarity in snapshot and change analysis between two different campus networks is intriguing. These similarities also appear in the longitudinal analysis shown in Figure 13. This similarity seems to result from the fact that devices like firewalls are designed to provide specific functionality to the network. Most changes involve *acl* for both campus networks. The actual content of commands will still differs, since network policies from two distinct networks are far from similar.

Deployment location matters. Figure 12 shows the top 30 firewall devices sorted in terms of number of stanza changes per month. Again, the graph shows that changes are heavily concentrated on *acl* stanzas. The number of changes per firewall is skewed, with a few firewalls experiencing many more changes and a large set of remaining firewalls experiencing minimal changes per month. Upon closer examination, we find that the top firewalls in the *GT* network (Figure 12a) are close to the border, and are physical devices where multiple virtual firewall instances actually spawn from. All changes to the virtual firewalls are reflected on these border firewalls, hence showing a lot of changes compared to others. This is true for the *UW* network as well. We also found a practice in the *GT* network that close-to-border firewalls are periodically backed up and commit to the CVS every six hours, while unit firewalls which are deployed closer to the edge are not managed that intensively. We assume this is due to the higher reliability requirement for the border firewalls compared to the unit firewalls; failure or misconfiguration on border firewalls can



(a) All devices - *GT* (b) Median - *GT* (c) Number of devices - *GT*

(d) All devices - *UW* (e) Median - *UW* (f) Number of devices - *UW*

Figure 13: Longitudinal analysis of firewalls over five years.

bring destructive consequences to the whole network.

Distinct practice using *sec* stanzas is captured. There are some noticeable *sec* stanzas on the *UW* network in Figure 12b. Inspection of the actual commands changed show that operators are turning off failover for the interfaces (internal and external). This prevents the virtual firewalls from failing over when the interfaces or connected devices are taken down. The anomalies occur in the firewalls used primarily by the campus IT department.

In summary, the two campus networks show very similar configuration change and usage patterns when it comes to firewalls. The majority of the changes occur at the border firewalls, where the changes on virtual firewalls are aggregated to. Border firewalls are also managed more intensively compared to simple unit firewalls at the edge.

3.4.3.2 Longitudinal analysis

Figure 13 shows the longitudinal view of firewall configuration evolution over a span of five years. The number of lines for *acl* commands increases sharply over five years. For the *GT* network, there was a factor of six increase. In the case of the *UW* network, firewalls were deployed starting in January 2006, and around 260 firewalls were added through April

2010.

Figures 13b and 13e show how the configuration of a typical firewall evolves over time. The small spike in the *GT* network around July 2007 is caused by adding new firewall machines and moving a substantial amount of access control lists to those firewalls. ACL configurations in previous firewalls are removed several days after the migration occurs. A slight increase of *qos* commands occurs around July 2008. Further inspection in the configuration shows a global policy map, which includes default inspection ports and Session Initiation Protocol (SIP) ports, is defined and inserted into around 40 firewalls. This policy map enforces further inspection on the specified services. In the *UW* network (Figure 13e), we observe a similar increase in *qos* commands around January 2008. These define two class maps: one for SIP ports as with the *GT* network, and one for a specific list of management services, ranging from DNS and rsh to SunRPC.

The decrease of *mgt* commands around January 2008 is due to the removal of the "pdm" (PIX Device Manager) commands from all firewalls. This command allows the device to determine who will have access to the management GUI for the firewalls. The removal of these commands prevents access to the GUI and forces the departments to submit all changes to the campus IT team, reflecting a new campus-wide security policy instituted in *UW* in 2008.

As with routers, we observe that a variety of network-wide factors, ranging from infrastructure expansion, to policy changes, and addition/modification of features, contribute to firewall configuration evolution.

3.4.3.3 Correlation analysis

Correlation analysis on firewall devices has not shown any specific correlated change behavior on stanzas, as most changes are solely made to *acl* stanzas. Clearly, the functionality abstraction we employ in this paper is too high-level for correlation analysis on firewalls. We do witness many correlated changes within the *acl* commands. Specifically, if there is

a change in the network object group definition (where operators can combine multiple IP addresses, ports, subnets to separate groups), there are also changes in the actual access control lists which reference that particular network object. More fine-grained classification of commands within the *acl* stanza may show interesting correlated change behavior. For example, there might be correlated changes within the access control lists; such an exploration would make a good direction for future work.

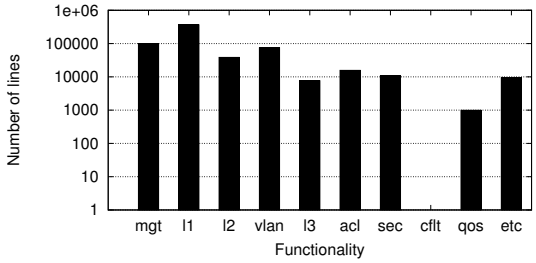
3.4.4 Switches

In this section, we study changes in switch configurations. We find: (i) interface definition and related subcommands dominate the changes in switches (Figures 14c and 14d); and (ii) extensive correlated changes (Table 7).

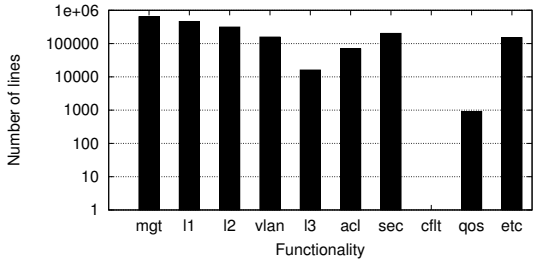
3.4.4.1 Snapshot and change analysis

Many changes involve *management, layer 1, layer 2* and *VLAN* commands. Figure 14 shows the static snapshot and change analysis of switches from both campuses. If we focus on changes of the configuration, Figures 14c and 14d show that there are many *mgt*, *l1*, *l2* and *vlan* command changes. Note from Figures 14a and 14b that these lines also constitute a majority of any given switch configuration snapshot. As seen previously, *l1* and *vlan* changes occur together naturally. *l2* changes are understandable as switches can offer layer 2 connectivity. The abnormality seems to be the significant number of changes in *mgt* commands. A majority of *mgt* changes in switch devices are in fact *snmp trap* and *logging* commands; *snmp trap* is for sending events to the SNMP server when particular conditions are met, and *logging* configures the logging behavior for the interface. The default is to define SNMP trap and logging commands along with the port configuration in the *UW* network. This induces the bulk of the *mgt* and *l1* changes to the network.

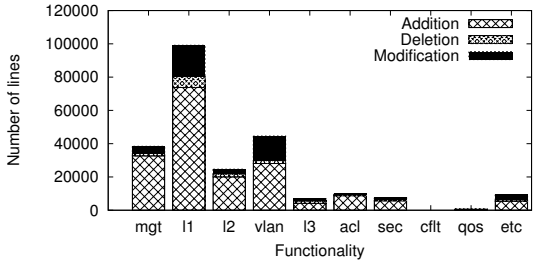
Certain practices shape how switch configuration evolves. The *GT* network has many additions, while the *UW* network experiences many modifications. Most of the changes



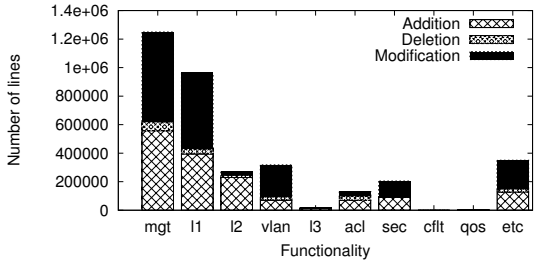
(a) Static analysis of latest snapshot (logscale) - *GT*



(b) Static analysis of latest snapshot (logscale) - *UW*

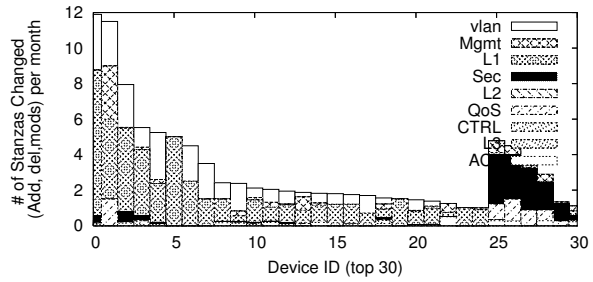


(c) Change characteristic over five years - *GT*

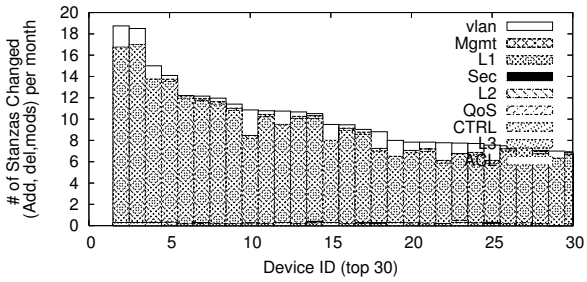


(d) Change characteristic over five years - *UW*

Figure 14: Static analysis and change analysis for switches. (a) and (b) shows a static analysis of the latest configurations all combined for each campus. (c) and (d) shows the number of additions, deletions and modifications of each functionalities in a row-stacked fashion, all devices combined over 5 years.

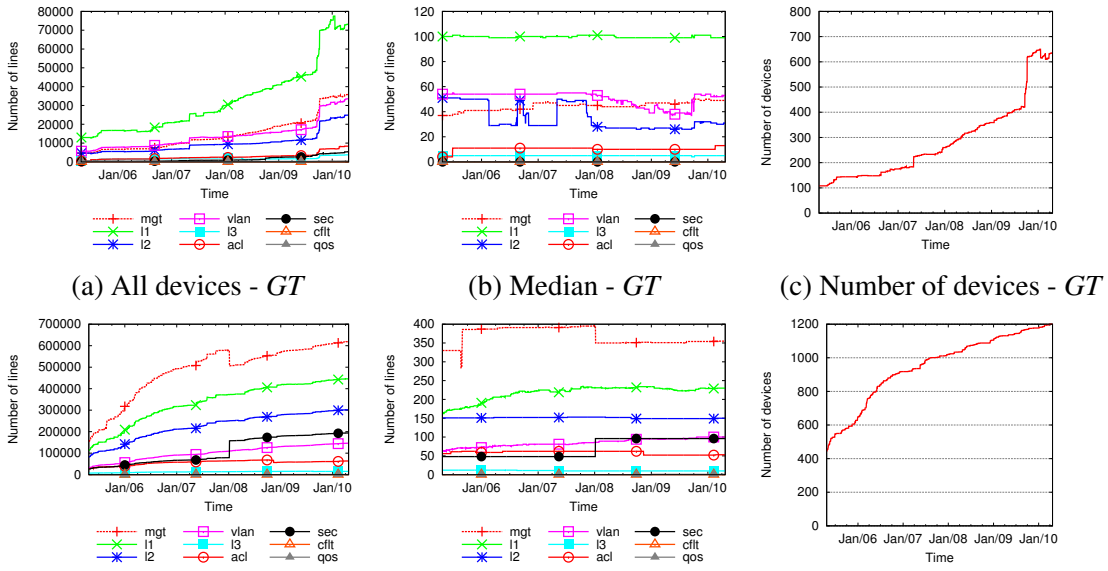


(a) *GT*



(b) *UW*

Figure 15: Number of changes to each switch per month for the *GT* and *UW* networks.



(a) All devices - *GT* (b) Median - *GT* (c) Number of devices - *GT*

(d) All devices - *UW* (e) Median - *UW* (f) Number of devices - *UW*

Figure 16: Longitudinal analysis of switches over five years.

in the *GT* network are actually due to additions in *l1* and *vlan*, followed by *mgt* and *l2* commands. However, the *UW* network looks different; there are a lot of modifications in general, with changes in *mgt* and *l1* followed by *vlan* appearing to dominate the changes. We found that this resulted from a large number of modifications that operators made to a majority of switches in the *UW* network, including interface and their subcommands, *snmp trap* and *logging*. From this behavior, it appears as though operators of the *UW* network have decided to uniformly enable or disable some functions on ports across most of its switches.

Specialized switch devices can be identified by their distinct patterns. Figure 15 presents the number of changes per stanza type, for each of the top 30 switches in both networks. The switches in *UW* are similar, in that most of the changes are in the interface commands, with a few devices having a significant amount of VLAN additions. Unlike *UW*, *GT* switches have several different patterns. *mgt* accounts for a significant amount for Device 1. This device is a switch for the SoX (Southern Crossroads) deployed within the

Georgia Tech campus, and the changes to *mgt* are due to changes to the device authentication configuration, namely aaa, radius-server, and file stanzas. This can be caused by the fact that multiple users manage this device. There is also a group of devices (devices 25–29) that have a lot of *sec* stanzas. Further inspection shows these devices are *VPN* boxes with many *ipsec* changes. These devices can be classified as specialized switches that provide unique functions of secure connection establishment between the internal and external network.

3.4.4.2 Longitudinal analysis

Evolution trend is heavily influenced by how switches are added, removed or swapped, as well as their initial configuration. Figures 16a and 16d show a gradual increase in the total number of configuration lines, for both networks. The increasing trend is mainly due to the increase in number of switch devices, as shown in Figures 16c and 16f. An abnormal spike appears around September 2010 in Figure 16a, when 150 new switches were added to the *GT* network. In Figure 16b, the *l2* command experiences several decreases and increases. Inspection of the switch configurations reveals that around February 2006, a set of new switches with 24 ports was added to the network; older switches had more ports and hence had more verbose configuration. The other increases and decreases are of the same nature: the trend depends on what type of switches are added to the network with specific initial configurations. This phenomenon is similar to the decreasing number of *vlan* commands in Figure 16b; switches with small *vlan* instances are being added to the network continuously.

Distinct practices and decisions in *UW*. Turning to *UW*, in Figures 16d and 16e, we see that in January 2008, the number of lines for *sec* doubles and *mgt* decreases. Additions to *sec* commands occurred due to changes to all interfaces on all switches to make the network more resilient to bursty traffic, by installing a limiting mechanism. The *mgt* drop in Jan, 2008 is due to operators removing the SNMP feature from all interfaces. In both cases, the

large changes reflect updates to all switches in the network.

We find that operational and network expansion practices of individual networks play an important role in shaping how the switch configuration evolves; understanding this evolution can facilitate the design of better network management tools.

3.4.4.3 Correlation analysis

In Table 7, we examine the set of correlated stanzas for the switches across the two campuses. Most changes in *UW* include either *l1* or *mgt* whereas most changes in *GT* include either *vlan* or *l1*. However, both networks agree on a set of correlated changes: (1) *vlan* and *l1*, (2) *acl* and *l1*, and (3) *mgt* and *l1*. As with other devices, interface definition and VLAN configuration commands have high correlation, as do ACLs that reference certain interfaces. *l1* and *mgt* correlates well in switches, because many default interface configuration in switches have *snmp trap* and *logging*. The *UW* network has an additional *l2* correlation with *l1* and *mgt*. This is due to heavily using spanning-tree related subcommands on interfaces. Although not shown, *UW* does in fact contain *mgt* and *vlan* correlations; *UW* contains half as many correlated occurrences of *mgt* and *vlan*. In addition to those, *GT* includes a unique combination *sec* and *qos*, this combination can be attributed to the usage of VPN devices, which are unique to *GT*. The *l1*,*l2*, and *mgt* combinations are unique to the *UW* campus; similar commands change as with the *l1* and *mgt* combination.

On the whole, we find significant evidence of correlated changes to stanzas, although the specific sets of stanzas that change depend on the network in question. As such, when using correlations to drive configuration management (*e.g.*, in providing guidance on configuration changes to install, as discussed in Section 3.5), we must first design techniques that learn the correlations prevalent in a given network.

Table 7: Correlated stanza changes for switches.

UW		GT	
Correlated Stanzas	%	Correlated Stanzas	%
acl, *l1*	24%	*mgt*, *l1*	15%
l1, *vlan*	11%	*l1*, *vlan*	13%
l1, *l2*, *mgt*	11%	*acl*, *l1*	11%
mgt, *l1*	10%	*mgt*, *vlan*	6%
vlan, *mgt*	9%	*sec*, *qos*	6%

3.5 Discussion

We now discuss how our observations can help augment existing and proposed configuration tools and mechanisms.

First, it may be possible to provide recommendations or suggestions to the operator concerning changes, based on the previous historical change logs. For example, we have found that *l1* and *l3* stanzas change the most in routers, while *l1* and *vlan* stanzas changes together frequently in switches. Based on the what the operator is changing, a system could provide feedback or possible changes associated with currently updated stanzas. For example, in firewalls, if there is a change in network object group definitions, the system can recommend the operator to also consider the ACLs where that group is used. A tool could also proactively offer *negative* recommendations. If the operator attempts to change a line or stanza of the network configuration that is not frequently changed, the configuration system could discourage the modification by raising an alert, forcing the operator to go through multiple confirmation steps, or blocking a low-rank operator from making changes.

Knowledge of network configuration changes also helps point designers of configuration automation tools to parts of the configuration that could be automatically generated, for various types of devices. Our results reveal that network operators make different types of configuration changes, depending on the network device type and, hence, point to possibilities for customizing these configuration change templates based on the type of device.

Highly specialized devices like firewalls are distinct, where most configuration is dedicated to access control, and the changes mostly happen on that specific area as well. For switches, a template that enables batch-modification on ports and VLAN configurations would be helpful.

Our work also opens many possible avenues for studying configuration changes in more detail. One possibility would be to look for changes to network configuration that were later reverted or changed again. Such changes might point to cases where operators made errors when configuring the network; knowledge about where in the configuration these errors occur could improve configuration testing methods. Another possibility would be to analyze the configuration for "copy-paste" behavior, as is done in software engineering studies of "code clones". Identifying parts of configuration that are commonly copied might also inspire design templates for configuration update.

3.6 Conclusion

We presented a longitudinal analysis of network configuration and its evolution over five years for two large campus networks. We focused on capturing configuration characteristics and common practices by examining change patterns from CVS commit log data, in contrast to earlier studies, which have mostly examined a single configuration snapshot. In contrast to earlier studies, which have focused mostly on router configuration, we also study configuration changes for firewalls and switches, which comprise a significant portion of the networks. Our study unearths a variety of important aspects regarding configuration evolution: we study the frequency and extent of configuration updates, showing the similarities and differences across networks; identify the prevalence of correlated changes; and shed light on how various network-wide factors affect evolution.

Our work is a first step toward understanding configuration evolution in more depth.

Just as mining software repositories has emerged into a mature field in software engineering for assisting programmers understand software engineering better, we believe that mining network repositories for configuration changes can help deepen our understanding of configuration and design practices. A better understanding of this process is imperative, as it is a cornerstone for improving current methods of configuring and managing network devices, easing the pain of network operators by providing recommendations, offering better templates, reducing misconfiguration and, ultimately, enabling automation.

3.7 Acknowledgements

We thank the campus network operators who provided us with network configuration data. We especially thank Dan Forsyth and Dale Carder for answering many questions about network management and configuration practices. We should also like to thank Lixia Zhang (our shepherd) and the anonymous reviewers for their insightful feedback. This work is supported in part by NSF grants CNS-0643974, CNS-0746531, CNS-1017545, CNS-1018021, and CNS-1050170.

CHAPTER IV

KINETIC: VERIFIABLE DYNAMIC CONTROL

4.1 Introduction

Network conditions are always changing. Traffic patterns change, hosts arrive and depart, topologies change, intrusions occur, and so forth. Despite the fact that many of these changes are predictable—and, in some cases, even planned—an operator's *control* over the network remains relatively static. In response to changing conditions, network operators typically manually change low-level network configurations. Our previous study of network configuration changes found that a campus network may experience anywhere from 1,000 to 18,000 changes per month [46]. Although tools like Puppet [87] and Chef [10] can automate some network device configuration tasks, this level of automation is still relatively hands-on and error-prone.

To underscore the importance of this problem, we analyzed acceptable use policies from more than 20 campus networks (many of which are publicly available [62]) and also surveyed network operators about their experience with existing tools for implementing these kinds of policies. These policies are written in English and typically express how the network's forwarding behavior should change in response to changing network conditions. For example, the University of Illinois's network use policy has an unrestricted class, and four restricted classes of traffic shaping; a user's traffic is downgraded into different classes based on their past usage over a 24-hour sliding window. Such policies sound simple enough when expressed in prose, but in fact they require complex instrumentation and "wrappers" that dynamically change low-level network configuration. Network operators currently have no concise way to express these functions, nor do they have any way of checking whether their changes will result in the intended behavior. In a recent survey we

conducted that included several hundred network operators, 89% of respondents said that they could not be certain that the changes they made to network configuration would not introduce new bugs.

Software-defined networking (SDN) is a powerful approach to managing computer networks [23] because it provides network-wide visibility of and control over a network's behavior; the Frenetic [28] family of languages provides higher-level abstractions for expressing network control. These languages are embedded in general-purpose programming languages (specifically, OCaml and Python), which makes it possible to write control programs that can respond to arbitrary events. Yet these languages do not provide intuitive abstractions for *automating* changes to network policy in response to dynamic conditions, nor do they make it possible to *verify* that these changes will match the operator's requirements for how network behavior should react to changing network conditions.

To address these problems, we present Kinetic, a domain specific language (DSL) and SDN controller that enables writing network control programs that capture responses to changing network conditions in a concise, intuitive, and verifiable language. Kinetic provides a structured language for expressing a network policy in terms of finite state machines (FSMs), which both concisely capture dynamics and are amenable to verification. States correspond to distinct forwarding behavior, and events trigger transitions between states. Kinetic's event handler listens to events and triggers transitions in policy, which in turn update the data plane.

Kinetic makes it possible to *verify* that changes to network behavior conform to a higher-level specification of correctness. For example, a network operator might want to prove that a control program would never allow a host access to certain parts of the network once an intrusion has been detected. Ongoing work has devoted much attention to verification of the network's data plane; tools such as VeriFlow [45] and HSA [44] can determine, for example, whether the forwarding table entries in a network's switches and routers would result in persistent loops or reachability problems. However, *these tools only operate on*

a snapshot of the data plane; they do not allow operators to reason about network *control programs*, or how network control would change in response to various events or changes in network conditions. They do not provide any way for a network operator to find errors in the control programs that install erroneous data-plane state in the first place. Kinetic's focus on automating and verifying the control plane is complementary to this previous work. Kinetic's use of computation tree logic (CTL) [14]—and its ability to automatically verify policies with the NuSMV model checker [12]— can allow network operators to verify the dynamic behavior of the controller before the control programs are ever run.

One significant challenge we faced when designing Kinetic is the potential for state explosion in Kinetic programs, due to the large number of hosts, flows, network events, and policies. A naïve encoding of dynamic policies in an FSM would result in an exponential number of states, even for simple programs because every flow, with all possible combination of fields (*e.g.*, src/dst IP, src/dst MAC, etc), can have its own state. To control this state explosion, Kinetic introduces an abstraction called a *Located Packet Equivalence Class (LPEC)*, through which a programmer can specify a division of the flow space and map an independent copy of an FSM (FSM instance) to each class of flow space. Using LPECs, a programmer can define groups of flows that should always map to same FSM instances (*e.g.*, all flows from the same source MAC address). Thus, each defined group of flows will be in the same state. Additionally, because Kinetic is itself based on Pyretic (a Python-based SDN control language in the Frenetic family) [75], Kinetic inherits Pyretic's language and runtime features. Specifically, Kinetic uses Pyretic's composition operators to express larger FSMs as multiple smaller ones that correspond to distinct network tasks (*e.g.*, authentication, intrusion detection, rate-limiting). Applying Pyretic's composition operators to independent Kinetic FSMs and classic product construction of automata [19] (combining multiple FSMs with union or product) greatly simplifies the construction of Kinetic's FSM expressions and allows the FSM-based policies to scale.

We evaluated two aspects of Kinetic: (1) its usability, in terms of both conciseness and

operators' facility with expressing realistic network policies; and (2) its performance, in terms of its ability to efficiently compile network policies into flow-table entries, particularly as the number of policies, the size of the network, and the rate of events grow. We conducted a user study with Kinetic of more than 650 participants, many of whom were network operators with no prior programming experience; most found Kinetic quite accessible: 79% thought that configuring the network with Kinetic was easier than current approaches, and 84% thought that Kinetic makes it easier to verify network configuration than existing alternatives.

Kinetic is open-source and publicly available; the project webpage provides access to the source code, a tutorial on Kinetic, and all of the code for the experimental evaluation [49]. The system has been used by SDN practitioners [29] and has served as the basis for projects and assignments in several university courses, as well as in a Coursera [16] course, where it has been used by thousands of students over the past two years.

4.2 Motivation and Background

Table 8: Demographics of participants in the Kinetic user study. We asked these participants about their experiences configuring existing networks, as well as their experiences using Kinetic. Section 5.4 discusses the participants' experience with Kinetic. Not all participants answered every question.

Profession		**Experience (years)**		**# Users in Network**	
Operator	216	1	32	1–10	156
Developer	251	1–5	310	10–100	137
Student	123	5–10	187	100–1,000	136
Vendor	80	10–15	150	1,000–10,000	118
Manager	69	15–20	122	$>$ 10,000	322
Other	138	$>$ 20	73		
Total	877		874		869

To motivate the need for Kinetic, we present the results of a survey of network operators about problems automating and verifying network configuration. We then present background on Pyretic, the language on which Kinetic is based; and on model checking

and computation tree logic, which we use to design Kinetic's verification engine.

4.2.1 Motivation: Network Operator Survey

To gain a better understanding of the extent to which network operators have to change their network configurations, as well as their level of confidence in their changes, we conducted an institutional review board (IRB)-approved survey of more than 800 participants, concerning their experience with configuring existing networks, as part of a Coursera course on software-defined networking that we offer [16]. Table **??** summarizes the demographics of the participants: about 870 students completed the survey, 216 of whom were full-time network operators. The majority of the students who completed the assignment and survey had more than 5 years of experience in networking, and many had more than 15 years of experience. More than 200 of the students had experience with networks of more than 1,000 devices, and more than 300 of the students had experience with networks with more than 10,000 users. Most of the participants had the most experience with campus or enterprise networks.

The responses we received demonstrate a clear need for better tools for automating and verifying network control. Nearly 20% of participants said that they must change their network configurations more than once a day. The most common causes of changes were provisioning, planned maintenance, and updates to security policies—exactly the types of configuration changes that we aim to automate with Kinetic. More strikingly, 89% of respondents indicated that they were never completely certain that their changes to the configuration would not introduce a new problem or bug, and 82% were concerned that the changes would introduce problems with *existing* functionality that was unrelated to the change. The two most common aspects of configuration that operators wanted to see *automated* were correctness testing (37%) and quality of service and performance assurances (24%). The two most common aspects of configuration that participants wanted to see *verified* were general correctness problems (37%) and security properties (26%). We asked

Table 9: Computation tree logic (CTL) operators.

Operator	Meaning
	(*Quantifiers over Groups of Paths)*
A ϕ	ϕ holds for all possible paths from the current state.
E ϕ	There exists a paths from the current state where ϕ holds.
	(*Quantifiers over a Specific Path)*
X ϕ	ϕ holds for ne**X**t state.
F ϕ	ϕ eventually holds sometime in the **F**uture.
G ϕ	ϕ holds for all current and following states, **G**lobally.
ϕ U ψ	ϕ holds at least **U**ntil ψ.

these same participants to write programs in Kinetic and other SDN controllers; we discuss the results of that part of our user study in Section 4.5.1.

4.2.2 Background: Pyretic and CTL

Pyretic.. To develop a language for expressing control dynamics that is both concise and easy to use, we based the Kinetic language on Pyretic [75], a Python-embedded domain-specific programming language for writing SDN control programs. It encodes network data-plane behavior in terms of policy functions that map an incoming "located packet" (*i.e.*, a packet and its location) to an outgoing set of located packets. Pyretic has a `policy` variable that determines the actions that the control program applies to incoming packets (*e.g.*, filtering, modification, forwarding). Pyretic ultimately compiles policies to OpenFlow-based switches. Pyretic's composition operators provide straightforward mechanisms for composing multiple distinct policies into a single coherent control program. Pyretic's parallel composition operator (+) makes a copy of the original packet and applies the corresponding policies to each copy in parallel. Sequential composition (>>) applies policies to a packet in sequence, so that the second policy is applied to the packet that is the output of the first policy. Pyretic is extensible, and its support for composing distinct policies and dynamically recompiling flow-table entries whenever the `policy` variable is updated are useful features for Kinetic. Still, the language itself does not provide a framework for writing concise, intuitive policies that respond to changing conditions, which is Kinetic's goal.

Model checking.. We wanted to design Kinetic so that policies were not only easy to automate, but also easy to verify. To do so, we applied a model checking framework developed by Clarke and Emerson [13, 14] and subsequently refined by McMillan [72]. Model checking can guarantee that a finite state machine (FSM) satisfies certain properties that are expressed in different types of logics; this feature makes FSMs a logical choice for expressing Kinetic policies. One such logic is computation tree logic (CTL), a branching-time logic that represents time as a tree structure. The initial state of an FSM is the root, and each node represents a different future state. A path through the tree represents an execution path of the FSM. CTL allows the expression of various types of temporal logic statements, such as those expressed in Table **??**. NuSMV is a widely used symbolic model checker for FSMs [12]. The Kinetic compiler automatically translates Kinetic programs into an SMV model, which can be tested against various CTL-based assertions.

4.3 Kinetic by Example

We illustrate various features of Kinetic by way of example programs. All of the examples that we present in this section are verifiable; we defer a discussion of verification, as well as the details of the Kinetic language and runtime, to Section 4.4. We have selected examples that demonstrate the design features of Kinetic; the Kinetic Github repository has more examples [49].

Kinetic programs capture control dynamics with a finite state machine (FSM) abstraction. To illustrate this abstraction, we start with a simple example involving intrusion detection. Although FSMs are intuitive, representing all possible network states in a monolithic FSM would result in state explosion; the second and third examples illustrate two abstractions that address this challenge: Located Packet Equivalence Classes (LPECs) and FSM composition. Finally, to show Kinetic's generality, we present a MAC learning switch implementation.

4.3.1 Capturing Dynamics

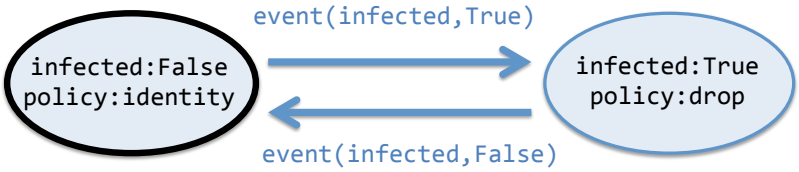


Figure 17: Intrusion detection FSM.

We begin with a simple dynamic policy involving intrusion detection. Suppose that a network operator wants the network to drop all packets to and from a host once it receives an event indicating that the host is infected (*e.g.*, from an intrusion detection system). Kinetic allows operators to concisely express these dynamics with finite state machines that determine how a policy should evolve in response to events such as intrusions. We chose FSMs as the basic abstraction for expressing Kinetic programs because (1) they intuitively and concisely capture control dynamics in response to network events; and (2) their structure makes them amenable to verification.

In this example, each host would have a single state variable, `infected`. When `infected` is false, the controller applies Pyretic's `identity` (allow) policy for traffic from that host; when it is true, the controller applies Pyretic's `drop` policy for the host's traffic. Figure 17 shows this logical FSM. To support verification, the actual specification of the FSM for this policy is slightly more complicated; we expand on this example in Section 4.4.2.

4.3.2 Capturing State for Groups of Packets

Defining FSMs in Kinetic has the potential to create state explosion, since dynamic policies must be defined over a state space that is exponential in the number of hosts and flows (and possibly other aspects of the network). For example, consider the previous example, a two-state FSM indicating whether a host is infected. If the network has N hosts, then representing the state of the network requires an FSM with 2^N states, which is intractable,

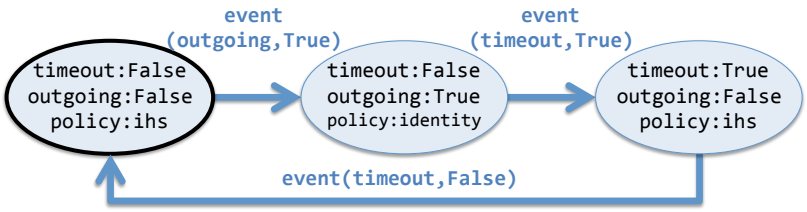


Figure 18: Stateful firewall FSM.

particularly as the size of the network and the complexity of policies grow. Instead of directly encoding an FSM that explicitly encodes all variable values, Kinetic encodes a single generic FSM that can be applied to any given group of packets (*e.g.*, all packets from the same host, in the case of the previous example). Each group of packets has a separate FSM instance; packets in the same group will always be in the same state. We call such a group of packets a *located packet equivalence class (LPEC)*.

To illustrate the use of LPECs, we describe the implementation of a stateful firewall that implements a common security policy. Figure 18 shows the Kinetic representation of the policy. This program always allows outbound traffic, but blocks inbound traffic unless the traffic flow is in response to corresponding outbound traffic for that flow. For example, if internal host ih_1 pings external host eh_2 then packets sent from eh_2 should be allowed back through the firewall until a certain timeout occurs, but only if ih_1 is the destination.

The firewall's initial state, in the left of the figure, shows the policy, `ihs`, which is a filter policy matching all traffic whose source address in the set of internal hosts. A Pyretic-encoded query collects outbound packets from hosts in `ihs` and produces (`outgoing,True`) event. This triggers the update of the `policy` variable to `identity` (indicating that traffic is now allowed), and `outgoing` is reset. The `timeout` event is provided by Kinetic event driver. After certain amount of time (*e.g.*, five seconds), a (`timeout,True`) event is invoked unless another outgoing packet is seen within the timeout. The program should regard inbound and outbound flows between the same pairs of endpoints with the same

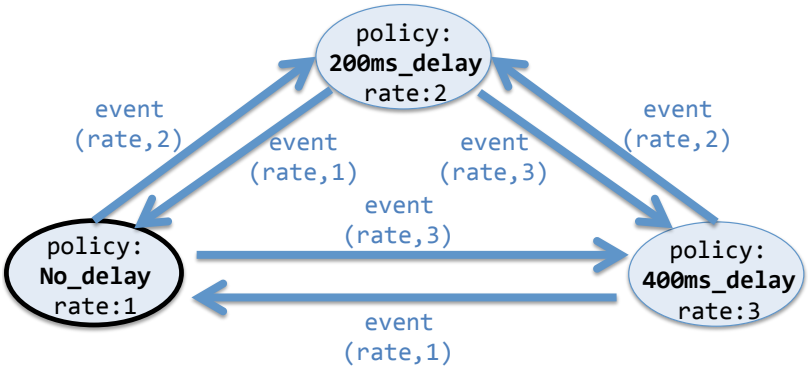


Figure 19: Data usage-based rate limiter FSM.

state, and the programmer should not have to explicitly encode state for every pair of end-points. To implement such a policy, the programmer can define an LPEC to correspond to a distinct source-destination IP address pair:

```
1 def lpec(pkt):
2   h1 = pkt['srcip']
3   h2 = pkt['dstip']
4   return (match(srcip=h1,dstip=h2) |
5           match(srcip=h2,dstip=h1) )
```

4.3.3 Composing Independent Policies

Many aspects of network state are logically independent. For example, whether a host has authenticated is independent of whether it is infected or whether it has exceeded a usage cap. This independence allows a programmer to represent the overall network state as a product automaton that can be decomposed in terms of simpler *tasks*, where each task has simpler (and smaller) FSMs. This example shows the composition of four independent network tasks.

In our survey of campus network policies, we found nearly 20 university campuses [62] that implemented some form of usage-based rate-limiting (*e.g.*, [15]). Network operators currently implement these policies using low-level scripts that interact with monitoring devices. Kinetic provides intuitive mechanisms for implementing such a policy. Figure 19

illustrates the FSM for a usage-based rate limiter, which forward traffic with different delays depending on the user's historical data usage patterns. By default, traffic is forwarded with no delay; depending on the events that the controller receives concerning usage, the controller may institute a policy that introduces additional delay on user traffic. (OpenFlow 1.0 does not support traffic shaping, so we use variable delay as an illustrative example; Kinetic could be coupled with controllers that support later versions of OpenFlow that can do traffic shaping.)

Naturally, a real network would not only have policies involving quality-of-service, but also other policies, such as those relating to authentication and security. For example, a control program might first check whether a host is authenticated, either through a Web login or via 802.1X mechanism. Subsequently, the host's traffic might be subject to an intrusion detection policy that allows traffic by default but blocks the traffic if an infection event occurs. Finally, it might be sequentially composed with the rate-limiting policy above, yielding the resulting policy:

```
1 (web_auth + 802.1X_auth) >> ids >> rate_limiter
```

To verify this program, Kinetic generates a single FSM model for input to a model checker. Thus, programmers can write CTL specifications for the resulting composed policy, not only for individual policies. For example, a logic statement involving the combination of policies such as "If a host is authenticated either by the web authentication system or with 802.1X and is not infected, the resulting policy should never drop packets" can be verified with a single CTL assertion, as shown in Table 10. (Section 4.4.4 discusses verification in more detail.)

4.3.4 Handling General Event Types

Figure 20 shows a Kinetic FSM for a MAC learner that responds to both packets from hosts and topology changes. Although the implementation of a MAC learning switch is just as simple in other languages (indeed, it is the "canonical" reference program for SDN

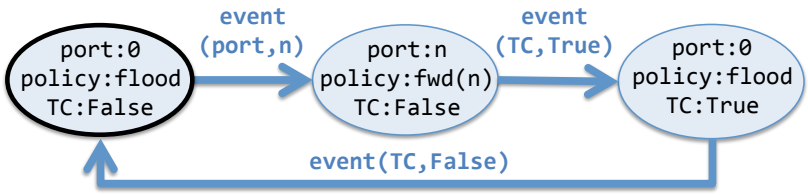


Figure 20: MAC learner FSM.

controllers), we present this example to illustrate that Kinetic programs can handle a variety of event types, including packet arrivals.

This program responds to two different types of events: `TC` (topo_change) and `port` events. The `TC` event is a built-in event that is invoked automatically whenever a topology change occurs. In Kinetic, programs can register and react to this built-in event. The `port` events are generated by a Pyretic query that collects the first packet for each (switch,srcmac) pair. The values of `policy` are defined by that of `port`: the value is `flood` when `port` is 0, and `fwd(n)` when `port=n`. Initially `port` is 0 (indicating the port has not yet been learned), and `TC` is `False`. When a `(port,n)` event arrives, which is invoked by the Pyretic runtime when it sees a packet from an unseen host, a transition occurs, setting the port to the value learned and the policy to unicast out that port. The MAC learner then unicasts packets to the appropriate hosts until a topology change occurs, triggering the transition to the right-most state in which `TC` is `True`, resulting in flooding for packets corresponding to that LPEC (*i.e.*, switch-source MAC address pair).

4.4 Kinetic Design & Implementation

We describe the details of Kinetic's architecture, language, runtime, and verification engine.

4.4.1 Architecture

We now describe the Kinetic system architecture, including the design of the Kinetic programming language. Figure 21 shows the Kinetic architecture, which is built on the Pyretic runtime. At the highest level, a Kinetic program has three parts: (1) a finite state machine (FSM) specification; (2) a specification of portions of flow space that are always in the

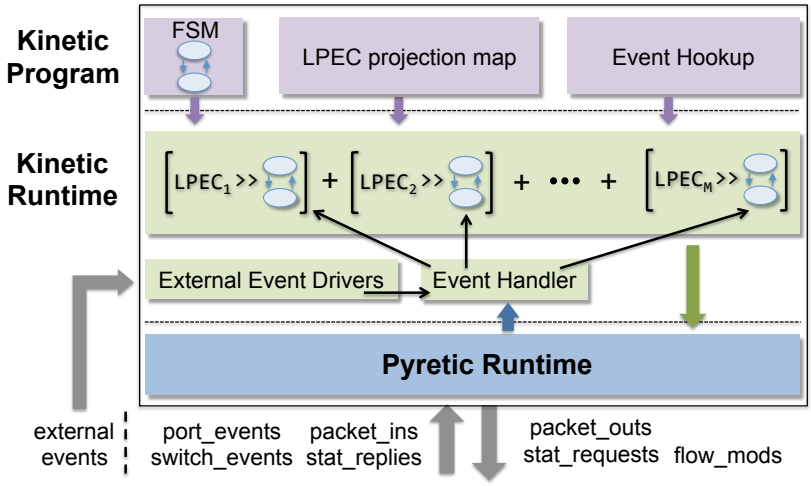


Figure 21: Kinetic architecture.

same state in any given FSM (an LPEC); and (3) mechanisms for incorporating external events that could change the state of any given LPEC's FSM.

Kinetic instantiates copies of programmer-specified FSMs (one per LPEC); the Kinetic event handler sends incoming events, which can arrive either from external event hookups or from the Pyretic runtime (*e.g.*, in the case of certain types of events such as incoming packets), to the appropriate FSMs. Kinetic FSMs register with one or more event drivers and update their states when new events arrive, responding to incoming events that may be processed by those drivers. Kinetic supports both native events and generic JSON events. Because Kinetic is embedded in Pyretic, these functions can be executed using Pyretic's runtime. We use the Pyretic runtime to exchange OpenFlow messages with the network switches; we also use the Pyretic runtime to handle certain types of events, such as those related to either network topology or traffic.

4.4.2 Language and Abstractions

We offer a complete description of the language and then discuss LPECs and FSM composition in more detail.

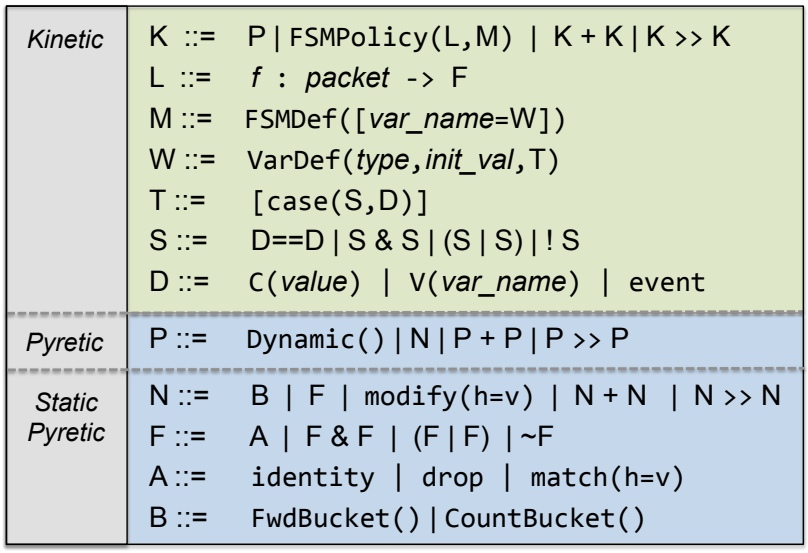
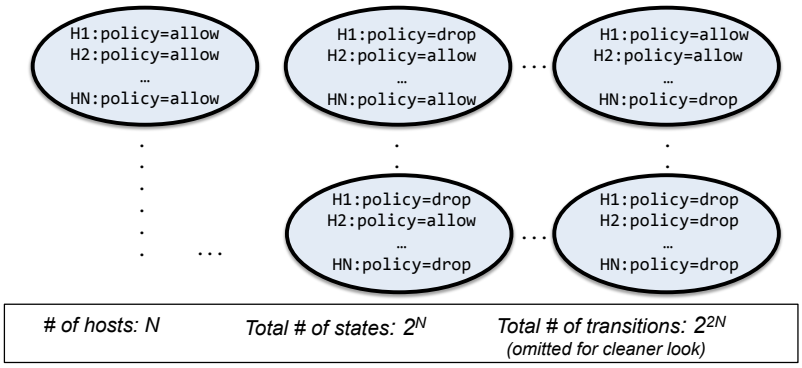


Figure 22: The Kinetic language grammar.

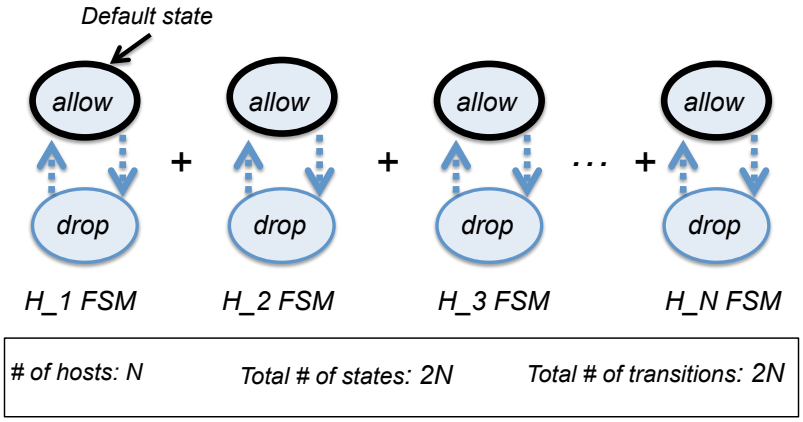
4.4.2.1 Language Overview

Figure 22 defines the Kinetic language, which extends Pyretic (**P**). Pyretic has *bucket* policies (notated by **B**) which collect packets and count packet statistics, respectively; *primitive filters* (**A**) and *derived filters* (**F**) that allow only matching packets through; and *static policies* (**N**). Static policies include buckets, filters, the `modify` policy, and the combination of these via parallel and sequential composition. `Dynamic` generates a stream of static policies and can be combined with other policies in parallel or sequence.

Kinetic extends the Pyretic DSL with a subclass of `Dynamic`—`FSMPolicy`—which takes two arguments: an LPEC projection map (**L**) and an FSM description (**M**). The LPEC projection map takes a packet and returns a filter policy. The FSM description is set of assignments from a variable name to a variable definition (**W**). Each variable is defined by its type, initial value, and associated transition function (**T**). Each transition function is a list of cases, each of which contains a test (**S**) and an associated basic value (**D**) to which this corresponding state variable will be set, should this case be the first one in which the test is true. Tests are the logical combination of other tests (using *and*, *or*, *not*) or equality comparison between basic values. Finally, basic values are constants (**C(value)**), state



(a) Explicit encoding is exponential in **N**.



(b) Decomposing to **N** LPEC FSMs.

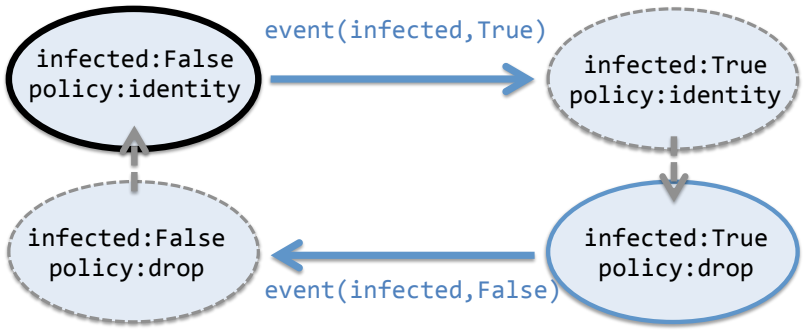
Figure 23: Reducing state explosion using an LPEC FSM.

variables (**V(variable_name)**), and events (**event**).

4.4.2.2 Located Packet Equivalence Classes

Recall from Section 4.3 that an LPEC allows an operator to encode a generic FSM for groups of packets (*e.g.*, all packets with the same source MAC address). Each distinct LPEC will have its own FSM instance, and the group of packets in each LPEC will be in the same state. Because each LPEC refers disjoint sets of packets, their FSMs (and corresponding policies) can be maintained independently, thus allowing their policies to be encoded in parallel. This mechanism allows the programmer to avoid explicit encoding of all combinations of network states (as shown in Figure 23a) and instead express each LPEC's FSM independently and compose them in parallel, as shown in Figure 23b.

Each LPEC has an FSM, which has a set of states, where each state has a Pyretic policy;



(a) Actual implementation of the Kinetic FSM.

```
1  @transition
2  def infected(self):
3    self.case(occurred(self.event),self.event)
4
5  @transition
6  def policy(self):
7    self.case(is_true(V('infected')),C(drop))
8    self.default(C(identity))
9
10 self.fsm_def = FSMDef(
11   infected=FSMVar(type=BoolType(),
12                   init=False,
13                   trans=infected),
14   policy=FSMVar(type=Type(Policy,{drop,identity}),
15                 init=identity,
16                 trans=policy))
17
18 def lpec(pkt):
19   return match(srcip=pkt['srcip'])
20
21 fsm_pol = FSMPolicy(lpec,self.fsm_def)
```

(b) Kinetic code that implements the Kinetic FSM.

Figure 24: Logical FSM for an IDS in Kinetic, and the Kinetic code that implements the policy.

and a set of transitions between those states, where transitions occur in response to events that the operators defines. When events arrive, the respective LPEC FSMs may transition between states, ultimately inducing the Pyretic runtime to recompile the policy and push updated rules to the switches. In Kinetic, a programmer can specify an LPEC in terms of a Pyretic filter policy. For example, `match(srcip=pkt['srcip'])` defines an LPEC FSM for each unique source IP address.

Returning to our IDS example from Section 4.3 (Figure 17), Figure 24b shows the code for the Kinetic program that implements the simple intrusion detection example from Section 4.3. Each host (*i.e.*, source IP address) can have a distinct state, so we need an

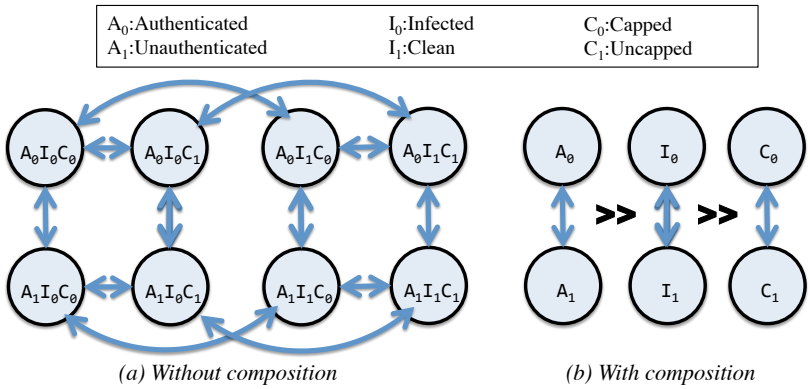
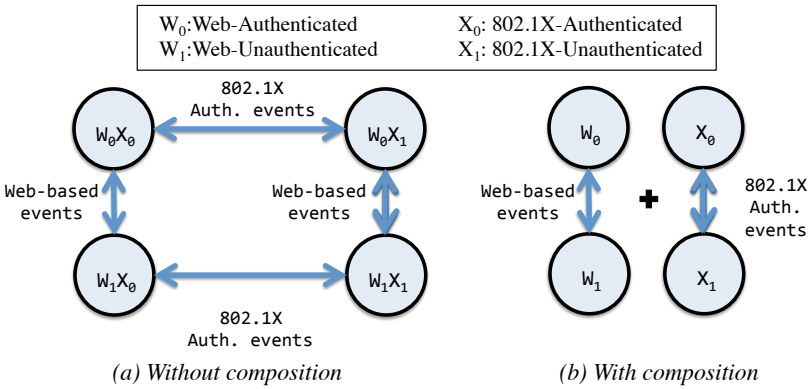


Figure 25: Composing independent tasks in sequence.

LPEC FSM per source IP address; lines 18–19 define the LPEC. To define an FSM that is amenable to model checking, we must separate the `infected` variable and the corresponding `policy` variable into two separate states, as shown in Figure 24a. *Exogenous* events trigger transitions between the `infected` variable states; a change in this variable's value in turn triggers an *endogenous* transition of the `policy` variable, which ultimately causes the Pyretic runtime to recompile flow-table entries for the network switches. Lines 1–3 in Figure 24b define the exogenous transition for `infected`; lines 5–8 defined the endogenous transition for `policy` (note that the value of `policy` is defined in terms of the value of `infected`). Finally, lines 10–16 define the FSM itself, in terms of the two variables; the FSM definition is simply a set of FSM variables, each of which has a type, an initial value, and a transition function.

4.4.2.3 FSM Composition

In Section 4.3, we showed an example of a campus network policy that composed FSMs for independent network tasks to control state explosion. Without FSM composition, a programmer would need to define FSMs for $\Pi_{i=1}^{N} a_i$ possible states, where a_i is the number of possible states for task i and N is the total number of tasks. Decomposing the product automaton reduces state complexity from exponential to linear in the number of independent tasks. For example, given ten tasks, each with two states, a monolithic program would



(a) Without composition *(b) With composition*

Figure 26: Composing multiple authentication tasks in parallel. Any successful authentication would result in allowing the host's traffic.

require 1,024 states, as opposed to just 20.

Pyretic allows policies to be composed either in parallel (*i.e.*, on independent copies of the same packet) or in sequence (*i.e.*, where the second policy is applied to the output from the first). It turns out that these operators are *also* useful for reducing state explosion. Figure 25 illustrates how sequential composition can reduce state complexity by decomposing a larger product automaton. Consider a simple control program that puts the host into a walled-garden until it has authenticated, quarantines the host if an infection has been detected, and rate limits a host if it has exceeded a usage cap. Each of these tasks has two possible states: authenticated or not, quarantined or not, capped or not, resulting in 2^3 possible states. By applying `auth >> IDS >> cap`, the same network control program requires only $2 \cdot 3$ states.

Figure 26 shows how parallel composition reduces state complexity. A Kinetic program might specify that either a network flow should be authenticated by a Web authentication mechanism or 802.1X. If *either* of these tasks places the host in an authenticated state, the host should be allowed to send traffic. Without composition, the network state machine would need a second set of states, requiring 2^N states, where N is the number of authentication tasks (in this case, $N = 2$). (Clearly, even more states would be needed if any independent task could assume more than two states.) As before, decomposition reduces

this to $\Sigma_{i=1}^{N} a_i$ states, where a_i is the number of possible states for task i, and N is the number of tasks.

4.4.3 Runtime

We now explain optimizations to the Kinetic runtime to support the efficient compilation of the large finite state machines that might result from networks with many hosts and policies and high event rates. The Kinetic runtime's main challenge is storing and processing the joint state of all LPEC FSMs to produce a single set of forwarding table entries in the network switch. To accomplish this goal, the runtime first decomposes the FSMs with combinators to achieve a representation of the network state that is linear in the number of hosts and policies. Second, Kinetic optimizes the compilation process itself by recognizing that the LPEC FSMs typically operate on disjoint flow space, which allows for optimizations that dramatically speed up parallel composition. Finally, Kinetic only expands the LPEC FSMs for which a transition has actually occurred. We describe each of these optimizations below.

Decomposing the product automata.. A Kinetic `FSMPolicy` encodes the complete FSM as the *product automaton* [19] of the individual LPEC FSMs. We can represent the Pyretic policy for the entire network, given a global network state s as the following product automata:

$$\text{policy} \quad = \quad \sum_{i=1}^{N} (\text{lpec}_\text{i} \;>>\; \text{lfsm}_i(s))$$

where the summation operator represents parallel composition of the corresponding policies, and each LPEC generator produces the appropriate packets that are processed by the corresponding LPEC FSM in state s.

Fast compilation of disjoint LPECs.. Compilation of policies that are composed in parallel is computationally expensive, as it requires producing the cross-product of all match and action rules: it involves computing the intersection of match statements and the union

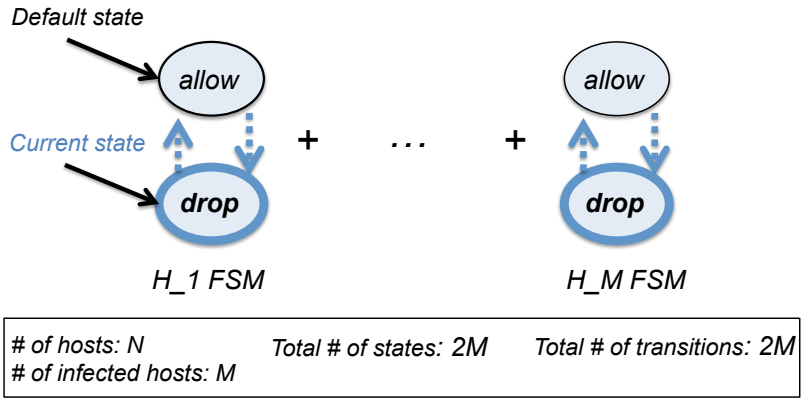


Figure 27: Expanding only **M** LPEC FSMs that have changed.

of actions, for every pair of match and action pairs between the two policies. If LPECs are disjoint, however, the resulting policies can simply be combined without explicitly computing the intersection of the match statements: the rules from each LPEC FSM can simply be inserted into the flow tables.

Default policies and on-demand LPEC FSM expansion.. Even a linear-sized representation may not scale. For example, the LPEC generator shown in Section 4.4.2.2 would generate 2^{32} LPECs if it were fully expanded, while a generator for each pair of hosts (based on hardware address) would produce 2^{96} LPECs. Fortunately, because all LPEC FSMs are generated from the same FSM specification, they start with the same initial state and, hence, the same default policy. Thus, Kinetic does not need to expand the FSM for an LPEC *unless and until it experiences a state transition*; until that point, the Kinetic runtime can simply apply whatever default policy is defined for that FSM. Figure 27 highlights this on-demand expansion.

Kinetic's runtime optimizations reduce the computational complexity of compilation from exponential in the number of LPECs (Figure 23a), to linear in the number of LPECs (Figure 23b), and finally to linear in the (much smaller) number of LPECs that have actually experienced a transition (Figure 27). Kinetic additionally employs additional optimizations, such as memoizing previously compiled policies, as other applications have used [32].

```
1  MODULE main
2    VAR
3      policy   : {identity,drop};
4      infected : boolean;
5    ASSIGN
6      init(policy)   := identity;
7      init(infected) := FALSE;
8      next(infected) :=
9        case
10         TRUE     : {FALSE,TRUE};
11       esac;
12     next(policy) :=
13       case
14         infected : drop;
15         TRUE     : identity;
16       esac;
```

Figure 28: NuSMV FSM model for IDS policy from Figure 24b.

4.4.4 Verification

When the programmer executes a Kinetic program, Kinetic automatically creates an FSM model for the NuSMV model checker. Kinetic obtains information about each state variable (*e.g.*, type, initial value, and transition relationship) by parsing the `fsm_def` data structure; Kinetic parses the transition function for additional information about transitions, which often depend on other variables.

Kinetic then uses NuSMV to test CTL specifications that the programmer writes against the FSM model. Kinetic outputs the CTL specifications that passed; for any failed specifications, Kinetic produces a counterexample, showing the sequence of events and variable changes that violated the specification. In addition to single `FSMPolicy` objects, Kinetic can convert composed policies into a single model that can be verified. For example, although the programmer specifies a composed policy as in Figure 25b and Figure 26b, verification will execute on a combined FSM model as in Figure 25a and Figure 26a.

Figure 28 shows the NuSMV FSM model corresponding to the IDS policy from Figure 24b. The model definition has two parts. The first is `VAR`, which declares the names and types of each variable (lines 2–4). The second is `ASSIGN`, where current and future variable values are assigned, using two functions for each variable: an `init` function that determines the variable's initial value (line 5–7), and a `next` function that specifies what

Table 10: NuSMV CTL rules for different Kinetic programs.

Program	CTL	Description
Mac learner	AG (topo_change → AX policy=flood)	Always resets to flooding when topology changes.
	AG (policy=flood → AG EF (port>0))	Can always go from flooding to unicasting a learned port.
	! AG (port=1 → EX port=2)	It is impossible to update the learned port without first flooding.
	AG (port>0 → A [port>0 U topo_change])	Port will stay learned until there is a topology change.
Stateful firewall	AG (outgoing & !timeout → AX policy=identity)	If first packet originated from internal host and timeout did not occur, the system should allow traffic.
	AG (outgoing & timeout → AX policy=matchFilter)	If first packet originated from internal host, but timeout occurred, the system should shut down traffic (apply match filter).
	AG (!outgoing → AX policy=matchFilter)	If first packet is not from internal host, the system should not allow traffic (apply match filter).
Composed policy	AG (infected → AX policy=drop)	If host is infected, drop packets.
	AG ((authenticated_web | authenticated_1x) & !infected → AX policy!=drop)	If host is authenticated either by Web or 802.1X, and is not infected, packets should never be dropped.
	AG (authenticated_web & !infected & rate=2 → AX policy=delay200)	If host is authenticated by Web, not infected, and the rate is 2, delay packets by 200ms.

value or values the variable may take, as a function of the current values of other variables in the model (line 8–16).

Within the `case` clause of each `next` function, the left-hand side shows the condition, while the right-hand side shows the variable's next value if the condition holds. `TRUE` on the left-hand side refers to a default transition. Lines 8–11 indicate that the `infected` variable can change between `FALSE` and `TRUE`, independent of any other state variable (in reality, the value changes based on external event of the same name). The `policy` variable in lines 12–16 shows that the value transitions to `drop` if `infected` is `True`, while the default is `identity`. Thus, it shows that `policy`'s next value depends on the `infected` variable. Table 10 shows examples of the types of temporal properties that Kinetic can verify.

4.5 Evaluation

In this section, we evaluate two aspects of Kinetic: (1) Does Kinetic make it easier for network operators to configure realistic network policies? (Section 4.5.1); and (2) How does Kinetic's performance scale with the number of flows, users, and policies? (Section 4.5.2).

4.5.1 Programming in Kinetic: User Study

Evaluating whether a new network configuration paradigm such as Kinetic makes it easier for network operators to write network policies is challenging. Network operators already know how to use existing tools and infrastructure, and deploying a new control framework requires overcoming both the inertia of network infrastructure that is already deployed *and* the knowledge base of network operators, many of whom are not programmers by training. We needed to find a way to ask many network operators to evaluate Kinetic in light of these obstacles. Fortunately, the Coursera course on software-defined networking that we teach [16] offers precisely this captive audience, as the course's demographic includes many network operators who are both educated about SDN and willing to experiment with cutting-edge tools. (Section 4.2 and Table **??** explained the initial survey and described the demographics of the participants.) We obtained approval from our institutional review board (IRB) to ask students to use Kinetic and other SDN controllers to complete a simple network management task and subsequently survey them.

We asked the students in the course to write a "walled garden" controller program that is inspired from real enterprise network management task that we have learned about in our discussions with network operators [17]. In summary, the students were asked to write a program that permitted all traffic to and from the Internet unless a host was deemed to be infected (*e.g.*, as determined from an intrusion detection system alert) *and* not a host that was exempt from the policy (one might imagine that certain classes of users, such as high-ranking administrators or executives would get different treatment than strict interruption of service). In the assignment, we asked the students to: (1) Write a Kinetic

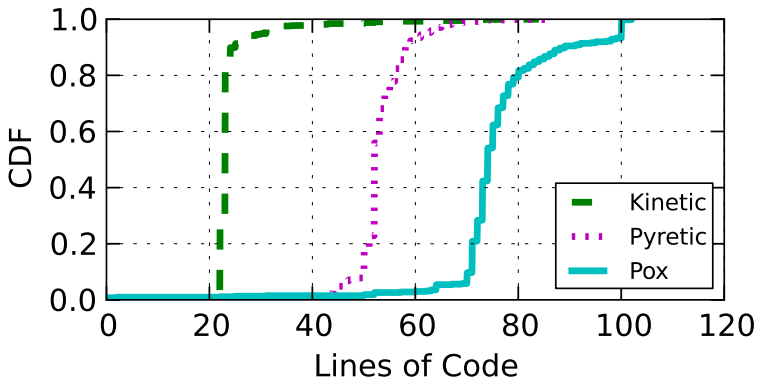


Figure 29: The lines of code required to implement the walled-garden program in different controller languages.

program that implements the policy; (2) Choose either Pyretic or POX to implement the same policy; (3) Optionally implement the policy in the remaining controller; (4) Answer survey questions about their experiences with each controller.

The course devoted one week each to each of the three controller platforms, and students had already completed assignments in both POX and Pyretic, so if anything, students should have found those platforms at least as familiar as Kinetic. In fact, there were three programming assignments in Pyretic while there was only one for Kinetic. Kinetic was discussed in only one lecture out of eight lectures in the course, and was not treated specially. To further minimize the bias in favor of Kinetic, students were instructed to complete the assignment in Kinetic first, as the first attempt is usually the hardest. With better understanding of the assignment, it is likely that programming in Pyretic or POX would have been easier.

Of the students who completed the survey from Section 4.2, 667 attempted the assignment, and 631 successfully completed it (a 95% completion rate), and 70% of those students completed the programming assignment in less than three hours. We asked students who did not complete the assignment why they did not complete it; most referenced external factors such as time constraints, as opposed to anything pertinent to Kinetic.

To compare the complexity of the different control programs, we compare the lines of

Table 11: Lines of code to implement programs in each controller.

Programs	**FL**	**Pox**	**Pyretic**	**Kinetic**
ids/firewall	416	22	46	17
mac_learner	314	73	17	33
server load balancer	951	145	34	37
stateful firewall	–	–	25	41
usage-based rate limiter	–	–	–	30

code in programs implemented with different controllers; we then conduct qualitative measurements by surveying the students of the course. Although the lines of code for a program depends on the language, programmer, and implementation style, a high-level comparison can nevertheless yield a rough but meaningful sense for the relative simplicity of a Kinetic program. Figure 29 shows the distribution of the lines of code that students needed to implement the walled-garden program in different controller languages. About 80% of the implementations using Kinetic required about 22 lines of code; in contrast, more than half of the Pyretic implementations required more than 50 lines, and half of the assignments written in POX required more than 75 lines of code. The fact that Kinetic requires fewer lines of code to implement this program highlights the utility of the abstractions that Pyretic provides. In addition to the experiment from the Coursera course, where we could find publicly available implementations on the Web, we compared the number of lines of code for several different programs to our own implementations of the same programs in Kinetic. (Blank entries in the table indicate that no implementation was available.) Table 11 shows these results, for four different controllers: Floodlight, POX, Pyretic, and Kinetic. The public Pyretic programs are occasionally slightly shorter than the corresponding Kinetic programs because they only handle built-in events such as packet arrivals and topology changes. Programs that need to handle arbitrary events would likely always be shorter in Kinetic.

In addition to analyzing quantitative measures such as lines of code, we asked students more qualitative questions about their experiences using Kinetic to implement the walled-garden assignment, relative to their experiences with Pyretic and POX. We asked students

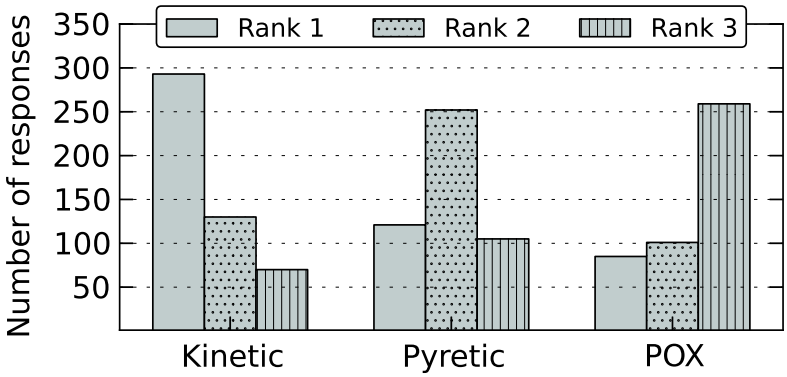


Figure 30: Number of students who preferred each controller.

to rank the controllers based on ease of use, as well as which platform they preferred. Figure 30 shows some highlights from this part of the survey. Of the three controllers, more than half of the students preferred writing the assignment in Kinetic versus either Pyretic or POX. We asked students whether Kinetic could make it easier to configure and verify policies in their networks. About 79% of students thought that configuring the network with Kinetic would be easier than current approaches, and about 84% agreed that Kinetic would make it easier to verify network policies.

Students who chose Kinetic as the language that they preferred best cited its abstractions, FSM-based structure, and support for intuition (*e.g.*, "Kinetic is more intuitive: the only thing I need to do is to define the FSM variable.", "intuitive and easy to understand", "reduces the number of lines of code", "programming state transitions in FSMs makes much more sense", "the logic is more concise"). Some students still preferred Kinetic, despite the fact that the syntax had a steeper learning curve: "Kinetic took less time and was actually more understandable using the templates even though the structure was very 'cryptic'... I thought the Pyretic would be the easiest...[but] I spent a lot more time chasing down weird bugs I had because of things I left out or perhaps didn't understand." Interestingly, many of the students actually preferred the lower-level trappings of POX to Pyretic (*e.g.*, "Pyretic was friendly, but the logic more intricate"). The results of this experiment

Table 12: The frequency of network events on a primary campus network, which we use for a trace-driven evaluation of Kinetic.

Statistic	# per day
Total Unique Authenticated Users	22,586
Total Unique Devices Authenticated	41,937
Number of WPA authentication Events	1,330,220
Number of WEB authentication Events	1,850

and survey highlight both the advantages and disadvantages of Kinetic's design, as well as the difficulty of designing "northbound" languages for SDN controllers: without intuitive abstractions, operators may even *prefer* the lower-level APIs to higher-level abstractions.

4.5.2 Performance and Scalability

We evaluate Kinetic's performance and scalability when handling incoming events, as well as the performance and scalability of verification, as those are the two main contributions of our work. We evaluated the Kinetic controller on a machine with an Intel Xeon CPU E5-1620 3.60 GHz processor and 32 GB of memory. We measured raw packet forwarding performance but do not focus those numbers, as the forwarding performance is not the focus of our work and is equivalent to what can be achieved with POX and Pyretic, in any case. Similarly, the rate at which updated rules can be installed depends on lower layers (*e.g.*, POX). Optimizing the number of rule updates [103] and applying them consistently [91] have been studied in previous work, so we do not focus on those aspects here.

Event handling and policy recompilation.. Because Kinetic recompiles the policy whenever an event causes a state transition, we must evaluate how fast Kinetic can react to events and recompile the policy for realistic network scenarios. We used the wireless network from a large university campus with more than 4,300 access points deployed across 200 buildings; the network authenticates nearly 42,000 unique devices for more than 22,000 users every day. Table 12 summarizes these statistics. On such a network, Kinetic would have to keep track of an equivalent number of devices, and each authentication event (about 1.3 million per day) would require Kinetic to recompile policy, resulting in an average of

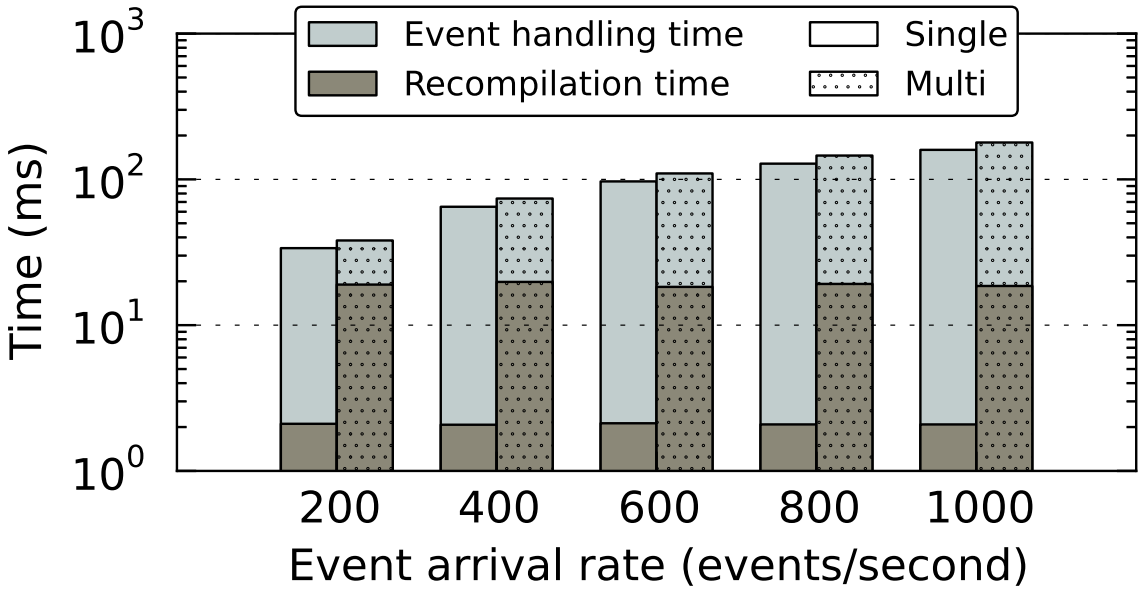


Figure 31: Time to handle a batch of incoming events and recompile policies in Kinetic, for different event arrival rates and policies.

15 events per second (though certainly higher during peak periods). We evaluate Kinetic for event arrival rates for up to 1,000 events per second, for both a single-FSM policy and a policy involving the composition of multiple FSMs, based on the example in Section 4.3.3. We create a Kinetic program that results in 42,000 LPEC FSMs and randomly distribute authentication events across these FSMs (*i.e.*, devices).

Figure 31 presents the results of this experiment. Recompilation time is longer for the program with multiple FSMs composed together as it embeds a more complex policy than the program with a single FSM. For both programs, event handling time increases as event arrival rate increases. Even for event arrival rates that are several orders of magnitude more frequent than an actual campus network, Kinetic's event handling and recompilation times are around (or less than) a few hundred milliseconds.

Verification.. The speed of verification depends on the performance of NuSMV, which in turn depends on three factors: (1) the size of the given FSM (*i.e.*, number of states and transitions), (2) the number of properties to verify, and (3) the kinds of properties to verify. To observe whether Kinetic's verification time is reasonable, we evaluate Kinetic's

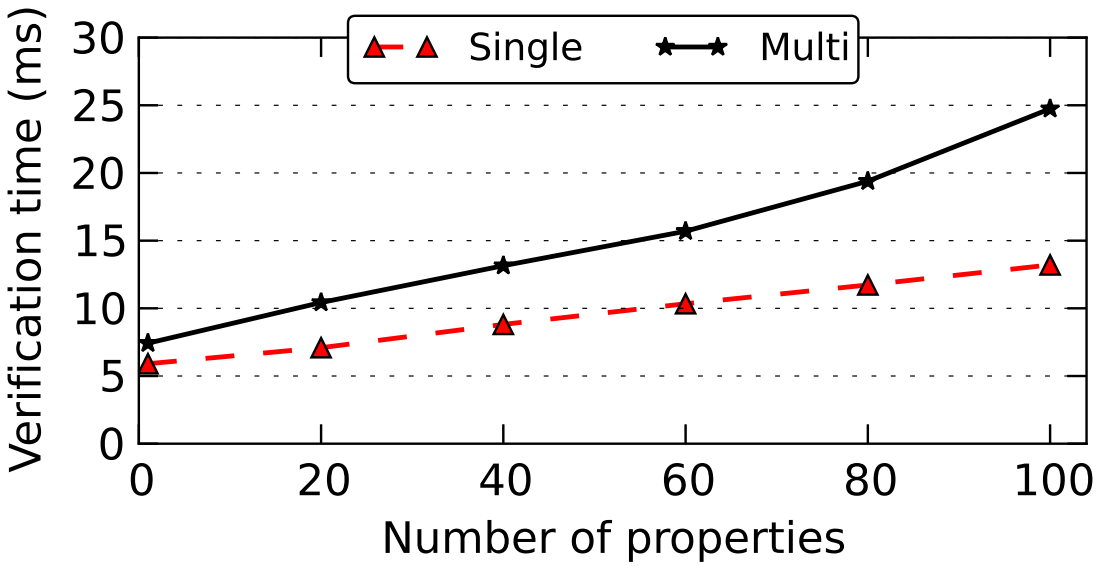


Figure 32: Verification time as a function of the number of CTL properties that Kinetic checks, for a policy with a single FSM and a policy with a composition of multiple FSMs.

verification performance with the same programs that we used to evaluate Kinetic's event handling and recompilation performance. The single-FSM authentication program produces an FSM with four states, and the program with multiple FSMs produces a combined FSM with 384 states. To test Kinetic with more than the handful of CTL specifications we manually created, we generate over one hundred specifications using random combinations of CTL operators. They are all syntactically correct (*i.e.*, NuSMV will not complain about the syntax), but are generated regardless of whether they will be true or false when each goes through the model checker, as our goal is merely to measure verification time.

Figure 32 shows the time to complete verification for different Kinetic programs with different numbers of properties. Each experiment had 1,000 independent trials; the variance across experiments was small, so we do not show the error bars. As expected, a Kinetic program with a larger FSM model takes longer to finish. The figure also shows that the number of properties affects verification time, but all verification finishes within 35 milliseconds. Kinetic performs verification before the Kinetic program ever runs, so this process has no effect on performance.

4.6 Related Work

We discuss SDN controllers with verifiable properties, approaches for formally verifying data-plane behavior, and other high-level SDN control languages.

Formal verification of SDN control programs.. FlowLog [78] provides a database-like programming model that unifies the control-plane logic with data-plane state and controller state. Aspects of FlowLog programs can be verified, but because the language does not naturally capture state transitions and temporal relationships, it cannot verify arbitrary temporal relationships, such as those that can be verified with CTL in Kinetic. FlowLog uses Alloy to perform bounded verification, so its analysis is not complete, and certain aspects of verification are manual. FlowLog has not been evaluated for realistic network policies or for large networks. It requires storing multiple database entries for each network state variable and handles certain aspects of control logic by sending data packets to the controller, so it is unlikely to scale. VeriCon [4] verifies that a program written in its language (CSDN) is correct for all topology and packet events (*e.g.*, packet arrivals, switch joins). It does not handle arbitrary network events, and there is no OpenFlow-based implementation, so its practicality is unclear.

Formal verification of data-plane behavior.. Recent work in network verification has focused on verifying static properties of the data-plane state [45, 91, 95]. Anteater [68] and HSA [43, 44] can verify properties of a static snapshot of a network's data-plane state. These systems can determine whether a static snapshot of data-plane state violates some invariant, but they do not verify the logic of the control program that generated the state in the first place, making it difficult to identify which aspect of the network's control-plane logic caused the incorrect data-plane state. In contrast, Kinetic helps operators verify control logic, such as "if an intrusion detection system determines that a host is infected, the host's traffic should be dropped". This capability helps operators both reason about future data-plane states that a control program could install and troubleshoot incorrect behavior

when it does arise. Because Kinetic's verifies the static programs themselves, it can detect logic errors before the control program is ever run on a live network. NICE [9] can test control-plane properties that might result from arbitrary sequences of standard OpenFlow events; it is not a controller, but rather a test harness for control programs written in existing low-level controllers (*e.g.*, NOX) and hence does not permit reasoning about arbitrary events.

Other SDN control languages.. Many languages raise the level of abstraction for writing network control programs, yet these languages do not offer constructs for concisely encoding policies that capture network dynamics, nor do they incorporate formal verification of control-plane behavior. FML [33] allows network operators to write and maintain policies in a declarative style. Nettle [100] is a domain specific language in Haskell. Procera [101] applies functional reactive programming to help operators express policies. Frenetic [27] is a family of languages that share fundamental constructs and techniques for efficient compilation to OpenFlow switches.

4.7 Conclusion

One of the reasons that network configuration is so challenging is that network conditions are continually changing, and network operators must adapt the network configuration whenever these conditions change. Network operators need means not only to automate these configuration changes but also to verify that the changes will be correct. Existing general-purpose SDN controllers lack intuitive constructs for expressing dynamic policy and ways to efficiently verify that the control programs conform to expected behavior.

To address these problems, we designed and developed Kinetic, a domain specific language and SDN controller for implementing dynamic network policies in a concise, verifiable language. Kinetic exposes a language that allows operators to express network policy in an intuitive language that maps directly to a CTL-based model checker. We evaluated Kinetic's usability and performance through both a large-scale user study and trace-driven

performance evaluation on realistic policies and found that network operators find Kinetic easy to use for expressing dynamic policies and that Kinetic can scale to a large number of policies, hosts, and network events.

Kinetic sits squarely in the realm of ongoing work on network verification and complements the growing body of work on data-plane verification, such as Veriflow [45] and NetPlumber [43]. As these tools can help network operators ask questions about snapshots of data-plane state, and Kinetic can help network operators reason about the dynamics of network policies (which ultimately compile to the corresponding data-plane state), the approaches are complementary. Similarly, Kinetic needs the path guarantees that consistent updates [91] provide to guarantee that the properties it verifies are preserved during state transitions; conversely, consistent updates could be extended to reason about temporal properties such as those that Kinetic can express. One natural next step would be to combine these approaches.

4.8 Acknowledgements

We thank the NSDI reviewers and our shepherd Dejan Kostic for providing valuable feedback for this paper. We also thank Jennifer Rexford, Nate Foster, David Walker, and Yoshio Turner for their helpful comments on earlier versions of this paper. This work was supported by NSF Awards CNS-1261357 and CNS-1409076.

CHAPTER V

CORONET: TRANSPARENT FAILURE RECOVERY SERVICE

5.1 Introduction

Software-Defined Networking (SDN) separates network control and data planes; a logically centralized controller makes traffic forwarding decisions and programs switches to carry out traffic forwarding in the data plane. SDN enables global network visibility, programmability, and extensibility. For example, both Google and Microsoft have deployed SDN in production to manage traffic flows between data-centers [34, 40].

Until recently, fault resilience has been an overlooked challenge for SDN, despite the high business costs of unplanned network component failures that cause service outages. There are three fault domains in SDN-based networks: (1) *Data-plane failure*, where a switch or link fails, (2) *control channel failure*, where the connection between the controller and switch fails, and (3) *controller failure*, where the controller machine or software fails. It is often acceptable to implement the SDN control channel using low-performance legacy network technologies,with legacy protocols such as Ethernet Spanning Tree Protocol (STP) providing control channel resilience. Recent work has explored more general approaches for providing a reliable control channel and controller [1]. Controller reliability can also be improved using well-understood distributed systems techniques for consistent maintenance and updates of replicated state [53].

While the SDN controller could emulate legacy distributed protocols like STP to recover from switch and link failures, that approach would fail to take advantage of the global visibility and central control offered by SDN to optimize network performance metrics. Alternatively, each SDN application could custom-implement an efficient fault resilience solution, but this poses a burden impeding SDN adoption.

In this paper, we present Coronet, a practical and general solution for fast *data-plane recovery* in arbitrary topology networks using standard, unmodified OpenFlow 1.3-compliant switches and controllers. Coronet focuses on data plane failures and assumes a reliable controller and control plane. Switch and link failure and recovery events can generally be detected by neighboring switches, which are programmed to notify the controller of detected topology changes.

The key challenge that Coronet addresses is to rapidly reconfigure routing rules in the data plane to quickly restore connectivity to all flows affected by switch or link failures. As we demonstrate through benchmarking state-of-the-art OpenFlow 1.3 capable switches, slow rule updates in modern switches can readily lead to recovery times on the order of minutes in large networks. Coronet achieves one-to-two-second failover in large networks, while supporting multi-path routing and transparent resiliency for SDN applications.

Coronet satisfies several important design goals in addition to fast failover in large networks. Specifically, Coronet supports *multipath routing* to maximize network throughput by exploiting alternative paths, with a routing policy that avoids congestion and unnecessary use of overly long paths. Coronet supports *arbitrary topologies*, unlike some prior work that requires special topologies [66], and it provides maximal connectivity even under network partition failures. Coronet is a practical solution that works with *existing protocols and platforms*, without need for hardware modification or new standards. Coronet achieves *graceful failover to backup paths* by taking into account traffic conditions and distributing affected traffic to backup paths to minimize introducing congestion. Finally, Coronet can act as a *general resilience service* that transparently restores connectivity when data-plane failures occur, on behalf of one or more higher-level SDN applications.

The key idea behind Coronet is to map a large number of flows onto a smaller number of routes, and then change only the routes when failures are detected. For this, Coronet makes use of an OpenFlow 1.3 feature called *group tables*, which provides a level of indirection during the forwarding action lookup at each switch. Updating only the group table entries

enables very fast recovery times compared to reconfiguring each individual affected flow. Coronet proactively plans and sets up primary routes and sufficient backup routes to rapidly recover from the failure of any single component. While OpenFlow group tables support a feature for fast failover, the failover action is limited to the switches that are immediately incident to a failed component, and thus this feature can implement only *local* traffic re-routing. In contrast, Coronet plans backups over the *network-wide* scope, changing traffic routes multiple hops away from the failed component, to optimize *global* network throughput and load balancing. While Coronet plans for single failure events, multiple concurrent failures may also occur. In some of these scenarios Coronet must reactively compute and install new routes, possibly a slower procedure than failing over to precomputed backups. Our evaluation focuses on single failures and largely leaves multiple failures for future work.

Coronet achieves efficient resource utilization in addition to fast failover. Coronet uses group table entries at ingress switches to represent routes. To reduce pressure on group table usage, and also to speed up failover by minimizing the number of group table entries that need to be updated after a failure is detected, Coronet can optionally represent routes as trees in the network instead of individual paths. In addition, Coronet represents each tree with a small number of group table entries determined using an algorithm that combines traffic to multiple egress switches while taking into account all of their possible failover requirements. To reduce pressure on link bandwidth usage in the network, Coronet dynamically assigns new flows to lower utilization paths, and upon failover distributes affected traffic across multiple backup traffic trees, using a linear optimizer to minimize worst case congestion.

We implemented Coronet on a production SDN controller, HP VAN, and evaluated its operation when controlling four Mininet-emulated network topologies. Our experimental results verify that Coronet achieves fast failover. Coronet achieves failover with congestion

avoidance in 50ms–250ms with small networks with less than 25 switches, and 500ms–1500ms for large networks with more than 100 switches. Without congestion avoidance, Coronet achieves subsecond recovery for large networks as well. Further, the results show that representing routes as trees instead of as paths significantly saves group table usage while achieving similar performance in terms of link bandwidth usage, both before and after failures.

5.2 Failure Recovery in SDNs

We present an example which shows that in a large network, a straightforward approach to recovering from a single failure requires a large number of flow table updates in the bottleneck switch. Based on the switch update rate benchmark(§5.4.2), such numerous updates lead to unacceptably long recovery times.

Though Coronet works with arbitrary topology, we use a more structured data center network example to demonstrate a general problem of failure recovery in SDNs, especially with large networks. Figure 33 shows a two-layer fat-tree topology with four spine switches (top switches in the figure) and eight leaf switches (bottom switches). A leaf switch has four 10G port towards the spine switches, and 32 1G ports connected to end hosts. Each spine switch has eight 10G ports. This topology allows $32 \times 8 = 256$ physical hosts, representing a fairly large data center network.

Assume the hosts exchange uniform all-to-all traffic. While *flows* can be defined differently and have a wide range of granularity, *e.g.*, per-TCP connection, subnet-to-subnet, etc., if we define each *flow* to be the unidirectional traffic from one host to another, the number of flows originating from one leaf switch to other seven leaf switches is $32 \times (7 \times 32) = 7168$ if we assume all hosts are active. Considering only the traffic that traverses the spine switches, and assuming perfect load balancing, each spine switch will have $7168/4 = 1792$ incoming flows from one leaf switch through a single link.

If the link between *s1–s2* fails, as shown in Figure 33, 1792 upward flows and 1792

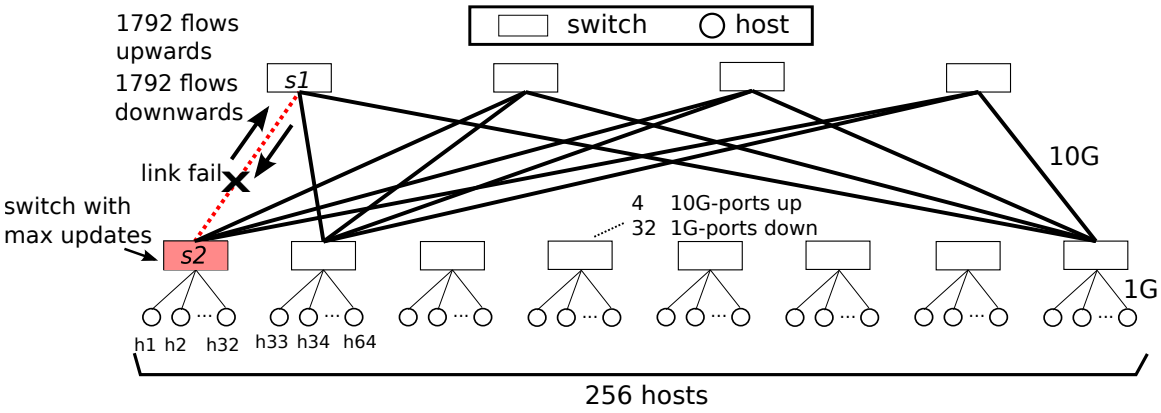


Figure 33: Two-layer fat-tree topology with four spine switches and eight leaf switches. Each leaf switch has four 10G port and 32 1G ports. Some links are omitted for better visualization.

Table 13: Number of modifications and time required to recover from link *s1*–*s2* failure.

Failover mechanism	**Number of flow table entries that needs modification**			**Recovery time**
	Leaf switches	Swith with most updates	Spine switches	
Naïve	3,584	1,792	0 w/ backups (1792 if not)	32,581 ms
Coronet	18	3	0	54 ms

downward flows are affected by the failure. Table 13 summarizes the number of modifications and time required to recover from a link failure. To recover, the controller needs to perform 1792 flow rule modifications in switch *s2* to forward the affected upward flows to a different spine switch. The 1792 downward flows originating from other leaf switches need to be modified as well. Therefore, in total 3584 flow entry modifications are necessary across all leaf switches. We assume the selected alternate spine switch was pre-installed with backup flow table entries for alternate paths.

Assuming that the controller can issue flow modifications in parallel to the switches, the recovery time is dominated by switch *s2* which requires the largest number of modifications (1792). Flow modification benchmarks reported by previous studies [37, 40, 63], as well as our benchmark in §5.4.2, show that modern switches have slow rule updates leading to long recovery times (*e.g.*, 55 rules/second update rate in our benchmark). Compared

Table 14: Definitions.

Term	Definition
Route tree	A set of switches and links that forms a tree structure, used for forwarding packets in the network
Host-attached switches	Switches that have one or more host attached
Ingress switch	The first host-attached switch a packet meets when it enters the network after it is sent from the source host
Egress switch	The last host-attached switch a packet meets in the network, responsible of forwarding it to its final destination host
Intermediate switches	Switches that a packet visits while forwarded in a route tree, which are not ingress or egress switches for that packet
Network components	Links and switches in the data plane
Shared components	Overlapping network components used by the given set of route trees

to the naïve solution, Coronet can recover the connectivity with a mere 18 modifications, 224 times less than 3584, with at most three modification per leaf switch, reducing the link failure recovery time from over 30 seconds to 54 milliseconds with a 55 rules/second update rate.

Using the same topology, the recovery time gets worse with the naïve approach as we increase the number of ports on leaf switches. For 64-port and 128-port switches, the recovery time increases to 130 seconds (>two minutes) and 521 seconds (>eight minutes), respectively. Such sensitivity to port count is particularly troublesome given the general trend in data center switches toward higher port counts. With Coronet, the recovery time does not increase as Coronet does not modify the flow table entries for individual flows to restore connectivity.

5.3 Design of Coronet

We describe Coronet using terminology listed in Table 14. Figure 34 shows the Coronet architecture. At start-up, Coronet's route-tree generation component computes a set of route trees within the physical topology, and pre-installs switch rules to forward packets along the route trees. In the meantime, the virtualization map component delivers the one-big-switch

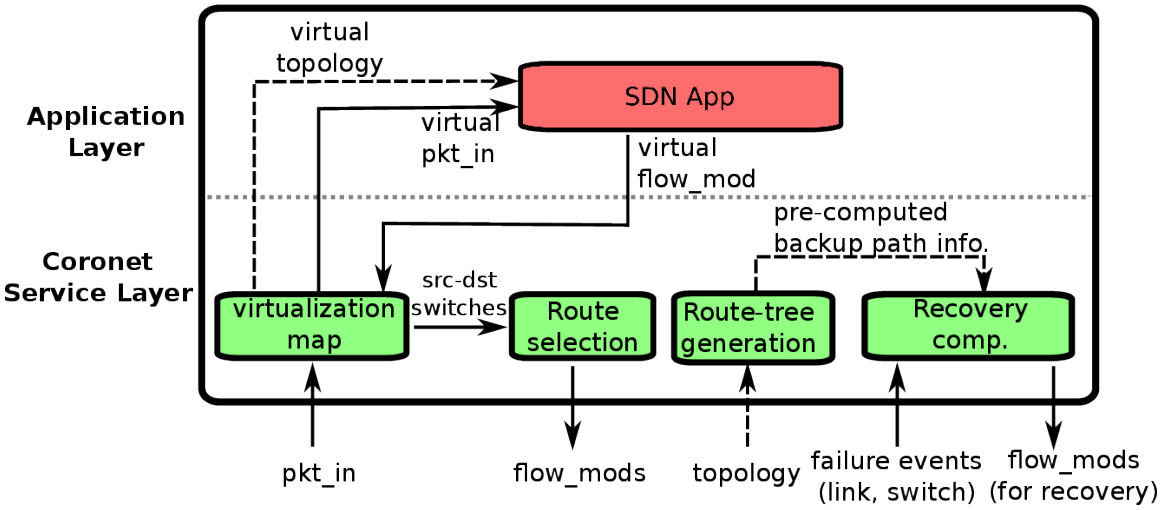


Figure 34: Coronet architecture in the control plane. Dotted lines indicate interactions that are done at start-up. Normal lines indicate real-time interactions.

abstraction to the SDN control application. At runtime, the virtualization map component intercepts packet-ins and translates them for the virtual topology and pass it onto the SDN control application above. When the SDN control application makes a forwarding decision, the message is eventually passed to Coronet's route selection component, which maps a forwarding rule to a route tree.

When a network component failure is detected, Coronet adjusts only the affected route trees rather than the much greater number of affected flows, which greatly reduces the number of rule table updates that the controller needs to issue, leading to a large reduction in recovery time. This is done by only updating the group table rules at ingress switches to specify a different tag (switching the flows to a different tree).

Furthermore, to minimize the chance of overwhelming a link over its capacity, Coronet recovery component formulates the distribution of traffic to backup paths as a linear programming problem, and solves it with a well-known LP solver, GLPK [30], which goal is to minimize the maximum load on links after traffic is redistributed on backup paths. We present the design details of Coronet in the following subsections.

5.3.1 Planning for Failures

Coronet attempts to compute and create multiple primary route trees that are as resource-disjoint as possible in terms of shared switches and links, to exploit path diversity for load balancing and failure resilience. Coronet also computes sufficient backup route trees to guarantee restoring connectivity despite any single component failure. A primary tree can also serve as backup for other primary trees.

In this section, we first describe how primary route trees are generated, and then explain how backup route trees are created and paired with each primary.

5.3.1.1 Route tree properties

Coronet builds each route tree to span all host-attached switches. Thus, a route tree can forward traffic between any host pair, along a single path which may or may not be a shortest hop path. The route tree generation component create route trees that avoid resource sharing, such as links and switches, between trees. Such resource *disjointness* between route trees helps Coronet to utilize redundant paths, and it enables many trees to serve dual roles – primary routes and backup routes for other primary routes.

Figure 35 shows the pseudocode for generating a set of primary route trees in Coronet. Note there are two parts: (1) creating the first set of resource-disjoint route trees, and (2) improving the set by adding route trees that have shorter hop paths for each host-attached switch pair.

Resource-disjoint route trees. First, the controller reports to Coronet the network topology, discovered using the Link Layer Discovery Protocol (LLDP), and a list of host-attached switches. Coronet then computes the route trees as a series of Minimum Spanning Trees (MST) using the well-known Kruskal algorithm [56], which has algorithmic complexity $O(|E|\ log|V|)$, where $|E|$ is the number of links and $|V|$ is the number of switches. Coronet prunes each spanning tree by removing all branches that are devoid of host-attached switches, yielding a tree that spans and connects all host-attached switches.

As shown in the first part of Figure 35 (line 1–15), Coronet computes route trees one by one, and adds a positive value w to the weight of links used by each tree as it is generated, thereby discouraging subsequently generated trees from selecting the same resources as previously generated trees. Figure 36 shows the procedure graphically. Route trees are generated until all links in the network are covered, thus enabling network-wide link utilization with a concise set of route trees. We refer to these as *primary route trees*, which are used to forward flows before failure.

The weight value w plays an important role in route tree construction. We want to encourage using alternative paths, so w has to be big enough to shift selection to different resources for subsequent trees. But it should not be too big because it would encourage paths with excessively long hop count which should be avoided. Coronet picks a different w value for different topologies: the value is equal to *w=1/(the average number of hops between all host-attached switch pairs)* in the topology. Even though simple, this approach works well for all topologies we have evaluated.

Improving route trees. Route trees created previously do not guarantee to have bounds in terms of path stretch. A route tree can provide very long paths for some host-attached switch pairs. The second part (line 16–26) in Figure 35 adds primary route trees to ensure that the length of the route between every host-attached switch pair is within the bound $\lceil minHop \times m \rceil$.

The m value decides the path stretch of using route trees in comparison to shortest paths. However, larger m value can create more route trees, which in turn affects the recovery time later when a failure occurs. Thus, the m value creates a trade off. There is no silver bullet m value that works for every situation, and the value should depend on topology, traffic forwarding requirements (*e.g.*, latency bound), and more. For this paper, we have fixed $m = 2$, thus bounding route trees to provide no longer than twice the shortest paths available in the topology. Future work should explore how to decide an appropriate m value for different situation.

```
1  // 1. Create initial set of route trees
2  R={}
3  while (∃ uncovered link):
4    r_i=BuildTreeWithKruskalAlg(G);
5    TrimTree(r_i);
6    R.add(r_i);
7
8    for each link l in r_i:
9      increment weight by w;
10
11   for each switch s in r_i:
12     for each link l attached to s:
13       increment weight by w;
14 ResetWeights(G)
15
16 // 2. Improve set
17 for each (src,dst) host-attached switch pair:
18   path = ShortestPath(G,src,dst);
19   minHop = path.length();
20
21   r = SelectShortestRouteTree(R,src,dst);
22   hopCount = NumOfHops(r,src,dst);
23
24   if (hopCount > ⌈minHop * m⌉):
25     r' = BuildTreeWithPath(G,path);
26     R.add(r');
```

Figure 35: Pseudocode for creating a set of primary route trees.

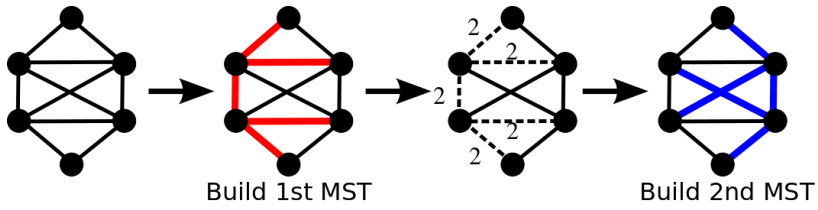


Figure 36: Constructing resource-disjoint route trees (numbers shown on links represent weights).

5.3.1.2 Backup route tree computation

Figure 37 shows the algorithm for backup route tree computation. For each primary route tree in $r_i \in R$, the algorithm first searches for a backup route tree r_j that shares the least number of components (e.g., links and switches) with r_i, and it is added to the primary r_i's backup set (line 2–4). If the backup is completely disjoint with the primary, such backup tree will restore connectivity for any component failure in the primary, thus the backup route selection process for r_i completes. If the primary and the backup is not completely disjoint, the algorithm tries to find a second backup r_k that does not contain the shared component of r_i and r_j (line 7–11). Such r_k survives failures that r_i and r_j do not survive,

```
1  NewR={} // newly created routes
2  for each r_i ∈ R,do:
3    ArgMin(|SharedComp(r_i,r_j)|,r_j),r_j ∈ (R∪newR)−{r_i}
4    B={r_j};
5    if SharedComp(r_i,r_j) = φ:
6      continue; // r_i and r_j are completely disjoint.
7    else:
8      Find all r_k ∈ R∪newR−{r_i,r_j} that does not contain SharedComp(r_i,r_j).
9      if ∃r_k:
10       ArgMin(|SharedComp(r_i,r_k)|,r_k);
11       B=B∪{r_k};
12     else:
13       AddWeight(G,r_i,w);
14       r_k=BuildTreeWithKruskal(G−SharedComp(r_i,r_j));
15       if r_k spans all host−attached switches:
16         TrimTree(r_k);
17         B=B∪{r_k}; newR=newR∪{r_k};
18       else:
19         // Remove shared components one by one.
20         for each c_i ∈ SharedComp(r_i,r_j), do:
21           r_k=BuildTreeWithKruskal(G−c_i);
22           if r_k spans all host−attached switches:
23             TrimTree(r_k);
24             B=B∪{r_k}; newR=newR∪{r_k};
25           else:
26             if c_i is a link:
27               r_i still works in each subgraph;
28             else if c_i is a switch:
29               r_k=BuildTreeWithKruskal(G−c_i);
30               TrimTree(r_k); newR=newR∪{r_k};
31               curVertexFailMap.put(c_i, r_k);
32
33   // r_i's backup set is complete
34   r_i.backupSet = B
35 }
```

Figure 37: Pseudocode for finding 1-failure resilient backup route trees for each primary route tree.

specially for the components shared by r_i and r_j. If such r_k cannot be found in the pre-computed set, the algorithm creates a new route tree without the shared components of r_i and r_j (line 12–17). If the new route tree r_k does not span all host-attached switches, that means the removal of all shared components of r_i and r_j disconnected the topology. In such case, Coronet removes each shared component one by one, and creates route tree for each removed component (line 18–24). If even removing one component disconnects the topology, it means it is a cut-edge of cut-vertex (line 25–31), which we discuss in §5.3.1.3.

After Figure 37's algorithm, Coronet runs another routine that checks the number of live backups after each component failure for each primary route tree. It constructs more backup route trees if the number is less than a pre-defined parameter, which is the minimum

number of backups guaranteed to be alive after a failure. This prevents Coronet from having too few choices when traffic needs to be distributed on backups, which can lead to excessive load on certain links (details in § 5.3.3). The value introduces a trade-off: bigger values will intuitively bring better load distribution on links and switches after a failure, but the computation time for finding the optimal load distribution will increase. Smaller values will have an opposite effect. We will highlight the trade-off in our evaluation section.

5.3.1.3 Network partition failures

In graph theory, a cut edge, or a *bridge* edge, is an edge that disconnects the graph into two or more subgraphs when removed. When such an edge fails in a topology (graph), all route trees generated by Coronet will be affected. In this case, the original primary route tree provides the best connectivity possible without any modification necessary[1]. This means the original tree will still span all vertices within each island when the graph is disconnected.

A vertex failure can also disconnect the topology, affecting all precomputed route tress. In graph theory, such vertices are known as *cut vertex* or *articulation points*. A vertex failure can remove multiple edges in the graph, thus the original primary route tree might not guarantee best connectivity within each island, which makes the situation different with the cut edge failure scenario above.

To immediately recover from such failures, Coronet prepares for cut vertex failures *in advance* by first identifying nodes that will disconnect the graph when removed. One or more backup spanning trees are precomputed and mapped to each identified cut vertex. Thus when a cut vertex failure is detected, Coronet can immediately return the relevant set of backup route trees that provides best connectivity in face of such a failure. The preparation of such failures is in line 28–31 in Figure 37. `curVertexFailMap` is a hashmap that maps a cut vertex to a backup route tree that can be used to provide best

[1] We omit proof for the limited space.

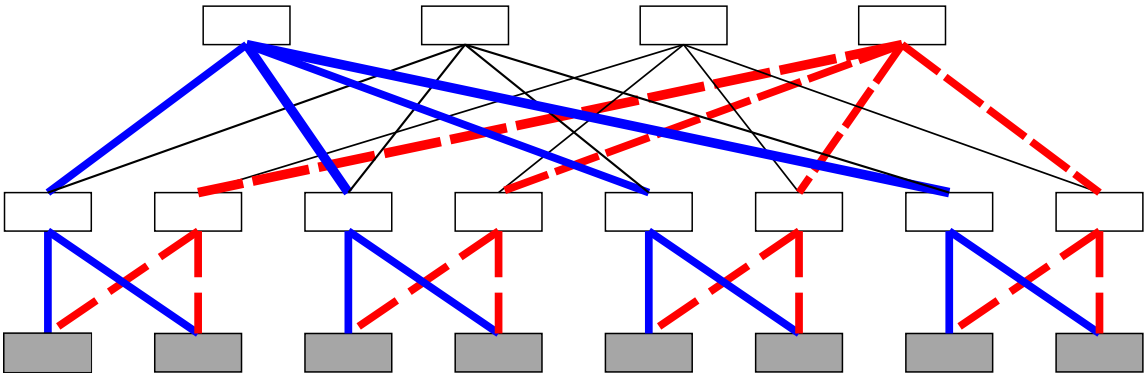


Figure 38: Two different route trees in a three-layer fat-tree topology, one shown in blue solid line, the other one shown in red dotted line. Grey boxes on the bottom layer are host-attached switches.

connectivity when the vertex (switch) fails. There exists efficient algorithms for finding such set including work from Hopcroft and Tarjan [35], and Schmidt *et al.* [94]. Coronet can also leverage such algorithms to identify such articulation points faster.

5.3.2 Traffic Forwarding

In this section, we describe how Coronet represents route trees in switches and how it forwards flows using route trees.

5.3.2.1 Representing route trees

Figure 38 shows an example of two route trees in a three-layer fat-tree topology. Whenever a source host initiates a new flow, Coronet's route selection component chooses an appropriate route tree to forward the flow. It favors a shorter path unless that path is experiencing high traffic volume, in which case another less loaded longer path is selected.

To implement route tree forwarding, Coronet assigns a unique label value to each route tree, and tags an incoming packet with the appropriate label to indicate the chosen route tree to forward the packet. Coronet programs the ingress switch to tag every packet of an incoming flow with the label corresponding to the route tree chosen for the flow. Coronet also programs the last hop egress switch to pop the label before forwarding the packet to the destination host.

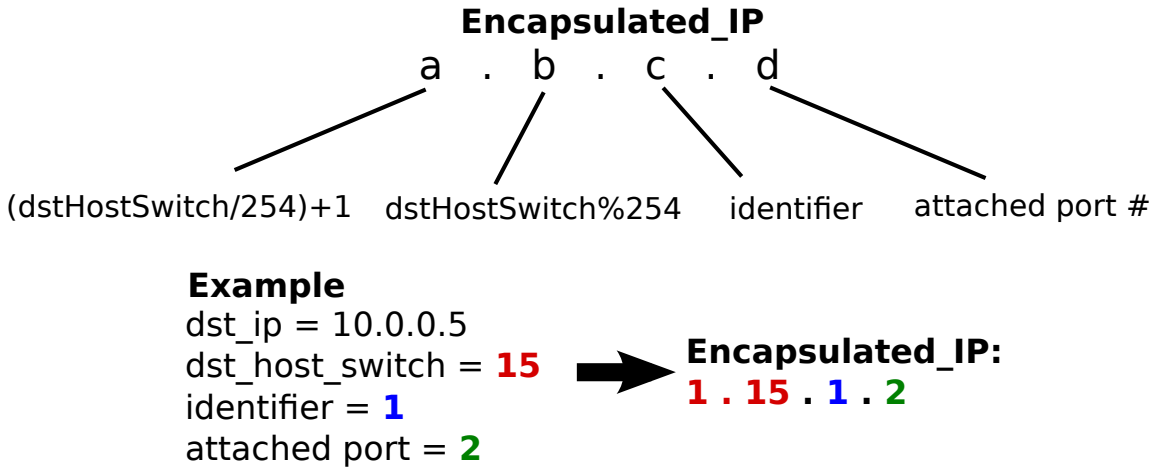


Figure 39: Encapsulation mechanism in Coronet. A destination host-attached switch can be identified with the first two positions, a and b. Thus, the flow table entry can match on $a.b.c.d$/16 to forward flows to the destination switch. The identifier can be used to differentiate different VMs attached to the same port. The port indicates the port number where the destination host is attached to on the destination host-attached switch.

The hop-by-hop route taken by a packet depends on its route tree and its destination address. To reduce the number of forwarding rules needed in intermediate switches, Coronet encapsulates each packet at the ingress switch with an IP header that identifies the destination egress switch. As Coronet has a global view of the network, it can dynamically track which switch each host or VM is currently attached to.

For each flow, Coronet first installs a rule at the ingress switch to encapsulate the flow's packets with an IP header destination address based on the egress switch and egress port as shown in Figure 39. Coronet installs another rule action to tag, or "push", a label identifying the chosen route tree for the flow. At all intermediate switches, Coronet installs rules to forward packets on each route tree to the corresponding egress switch, instead of to the much larger number of destination hosts as would be needed without encapsulation. Finally, Coronet installs a rule at the egress switch to decapsulate the destination IP address, and another rule action to "pop" the label and forward the packet to its destination host. Coronet currently uses the IP header for encapsulation and decapsulation, and the VLAN header for label tagging, but other fields can be used for the same purpose.

5.3.2.2 *Group table usage*

The OpenFlow specification v1.1.0 [84] introduced an OpenFlow table type called Group Table. An OpenFlow switch can have one group table. The group table provides a level of indirection, in which the flow table entry that matches a packet can point to a group table entry which is subsequently accessed to complete the lookup of the packet's forwarding action. One or more flow table entries can point to a single group table entry (many-to-one mapping). Thus, a single update to a shared group table entry can affect multiple flows.

Reducing table updates for recovery. Using both flow table and group table entries enables Coronet to recover from failures with far fewer control-plane messages than using flow tables alone. Figure 40 illustrates how flow table and group table entries are used in host-attached switches (depicted as grey boxes). First, the ingress switch for a flow has flow table entries that encapsulate the destination IP address and direct the flow to a group table entry based on group ID. In an unoptimized approach, the group ID is unique for each (route tree, egress switch) pair (we describe how to optimize and reduce group table entries below). In Figure 40, multiple flows are directed to a group table entry. Each group table entry tags a flow with a label (to identify the route tree) and forwards it to a specified output port. When a failure disconnects a route tree, Coronet modifies the actions of affected group table entries to tag flows with different labels representing live backup route trees, and modifies the output port too if necessary.

Reducing the number of group table entries. In the approach just described, the number of group table entries used at each host-attached switch is $H \times N$, for a network with H host-attached switches and N route trees. In a large network this may exhaust group table capacity. It also requires a large number of entries to be modified after a failure, leading to slow recovery time.

To reduce the number of group table entries needed, we observe that flows that always have the same forwarding action can be grouped together. In particular, flows that share

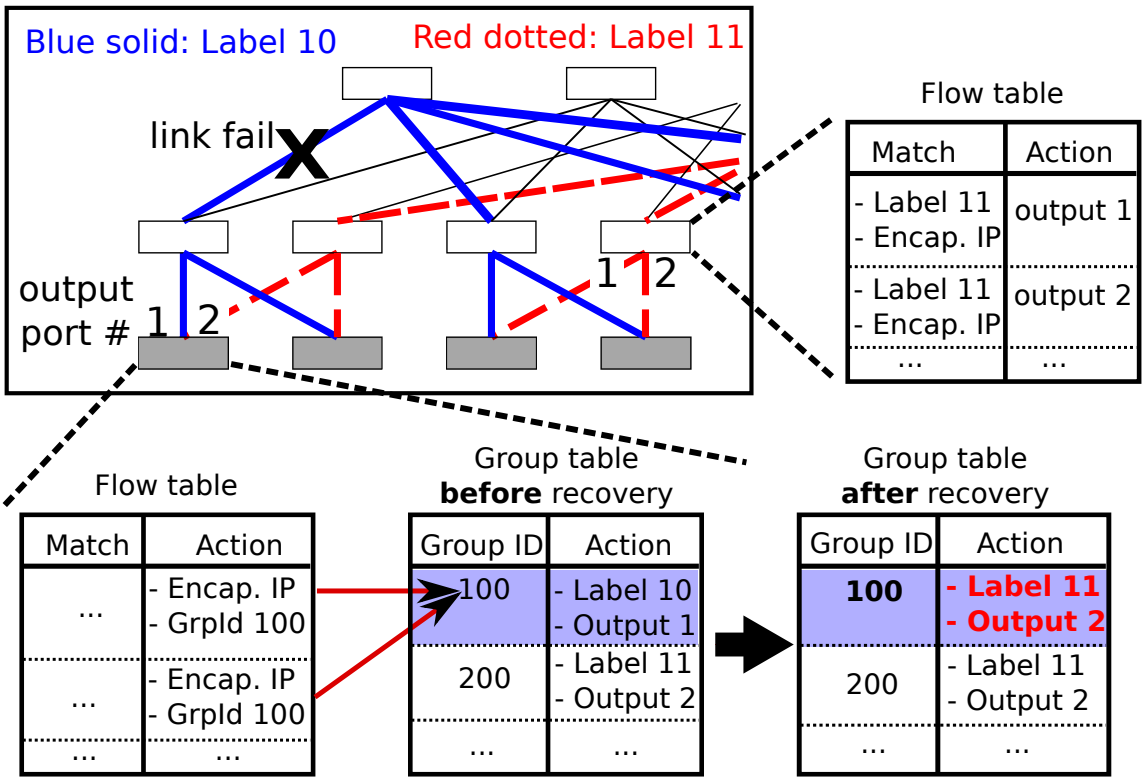


Figure 40: Flow table and group table in Coronet. Group ID numbers are unique for each group table entry. The numbering does not need to be sequential or start from 1.

the same route tree, i.e., tagged with the same label, and forwarded to the same output port, both *before* failure and *after* failure, can share a group ID. Figure 41 shows an example of forwarding rules at a host-attached ingress switch for traffic to different host-attached egress switches. In this example, a primary tree has two backup trees, and there are nine egress host-attached switches (A to I). The table on the left shows the forwarding action for each tree and egress switch. Without group table reduction, the ingress switch would require nine entries for the primary tree. As shown in the table on the right, the group table usage is reduced to five entries by grouping traffic to different egress switches that share the same forwarding action.

Figure 42 shows the pseudocode for reducing group table entries described above. Given a primary route tree r and its backup route tree set B_r, for each host-attached switch v in r, a set of other host-attached switches, named $NewDstSet$, that are reachable through

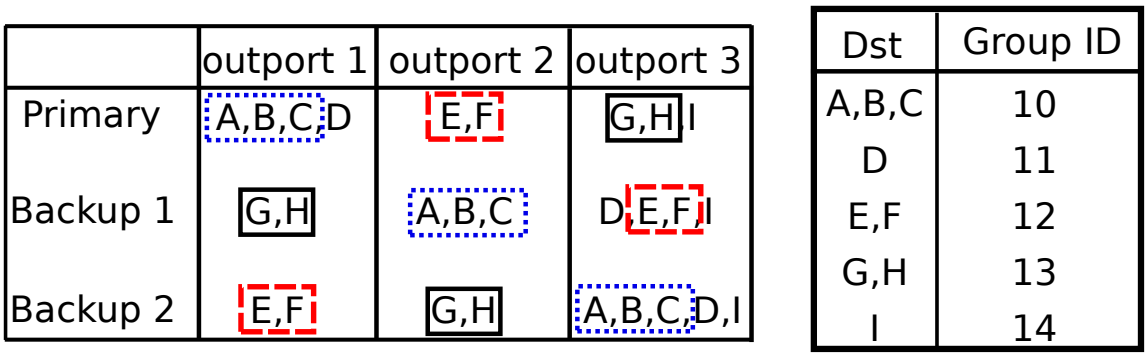


Figure 41: Reducing the number of group table entries by bundling destination switches (in alphabet) that are reachable by the same output port in primary and backup route trees. Bundles are identified by rectangular boxes around them.

each output port p of v is computed (line 1–4). Then the $FindNewGroup$ method finds if the whole set or a subset of $NewDstSet$ can be reached by using some output port in r's backup paths. $FindNewGroup$ method calls itself recursively until $NewDstSet$ becomes empty (line 8–10), which eventually happens after all host-attached destinations are considered. The $NewDstSet$ is first preserved (line 12–13). Next, the destinations in $NewDstSet$ are tested whether the whole set or the subset of them are reachable through a particular output port using each of backup route trees (line 15–21). If a group of destination host-attached switches are found (or ends up having a single destination host-attached switch), a new group table ID is assigned to the group (line 24), and then the method is called recursively with the remaining un-dealt set of destinations (linke 25–26).

5.3.3 Avoiding Congestion in Failure Recovery

When a data-plane failure is detected, Coronet looks up and implements the pre-computed action to restore connectivity to affected routes. For each affected route, Coronet only updates the corresponding group table entries at ingress switches to convert them to use one or more backup trees. Compared to adjusting every affected flow, adjusting only the tagging and tree routes leads to a tremendous reduction in the number of updates. When multiple backups are available for an affected primary route, Coronet computes appropriate weights for redistributing the traffic from the affected route to the backups in order to minimize

```
1  Minimization(r,B_r):
2    for each v ∈ {r's host-attached switches}, do:
3      for each p ∈ outputPorts(v), do:
4        NewDstSet=GetReachableDst(p,r)
5        FindNewGroup(NewDstSet,B_r,v)
6
7  FindNewGroup(NewDstSet,B_r,v):
8    // If NewDstSet becomes null, stop.
9    if NewDstSet=φ,
10     return
11
12   // Maintain original set
13   OrgSet=NewDstSet
14
15   // Start searching in backup's links
16   for each p ∈ outputPorts(v),do:
17     for each b ∈ B_r,do:
18       // Find reachable DstSet for this p and b.
19       Tmp=NewDstSet ∩ GetReachableDst(p,b)
20       if Tmp ≠ φ,
21         NewDstSet=NewDstSet ∩ GetReachableDst(p,b)
22
23   // Assign new group ID
24   assignNewGroupId(NewDstSet)
25   // Deal with remaining dst set
26   FindNewGroup(OrgSet-NewDstSet,B_r,v)
```

Figure 42: Pseudocode for reducing the number of group table entries.

network congestion introduced by the failover.

The goal of selecting backup route weights to minimize congestion is formulated as the linear programming (LP) problem shown in Figure 43, using the definitions listed in Table 15. The LP objective is to minimize z, the maximum load among all links in the topology after traffic is distributed to the backup paths for every primary path that is disconnected by the failure. Constraint (1) states that z is the upper bound of loads on all links in the network. The load of a link after failure is determined by multiplying three values: the weight (in ratio) given to a backup (w), the traffic amount of the original flow (T), and a binary value that indicates whether the link is included in a backup route (L). Constraint (2) states that for each (primary path, ingress switch) pair, the addition of the weights of all backup paths will sum to `1`. Constraint (3) presents the case where different egress switches are reachable by the same group table entry, due to the group minimization process described in § 5.3.2 and Figure 41.

The model above is expressed and solved using the GNU Linear Programming Kit

Table 15: Definitions used in the LP model.

Term	Definition
z	Maximum load on a link.
E	Set of all links in the topology.
$Load_e$	Traffic amount at link e.
R	Set of route trees computed so far.
B_r	Set of backup route trees for a primary route tree *r*.
H	Set of all host-attached switches.
$\vec{T}_{r,s}$	|H|x1 vector. Traffic amount from s to each destination host-attached switch that uses route tree r.
$L_{e,s,r}$	|H|x1 binary vector. 1 if link *e* is included in the path between s and list of destination host-attached switches that uses route tree *r*. 0 if not.
$\vec{w}_{r,s,b}$	|H|x1 vector. Weight setting distribution among backups in B_r for each host-attached switches for backup b at switch s when primary route tree r is used.
getGroupId(r,s,d)	Returns the group entry ID for route tree r, source switch s, and destination switch d.

- Minimize z
- Constraints:
  (1) For every link $e \in E$: $Load_e \leq z$.
  $Load_e = \sum_{r \in R} (\sum_{s \in H} (\sum_{b \in B_r} (\vec{w}_{r,s,b} (\vec{L}^T_{e,s,b} \vec{T}_{r,s}))))$
  (2) For each (r, s) combination, $\sum_{b \in B_r} \vec{w}_{r,s,b} = \vec{1}$,
  (3) For each $(r, b, s, d1, d2)$ combination,
  If getGroupId$(r, s, d1)$==getGroupId$(r, s, d2)$,
  $\vec{w}_{r,s,b}[d1] == \vec{w}_{r,s,b}[d2]$

where $r \in R, b \in B_r, s \in H, d1 \in H, d2 \in H$

Figure 43: Weight adjustment formulated as LP model.

(GLPK) [30]. The number of weight variables can grow as the square of the number of host-attached switches, leading to long execution time in some cases. Thus, we introduce a simplified formulation in Figure 43 that effectively reduces the number of variables by constraining all the group table entries for a single primary tree at an ingress switch to share the same backup tree weight distribution. In §5.4.3 we show that this simplified formulation produces almost identical results as the full LP model but is quicker to solve.

Coronet fails over to multiple backup routes by using a *select* group table entry type introduced since OpenFlow v1.1.0. This feature allows different actions to be performed on each unique flow based on selection *weights*. Specifically, a $select$ type group table

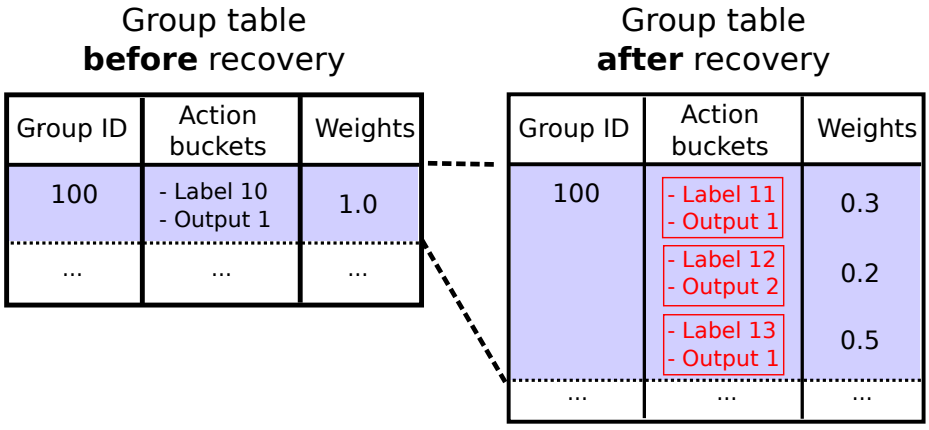


Figure 44: *select* group table entry type used in Coronet. Red boxes in the right group table denote different backup route trees' action buckets. How traffic is redistributed depends on how weights are assigned to each backup action bucket.

entry can have multiple *action buckets*, each specifying its own list of actions. Each flow gets hash mapped to a bucket with probability proportional to an assigned bucket weight. Figure 44 illustrates the use of *select* group table entry and weights in Coronet.

5.3.4 Fault-tolerance as a Service

Network operators or developers build their own SDN applications to solve specific tasks for their networks, yet it is cumbersome to build their own algorithm or method to handle unexpected data-plane failures. Coronet's purpose is to fill this gap, and provide a general "service", specifically a *fault-tolerant connectivity service* that maintains connectivity in face of link and switch failures.

Coronet exports a simplified *virtual topology* to an SDN control application allowing it to implement high-level policies while Coronet transparently handles failure recovery in the real physical topology. As illustrated in Figure 34, Coronet first receives a packet-in message from the data plane, and passes it to the SDN control application. The application can issue OpenFlow flow table modification messages for the virtual topology, which Coronet intercepts and translates into the form that can actually be realized through Coronet's routing mechanism. Coronet has several techniques that help it to be integrated with other SDN applications: (1) physical topology abstraction, (2) packet-in message interception,

and (3) flow-mod message transformation.

Topology abstraction. Previous works have suggested of presenting a "one big switch" abstraction to operators, liberating them from grappling with hop-by-hop forwarding in the physical network [42, 76]. With such topology abstraction, operators can focus on programming the forwarding behaviors between end-hosts rather than meddling with hop-by-hop routing in the network. Following this approach, Coronet retrieves the physical topology and creates a single virtual switch with ports on the perimeter of the physical topology. Coronet maintains the mapping of `(virt_switch, virt_port)` to `(real_switch, real_port)` and vice versa.

Intercepting packet-in messages. Coronet intercepts *packet-in* messages that directly come from the data plane and transforms the original packet-in messages to the virtual ones in accordance to the topology abstraction. The SDN App has an illusion that packet-in messages are coming from the data plane directly. With virtual packet-in messages, the SDN App can built its own logic for handling packet-in events and react to (*e.g.*, forward, block, ignore) as intended.

Transforming flow-mod messages. The SDN App can create OpenFlow messages that modifies (*e.g.*, add, delete, modify) flow table entries in an SDN-enabled switch. However, such flow-mod message from the SDN App is created with the virtual topology and cannot be applied directly on the physical topology. To translate such flow-mod messages to relevant ones, Coronet provides a special API that SDN Apps can leverage to pass its flow-mod message to Coronet, which in turn translates it into a series of flow-mods that actually works on the physical topology.

Coronet's current abstraction only supports one SDN application at a time with one virtual topology. For future work, we plan to support multiple SDN applications simultaneously where each SDN application could have a different view of the topology.

5.4 Implementation and Evaluation

In this section, we first present how we implemented Coronet using HP's VAN SDN controller, and how we evaluated Coronet with various network topologies. We then show the performance of Coronet both before and after data-plane failures.

5.4.1 Implementation

Coronet requires support for OpenFlow v1.1 [84] or higher in switches and the controller to leverage the group tables for routing and fast failover. We built the Coronet controller program (around 13,500 lines of Java code) using the HP VAN SDN controller platform. The HP VAN controller [36] is a production grade platform built with Java, and supports OpenFlow v1.3. The controller platform uses the OSGi (Open Service Gateway initiative) framework, which allows modular software deployment, thus it is possible to load and unload multiple modular SDN apps dynamically. The controller also has extensive REST API support, and allows external scripts or applications to indirectly control the network through the controller.

The HP VAN controller has notable similarities with other SDN controllers such as Floodlight [26] and OpenDaylight [82] in terms of programming language, framework, and controller structure. Therefore, porting Coronet to either platform should be feasible.

The Coronet controller program runs on a machine with an Intel Xeon CPU E5-1620 3.60 GHz processor and 32 GB of memory.

5.4.2 Benchmark of flow table update time

The flow table update time (or rule update time) is the time it takes to modify an entry in a flow table of an OpenFlow-enabled switch. This has been measured by previous studies with commercial OpenFlow-enabled switches [37, 40, 63], showing that the time to update an entry could be substantial; FFC [63] reported that the median time to update 100 rules is around one second (100 rules/second), with the worst case being over 200 seconds (0.5

rules/second) in a controlled environment. We also benchmarked two OpenFlow-enabled switches from two different vendors, using HP VAN controller as our controller platform, which one performed around 35 rules/second and the other around 55 rules/second on average.

Measurement mechanism: Figure 45 shows our experiment setup. We first populate the flow table with hundreds of unique entries, all forwarding packet to a black hole output port (port 3). Then we start a script in host `h1` to continuously send UDP traffic to host `h2`, which will never go through unless the output port of a particular flow entry is modified correctly. To measure time needed to modify N flows, we sequentially send flow modification messages from the controller starting from the first flow to the $(N-1)$th flow that will modify the output port to another black hole output port (port 4). Then controller sends a `barrier request` message to the switch, which forces the switch to complete queued control messages first before processing the next arriving one. Next, the controller sends a flow modification for the Nth flow, corresponding to the UDP traffic flow. The flow modification changes the output port to direct packets to host server, `h2`. The total rule update time is measured as the difference between *the time when the UDP packet arrived at* `h2`, and *the time when the first flow's modification message was sent from the controller*. Hosts `h1` and `h2` and the controller all physically reside on the same well provisioned server and use a common clock for timing.

Figure 46 shows the rule update rate of one of the two OpenFlow switches we benchmarked (we omit the lesser performing one due to space limit). The switch shows a rule update rate around 50–55 rules per second with varying number of flow modifications. Based on this real switch measurement, we extrapolate estimated recovery time for our evaluation by dividing the number of modifications with the update rate, which is 55 rules per second, as we did in Figure 13.

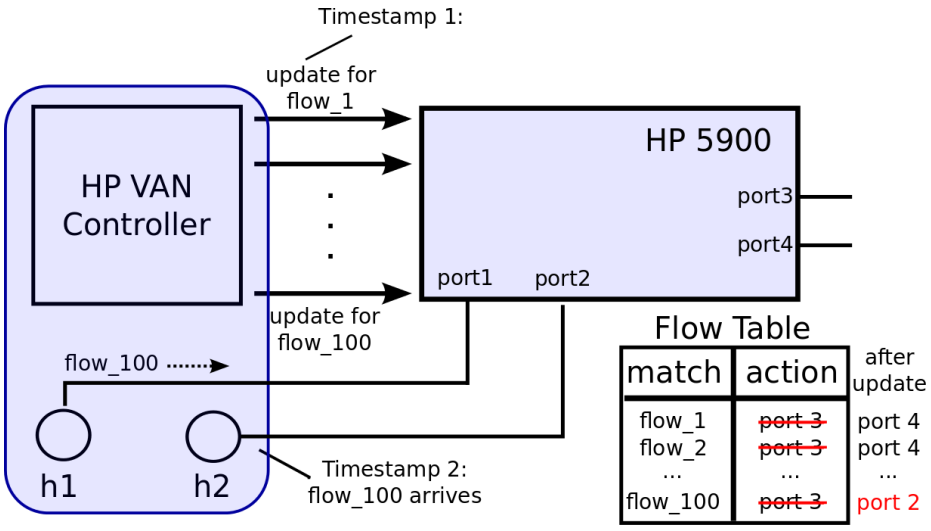


Figure 45: Setup to measure rule update delay of 100 flows.

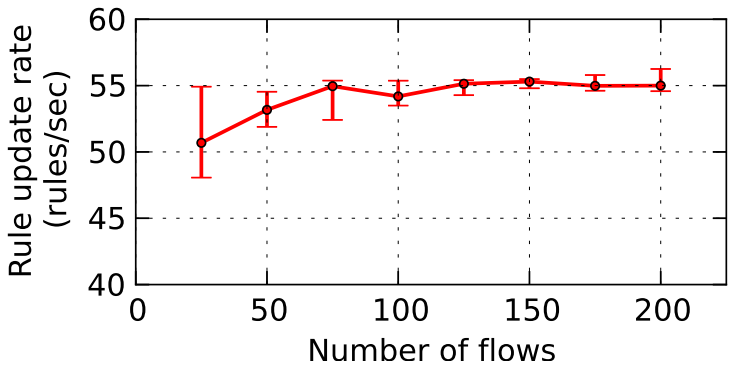


Figure 46: OpenFlow rule update rate in an OpenFlow switch.

5.4.3 Evaluation methodology

Testbed emulation. We evaluated Coronet in a Mininet-emulated testbed with four different topologies. The topologies are summarized in Table 16. The `B4` topology is Google's WAN, which provides connectivity among Google data centers around globe, and it delivers over 90% of internal application traffic [40]. `AT&T` is an ISP backbone network in North America, which is also a real network obtained from the Internet Topology Zoo [99]. Both the `Jellyfish` [96] and `Clos` [2] are not emulated based on a real networks but are widely used topologies in data centers. Mininet-emulated testbeds use Open vSwitch [85] for switches that run OpenFlow v1.3.

Table 16: Testbed topologies.

Size	B4	AT&T	Jellyfish	Clos
Number of switches	12	25	100	125
Number links	19	58	200	500

Traffic matrix. We use network traffic collected from real inter-connected data centers that are grouped by (source cluster, destination cluster, application) tuple, the same traffic traces used in [7]. We estimate the number of virtual machine instances in each cluster by the incoming and outgoing traffic amount. The clusters are mapped to ingress/egress switches in the testbed topology in a way to equally spread the number of virtual machines in each switch and also minimize the amount of inter-cluster traffic, which is a widely used practice for data centers.

Coronet-K-Shortest algorithm. We evaluate Coronet and its tree generation algorithm (now we denote as Coronet-Trees or Coronet interchangeably) by comparing it with a reasonable but more naïve path generation approach called *Coronet-K-Shortest*. Coronet-K-Shortest algorithm creates k number of shortest paths between each host-attached switch pair in the network. The paths are rendered to be disjoint with each other by a similar technique used in Coronet: when one shortest path is created, it adds weights to links used by the path so that the next path chooses different links in the next round, up to k rounds. We fix k=4 for our experiments to make sure it has enough redundancy when recovering from failures. Due to larger and more diverse set of routing paths, we expect Coronet-K-Shortest to perform better in terms of load balancing and avoiding link congestion but with a cost of using more flow and group table space. Except the route generation algorithm, Coronet-K-Shortest uses the same techniques and algorithms as Coronet-Trees: Coronet-K-Shortest uses group tables to reduce the number of modification needed for recovery and the GLPK LP solver to distribute traffic on multiple backups to avoid link congestion.

We first evaluate if Coronet-Trees provides reasonable traffic distribution and link utilization in the network *before* data-plane failures (§ 5.4.4). More specifically, Coronet-Trees should spread out the traffic on the network for load balancing, and also ensure no single link is overly utilized. We then evaluate how fast Coronet-Trees recovers from single link failures in the network (§ 5.4.5), and how well load on affected links are distributed *after* data-plane failures (§ 5.4.6). Switch failure can be regarded as special case of link failures: simultaneous failures of multiple links which Coronet-Trees guarantees to have backups available with any switch failure. We omit the evaluation for switch failures as the evaluation shows similar trend.

Our evaluation shows that Coronet-Trees provides comparable link load distribution to the Coronet-K-Shortest algorithm before and after failures, and the maximum link utilization can even be smaller in some topologies. Coronet is up to three times faster when recovering from failures compared to the Coronet-K-Shortest algorithm except for the `Clos` network. Coronet is able to achieve this with less cost, specifically using less number of group table entries in the switches.

5.4.4 Normal operation before failure

Coronet leverages group tables to perform forwarding during normal operation. However, most switches, including the switches we have benchmarked, have a limit around 100–150 group table entries per switch. As group table entries are scarce resources, it is necessary to keep the number of entries as low as possible. We measure the number of group table entries required in host-attached switches with Coronet-Trees and Coronet-K-Shortest, respectively. Figure 47 shows the median number of group table entries in all host-attached switches in the network for different topologies. Overall, Coronet-K-Shortest requires more group table entries than Coronet. Coronet requires around 10 to 100 group table entries per host-attached switch for various topologies. On the contrary, Coronet-K-Shortest path requires over 40 group table entries for all topologies, and up to 200 for the `Jellyfish` and

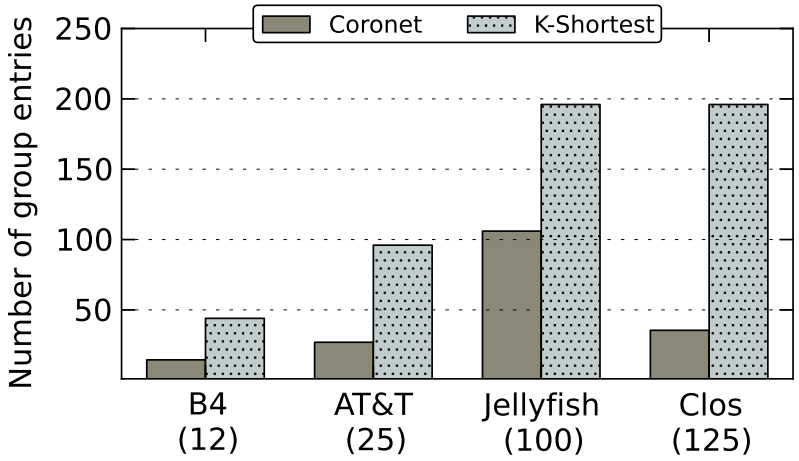
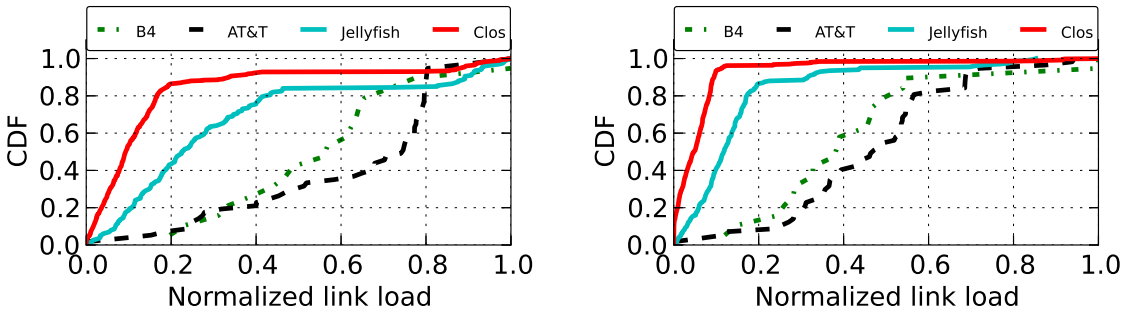


Figure 47: Number of group table entries required by Coronet and Coronet-K-Shortest.

`Clos` topologies. This is because Coronet-K-Shortest approach creates multiple shortest paths for every host-attached pair while Coronet relies on a smaller number of route trees, each of which can route any flow in the network. Coronet is able to further reduce the number of group table entries by incorporating the group minimization algorithm described in Section 5.3.2.2.

To better understand link utilization in the network, Figure 48 shows the distribution of loads on links for each topology before any failure for Coronet and Coronet-K-Shortest. Link loads are normalized against the maximum link load in Coronet for each topology, respectively. As shown in the figure, Coronet has comparable link load distribution to Coronet-K-Shortest, and comparable maximum link load for each topology. We can see that for `Clos` and `Jellyfish` topologies, more than 80% of the links are lightly loaded with only around 20% of the maximum load, for both Coronet and Coronet-K-Shortest; while for `B4` and `AT&T`, the loads on links are more evenly distributed. Overall, Coronet incurs slightly higher link loads than Coronet-K-Shortest, mainly because Coronet-K-Shortest has a larger and more diverse set of forwarding paths, which allows to spread the traffic better than route trees. However, the much smaller number of group table entries required by Coronet-Trees justifies this overhead.



(a) Coronet (b) Coronet-K-Shortest

Figure 48: Link load distribution for different topologies before any failure. Link loads shown are normalized to the link with maximum load in Coronet, for each topology respectively.

5.4.5 Recovery time after failure

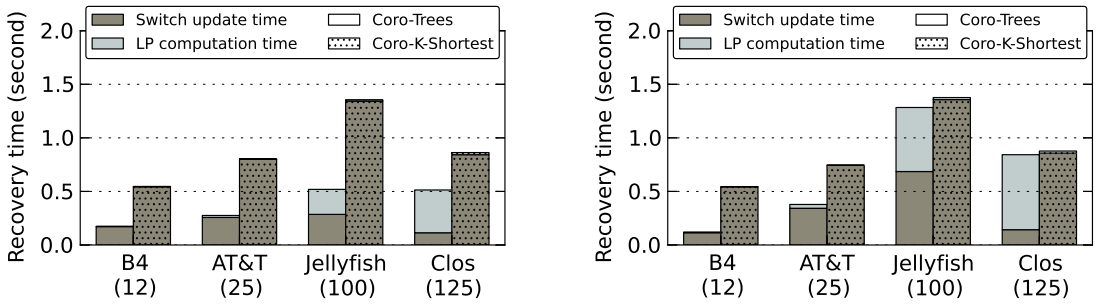
Recovery time is the most important metric for any failure recovery scheme. Here, we define the recovery time as the time that the LP solver takes to calculate back up route weights at the controller, plus the time it takes to complete updates in switches.

We evaluated the GLPK LP solver computation time for each topology by failing one link at a time, testing all links in the network (kill link, solve, restore link, repeat with next link).

Table 17: Average LP completion time with full and simplified LP model.

	B4	**AT&T**	**Jellyfish**	**Clos**
Full	10 ms	70 ms	2200 ms	2300 ms
Simplified	5 ms	30 ms	400 ms	610 ms

The first row in Table 17 shows the average LP computation time for the full LP model, which completes within one second for `B4` and `AT&T` but over two seconds for both `Jellyfish` and `Clos` topology. The LP computation time increases as the number of variables increase, which is determined by a number of factors: the number of primary route trees affected by the failure, the number of backups alive for each primary route tree, and the number of source and destination host-attached switches. Thus, the full LP model can become unusable as the size of the network increase. To overcome



(a) Recovery time with one minimum backup for any failure

(b) Recovery time with two minimum backups for any failure

Figure 49: Recovery time comparison between Coronet-Trees and Coronet-K-Shortest for the worst-case switch (switch with most updates). Coronet-Trees is shown in solid bar and Coronet-K-Shortest in shown in bar with dots. Recovery time is the sum of switch update time (grey) and LP computation (light blue). We vary the number of minimum backups guaranteed to be alive for each primary route tree after every single link failure from one to two.

lengthy computation time for large networks, we create a simplified LP model by reducing the number of weight variables, which is achieved by defining one weight variable for each `(route tree, src switch)` pair instead of for each `(route tree, src switch, dst switch)` pair. In this case, the weight distribution for a particular route tree in a switch will be identical for all group table entries corresponding to that route tree. The simplified LP model completes within 1 second for any topology, 2-6 times faster than the full LP model, which is a significant improvement. We use this simplified version of the LP model to produce the weight distribution to backups for avoiding congestion.

Figure 49 shows the recovery time after a single link failure occurs in the network for both Coronet-Trees and Coronet-K-Shortest. The switch update time is computed by dividing the number of required group table modification messages by the rule update rate we measured in § 5.4.2, which is 55 rules/second. The figure plots recovery time of the switch that needs the most number of updates in the network after a failure. All possible link failures are evaluated, and the median is taken.

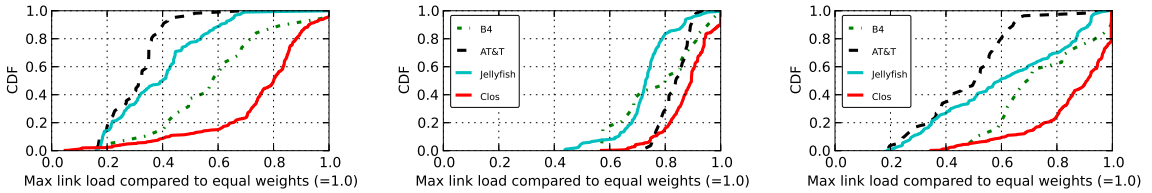
Figure 49a shows the recovery time when Coronet-Trees just guarantees one available

backup for any single failure while Figure 49b shows the recovery time when Coronet-Trees guarantees two available backups. The result shows that Coronet-Trees always has shorter switch update time for all topologies regardless of the number of minimum backups. Coronet's efficient switch update mechanism through group tables accompanied with the group table minimization process has a significant effect on creating much smaller number of modification messages required to recover from any failure. However, Figure 49b shows that Coronet has longer recovery time for `Jellyfish` and `Clos` caused by longer LP computation time. As anticipated in §5.3.1.2, the minimum number backups that are guaranteed to be available for every failure for every primary route tree certainly introduces a trade off. Larger number of backups possibly produces better spread of traffic on remaining links by having more backups available, but increases LP computation due to producing more weight variables in the linear program.

5.4.6 Avoiding congestion after failure

We now evaluate how well the link load minimization process reduces the maximum link load after a failure occurs. Figure 50 shows the cumulative distribution (CDF) of maximum link loads after every single link failure. The link load values for Coronet-Trees and Coronet-K-Shortest are normalized against the loads produced when failover traffic is equally distributed to all available backups and when Coronet-Trees is used *without* link load minimization. This is similar to what ECMP does to distribute traffic to output ports. Figure 50a shows Coronet-K-Shortest with link load minimization. Figure 50b and Figure 50c show the cases where Coronet-Trees has at least one and two backup guaranteed to be available for every failure, respectively.

The figure shows that Coronet-K-Shortest (Figure 50a) generally has lower maximum link utilization compared to Coronet-Trees (Figure 50b and 50c) . This is not surprising because Coronet-K-Shortest has a larger number of forwarding paths precomputed which makes it have a more diverse set of paths. Coronet-K-Shortest also creates shorter paths



(a) Coronet-K-Shortest max link utilization

(b) Coronet-Trees max link utilization w/ one or more backups

(c) Coronet-Trees max link utilization w/ two or more backups

Figure 50: Normalized traffic amount on the link that has maximum load. Load values are normalized against equal weight distribution with Coronet-Trees (*i.e.*, without running link load minimization) with minimum one available backup for any failure.

in terms of number of hops for each host-attached switch pair, thus fewer links are used when forwarding traffic, reducing the overall traffic load. However, Coronet-Trees shows comparable results to Coronet-K-Shortest, which is noteworthy because the number of group table entries Coronet-Trees uses is two to five times less then Coronet-K-Shortest (Figure 47), and also Coronet-Trees results in shorter recovery time (Figure 49).

Comparing Figure 50b and 50c, we see that the minimum number backups that are guaranteed to be available does have an impact on maximum link utilization. With one minimum backup, the lowest maximum link load among all single failures is around 40–60% for all topologies while with two minimum backups, the lowest maximum link load around 20–40% for all topologies. However, more guaranteed backups increases the LP computation time and eventually the overall recovery time, as shown in §5.4.5.

5.5 Related Work

While network failure resilience has been extensively studied for many years, the advent of SDN presents new challenges and opportunities for resilience. For example, the OpenVirtex [3] "network hypervisor" presents virtual topologies to SDN controllers and applications while assuming responsibility for ensuring physical connectivity when network components fail, similar to Coronet, but lacks efficient mechanisms for fast failover and

congestion avoidance, and performs only shortest path route planning. FatTire [90] proposes an SDN policy language that allows operators to specify flow-specific resilience requirements, among other properties. Coronet could be customized to provide an efficient, scalable implementation of such policies.

Other recent research efforts, Keep Forwarding [105] and Data Driven Connectivity [65], propose new data-plane protocols for extremely fast restoration of data-plane connectivity after failures. While these protocols could be used with an SDN control plane that optimizes routes over longer time scales, they require switch hardware modification, unlike Coronet which works with unmodified OpenFlow switches.

Several efforts have proposed routing traffic on diverse trees or other graphs mapped onto the network topology to achieve multi-path load balancing and/or tolerance of component failures, such as Resilient Routing Layers [57, 58, 74], multiple spanning tree routing [106], routing over vertex- or edge-redundant paths [73], and Ensemble Routing [93]. Extensive comparison of the algorithms used in these works to the tree construction in Coronet is left for future work; Coronet's algorithms appear unique in combining resilience to single failures, maximal recovery in network partition cases, and congestion avoidance through weighted failover to multiple backup trees. Less related efforts have focused on resilience using different routing techniques, e.g., source routing [97], routing tables [24], or routing in specific topologies [66].

Other recent work has used SDN to avoid congestion. For example, SWAN [34] and B4 [40] use SDN to manage traffic routing for expensive WAN links. zUpdate [64] produces schedules for congestion-free re-routing of traffic flows for planned network maintenance events. Forward Fault Correction (FFC) [63] also uses an LP solver after unplanned failures, but reserves spare bandwidth in advance of unplanned failures to avoid congestion. R3 [102] provides congestion-free recovery in IP networks using legacy OSPF routing and modified MPLS. In contrast to previous work, Coronet provides best-effort congestion avoidance with weighted fast failover to multiple backup routes, and more suited to SDN

environment and amenable to providing transparent failover service to SDN-based control applications.

Finally, prior research has proposed improvements for controller consistency and reliability [53], and reliable switch-controller communication [5]. Aditya *et al.* suggest improvements to mitigate unreliability of switch-controller and controller-controller communication [1].

5.6 Conclusion

Coronet demonstrates support for fast failover and multipath routing in arbitrary topologies using automatic construction of resource-disjoint route trees, and transparent data plane resilience for SDN applications using network topology virtualization.

There are several opportunities for future work, including: 1) Extending Coronet for k-failure resilience, possibly taking into account specific multiple-failure scenarios indicated by network operators; 2) Enabling Coronet-integrated SDN applications to themselves make use of group tables; and 3) Enabling Coronet to present different virtual topologies to meet the individual needs of different SDN apps, while minimizing the number of routing trees needed to support all the different topology abstractions simultaneously.

CHAPTER VI

RESEARCH IN NETWORK MANAGEMENT

Research in network management is hard. It is a field that touches a lot of corners, which can even extend into fields like human computer interaction (HCI) when it comes to evaluating a "good" or "bad" management system for operators. In this chapter, I will try to (1) point out why research in network management is hard, (2) provide suggestions and tips, and (3) share related resources and data we have collected so far.

6.1 Why is Network Management Research Hard?

It is hard to gain access to resources. First step in network management research is analyzing how the current network is configured and managed. To do this, it is vital to have a good and collaborative relationship with people in the IT group, which is often not the case for many enterprise networks. It is generally hard to find a time to sit down with the IT group and hear what they say, and even harder to have regular meetings or discussions. Thus, gaining access to detailed information like network topology, connections, network device and middlebox inventory, monitoring logs, and bandwidth and latency measurements does not come easily. Some sensitive data even require signing forms like Non-disclosure Agreement (NDA) form, such as network configuration files or firewall rules. Sometimes, an enterprise network lacks such valuable data because the IT department decides not to store and maintain such historical data about the network.

Some data is just hard to generate in the first place, and one example is a collection of well-maintained *network policies*. A network policy defines various kinds of access control rules, guidelines on network usage such as excessive bandwidth consumption, rate limiting, and QoS policies. However, there is no standard way or format for defining and expressing

such policy. At best, you have a set of documents that describe multiple policies in a crude manner with natural human language.

It is hard to test and validate. Even if you come up with a novel network management scheme or solution, it is extremely hard to test it out and validate its usability or performance on a realistic network. As network connectivity cannot be disturbed on the production network, there is almost no chance that you will be able to do a live test run on a production network. However, emulating a testbed and simulating realistic traffic pattern on a network is not trivial as well. It is not easy, if not impossible to capture all traffic and details of the network, model it, and make a reasonably realistic testbed. One way is to create or transform a small portion of the real network into a testbed and test a new solution in a smaller scale. However, a question remains on how to derive a scientifically solid conclusion that the solution will remain feasible on a network with larger scale.

Measuring and evaluating *usability* is even harder. How to do you know a solution makes a certain network task *easier* than others? What is the best way to compare usability between different solutions? One way is to do well-planned survey against network operators who used various different tools and management solutions, but this is not always easy to do, especially in large scale.

It is hard to change. A network cannot be changed easily, especially if it involves changing the physical network infrastructure as well. It is the same reason as why widespread IPv6 deployment is far from being the reality even after decades of discussion and development. Whatever solution a researcher come up with, it would have the best chance of being deployed if it can be done incrementally.

6.2 Suggestions

Reach out to network operators. Network engineers and operators have the most hands-on experience when it comes to network management and configuration. They know the

problem areas, such as what is missing and what is most frustrating task in network management. They have stories, best practices, and tools and scripts they use when configuring their network. They have invaluable data, which is very important when it comes to research. Examples are: current and historic network configuration files from network devices, network topologies, syslogs, measurements on bandwidth, latency, jitter, and packet loss. They will also likely have system monitoring data such as network component's CPU usage, memory, and rule table size. These data can provide invaluable insight into how networks are configured and managed. Thus, it is very important for researchers to reach out to network engineers and operators who work on the field, *i.e.*, the real world.

There are conferences and workshops that can give researches opportunity to meet and discuss with operators, even outside of their own campus or enterprise network. In North America, the North American Network Operators's Group, or NANOG [77], is an organization of network operators in North America, and they have a meeting in every four months primarily held in the United States and Canada, where they share their experience on networks, current and future deployment or upgrade plans, as well as best practices. Internet2 [39] is a more research-friendly organization where not only network operators but also researchers and government representatives gather around to collaborate via innovative technologies with great focus on networking. Internet2 has a Technology Exchange event every year, as well as a Global Summit event annually. To gain insight and knowledge about standard protocols, The Internet Engineering Task Force (IETF) meetings [38], which are held three times a year, is the best event to attend.

Don't rule out the method of doing a large scale survey. It is rare for a computer science or engineering researcher to conduct survey with human subjects, especially in a large scale. However, in certain cases, such large-scale survey is the best way to identify a problem area, and also evaluate and compare different solutions. It is important to note that such situation appears frequently in network management research where one needs a result that statistically proves that a proposed solution makes network management *easier* than using

other solutions. Such approach becomes more applicable when there is no straightforward and systematic way to evaluate the "easiness" of a network management approach or solution. Therefore, researchers in network management, especially the ones who aim to build a new management solution or approach that makes network management easier, should not be shy of orchestrating and performing large-scale surveys against human subjects. Of course, as it is a study against human subjects, protocols and procedures should be approved by the Institutional Review Board (IRB).

6.3 Data and Resources

Example Network Policies. Real network polices used in production enterprise or campus networks are hard to find. At most, they are described in a very high-level without much details. Nonetheless, they are still valuable. We have mined the Internet in hope to find real network policies, particularly used in campus networks because they are mostly public. Such examples are gathered and uploaded in a public repository, in hope to help researchers find realistic network policies [21].

Survey Template. We have uploaded the questionnaire we have used to survey the users who have used Kinetic, our dynamic network control platform [51]. Researchers may consult this example questionnaire to make their own version, which they can use to do a survey to evaluate the effectiveness of a solution.

Kinetic Code. As mentioned in Chapter 4, the source code for Kinetic can be found here: [49]. More information about the Kinetic project can be found in the Kinetic Website [50]. Tutorial for Kinetic can be found here: [52].

CHAPTER VII

CONCLUSION

7.1 Summary of contributions

The first extensive longitudinal analysis study on network configuration changes. By inspecting and analyzing over five years of configuration file change logs from network devices from two large campus networks, we have shed light on understanding how configuration changes over time, and why. We focused on capturing configuration characteristics and common practices by examining change patterns from CVS commit log data, in contrast to earlier studies, which have mostly examined a single configuration snapshot. Moving a step forward from earlier studies, which have focused mostly on router configuration, we also study configuration changes for firewalls and switches, which comprise a significant portion of the networks. We study the frequency and extent of configuration updates, showing the similarities and differences across networks; identify the prevalence of correlated changes; and shed light on how various network-wide factors affect evolution. We found that a variety of network-wide tasks contribute to changes in devices, including provisioning, management, traffic engineering, security, and QoS. We also confirmed that network configuration changes a lot; a campus network may experience anywhere from 1,000 to 18,000 changes per month.

Making network configuration easier with abstractions and reusable modules. This dissertation focuses on a variety of network events, and posits that such events and corresponding reactions (*i.e.*, change of forwarding behavior) should be modeled into the network policy in the first place. Instead of thinking a network-wide network policy as a combination of multiple static network policies with network events triggering transitions between them, we present a domain specific language and platform that allow operators

to encode network events and transitions between network states in a software program. Through an extensive user study and quantitative evaluation, we show that such approach enables to express network policies in a more concise, intuitive, and less error-prone way. Our system allows composing multiple network control programs that have distinctive network tasks together, enabling to express more complex network policies. This dissertation presents modular services and network control programs that can be reused by operators, and promotes code reusability.

Bringing verification and guarantees in networking through software. We explore ways to provide verifications and guarantees in the network through software. Kinetic not only enables operators to write network policies in a concise and intuitive way, but also provides verification capabilities that can be used to verify whether the created software program correctly encodes the event and reaction control logic using model checking in temporal logic. Coronet uses graph theory to create a set of primary and backup paths that guarantee the network to survive any single data-plane failure in the network, and uses linear programming to guarantee to find the optimal traffic failover placement on backup paths to minimize maximum load on links.

7.2 Lessons learned

We provide lessons we have learned throughout our study in hope to guide anyone who plans to do research related to this dissertation.

Appropriate level of constraints are better than none. SDN has opened a lot of doors to improving network management and configuration by separating the control and data plane, providing unprecedented flexibility to network control through network programmability. Such openness, especially in the control plane, has encouraged lots of different control platforms and languages for building network control programs. This, however, can introduce more problems then promises with a lack of guidance. Too much freedom and flexibility in the control plane can make SDN control programs inefficient, buggy, and hard

to understand and debug. Network control programs will also become incompatible with each other. Through our studies, we have discovered that a constrained but structured language and control platform can be better than an environment with more flexibility. With Kinetic, we have constrained the language to guide operators to write control programs that are easier to write and understand, and also that is automatically verifiable. With Coronet, we have constrained the operators to build SDN control applications on a simplified virtual topology while Coronet takes care of routing and recovery in the physical data plane. Such approaches indeed limits operator's freedom in the control plane, but brings benefits to creating a more intuitive, secure, and correct network control program.

Usability is important. It is important to have the right level of abstraction for usability. Throughout out studies, we have discovered that users will prefer low-level language and methods to higher-level abstractions if the level of abstraction is not appropriate. Usability of the system is very important. To test the usability of a system or language, a qualitative user study against a wide audience is essential. Quantitative evaluation of the usability or complexity of a language or system has limitations, and it is generally hard to do. A well-designed survey or user study against humans directly assesses the usability of a system, and likely produces more relevant and meaningful results. In this dissertation, we have conducted an extensive user study against over 870 participants who used Kinetic to evaluate its usability. uCap, a wireless router platform that has Kinetic as its backend to enable dynamic control, also went through an extensive user study and hour-long interviews to assess its usability [70].

Verification is needed in every stack in the network. Based on our survey with Kinetic, verification is clearly needed in network configuration. Nearly 20% of participants said that they must change their network configurations more than once a day, and more strikingly, 89% of respondents indicated that they were never completely certain that their changes to the configuration would not introduce a new problem or bug, and 82% were concerned

that the changes would introduce problems with *existing* functionality that was unrelated to the change. The two most common aspects of configuration that operators wanted to see *automated* were correctness testing (37%) and quality of service and performance assurances (24%). The two most common aspects of configuration that participants wanted to see *verified* were general correctness problems (37%) and security properties (26%).

Such verifications should happen in every stack in the network. Network operators give us feedback in the NANOG 62 meeting that some type of verification is extremely needed in the data plane; some mechanism that confirms the instructed changes from the control plane actually took place in the switches in the data plane. Operators need (1) verification of the correctness of the configuration or SDN control program they wrote, (2) verification that the instructions are delivered correctly to the data plane, and (3) verification that the network devices in the data plane has the correct rules and performs correctly according to the rules.

Research in software engineering, theoretical computer science, and math can bring great improvements to computer networks. There are lots of great research in software engineering, machine learning, theoretical computer science, and math, which can be applied to various parts of networking and telecommunication research. We have used practices and techniques in software engineering for analyzing revisions in network configuration files. To verify the correctness of the control logic encoded in Kinetic control programs we applied a model checking framework developed by Clarke and Emerson [13, 14] and subsequently refined by McMillan [72]. Model checking can guarantee that a finite state machine (FSM) satisfies certain properties that are expressed in different types of logics; this feature makes FSMs a logical choice for expressing Kinetic policies. We have extensively used graph theory work in Coronet in the process of creating algorithms to build a set of primary and backup routing paths (or trees) that guarantees *fast* failure recovery for any single data-plane component failure. We believe there are much more interesting and useful research in various scientific fields that are applicable to networking research, which

can improve how we manage and configure our network.

7.3 Future work

This dissertation concludes with summarizing interesting research opportunities and ideas related to network configuration analysis, automating configuration, composing network control application and services, and formal verification in control and data plane.

Network analysis and data mining. If we can identify configuration change patterns that frequently occur in the network and use such information to automatically synthesize programs that test and apply relevant changes according to a given high-level operational task, the number of human-induced errors can be reduced while also cutting down operator's workload needed to make such change. A promising approach is to use data mining techniques; they are effective at finding patterns with specific characteristics that occur repetitively. I want to explore how archetypical data mining algorithms can be used on network configuration data to reveal common changes operators make in the network and help design a better solution that can automate changes in the network.

Verifying correctness. Kinetic sits squarely in the realm of ongoing work on network verification and complements the growing body of work on data-plane verification, such as Veriflow [45] and NetPlumber [43]. As these tools can help network operators ask questions about snapshots of data-plane state, and Kinetic can help network operators reason about the dynamics of network policies (which ultimately compile to the corresponding data-plane state), the approaches are complementary. Similarly, Kinetic needs the path guarantees that consistent updates [91] provide to guarantee that the properties it verifies are preserved during state transitions; conversely, consistent updates could be extended to reason about temporal properties such as those that Kinetic can express. One natural next step would be to combine these approaches.

Conflict resolution in SDN. Software-Defined Networking is opening a lot of doors to new

applications that provide various types of services in the network. Some people anticipates an "SDN App Store" to emerge where network operators can "shop" for applications or services that meet their needs. The challenge is how to compose and run multiple applications or services together without having unforeseen and undesirable consequences. Such conflicts are inevitable to happen if the running applications are unaware of each other, and no precaution was made to prevent such conflicts. An attempt to make network services and applications conflict-free has been made in the past in the pre-SDN era with legacy network devices, but without any meaningful success. I want to explore how such goal can be achieved with SDN, which I believe to have a better chance of success because: 1) Reasoning with software is easier than hardware, and 2) SDN programs use a shared protocol (*e.g.*, OpenFlow).

SDN for home networks. The *uCap* project was the first attempt to enable non-tech-savvy home network users to have better command over their network. By using *uCap*, normal home users were able to set data caps on end-host devices (*e.g.*, tablet PCs, laptops) through simple clicking and entering numbers in the Web-based user interface. Going further, we envision home users setting much more sophisticated network polices, such as application or user-specific rate-limiting, or setting different bandwidth and caps based on time of day. Research in finding the right design, abstractions, and leveraging software and hardware capabilities will make such vision possible.

REFERENCES

[1] AKELLA, A. and KRISHNAMURTHY, A., "A highly available software defined fabric," in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, p. 21, ACM, 2014.

[2] AL-FARES, M., LOUKISSAS, A., and VAHDAT, A., "A scalable, commodity, data center network architecture," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, (New York, NY, USA), pp. 63–74, ACM, 2008.

[3] AL-SHABIBI, A., DE LEENHEER, M., GEROLA, M., KOSHIBE, A., PARULKAR, G., SALVADORI, E., and SNOW, B., "Openvirtex: Make your virtual sdns programmable," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, (New York, NY, USA), pp. 25–30, ACM, 2014.

[4] BALL, T., BJØRNER, N., GEMBER, A., ITZHAKY, S., KARBYSHEV, A., SAGIV, M., SCHAPIRA, M., and VALADARSKY, A., "Vericon: towards verifying controller programs in software-defined networks," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, p. 31, ACM, 2014.

[5] BEHESHTI, N. and ZHANG, Y., "Fast failover for control traffic in software-defined networks," in *Global Communications Conference (GLOBECOM), 2012 IEEE*, pp. 2665–2670, Dec 2012.

[6] BENSON, T., AKELLA, A., and MALTZ, D., "Unraveling Complexity in Network Management," in *Proc. 6th USENIX NSDI*, (Boston, MA), Apr. 2009.

[7] BODÍK, P., MENACHE, I., CHOWDHURY, M., MANI, P., MALTZ, D. A., and STOICA, I., "Surviving failures in bandwidth-constrained datacenters," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, (New York, NY, USA), pp. 431–442, ACM, 2012.

[8] CALDWELL, D., GILBERT, A., GOTTLIEB, J., GREENBERG, A., HJALMTYSSON, G., and REXFORD, J., "The cutting edge of IP router configuration," in *Hotnets-II*, (Cambridge, MA), Nov. 2003.

[9] CANINI, M., VENZANO, D., PERESINI, P., KOSTIC, D., and REXFORD, J., "A NICE Way to Test OpenFlow Applications," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2012.

[10] "Chef." `http://www.opscode.com/chef/`.

[11] CHEN, X., MAO, Z., and VAN DER MERWE, J., "Towards automated network management: network operations using dynamic views," in *INM 07*.

[12] CIMATTI, A., CLARKE, E., GIUNCHIGLIA, E., GIUNCHIGLIA, F., PISTORE, M., ROVERI, M., SEBASTIANI, R., and TACCHELLA, A., "NuSMV 2: An Opensource Tool for Symbolic Model Checking," in *Computer Aided Verification*, pp. 359–364, Springer, 2002.

[13] CLARKE, E. M., EMERSON, E. A., and SISTLA, A. P., "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[14] CLARKE, E. M., GRUMBERG, O., and PELED, D. A., *Model checking*. MIT press, 1999.

[15] "CMU Network Bandwidth Usage Guidelines." `http://www.cmu.edu/computing/network/connect/guidelines/bandwidth.html`.

[16] "Coursera: SDN class." `https://www.coursera.org/course/sdn`, 2014.

[17] "Coursera: SDN Walled Garden Assignment." `http://goo.gl/JYMret`, 2014.

[18] EICK, S., GRAVES, T., KARR, A., MARRON, J., and MOCKUS, A., "Does code decay? assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001.

[19] EILENBERG, S., *Automata, languages, and machines*, vol. 1. Access Online via Elsevier, 1974.

[20] ENCK, W., MOYER, T., MCDANIEL, P., SEN, S., SEBOS, P., SPOEREL, S., GREENBERG, A., SUNG, Y., RAO, S., and AIELLO, W., "Configuration management at massive scale: system design and experience," *IEEE Journal on Selected Areas in Communications*, vol. 27, no. 3, pp. 323–335, 2009.

[21] "Example campus network policies." `https://github.com/hyojoonkim/Resources/tree/master/Example_Network_Policies`, 2015.

[22] FEAMSTER, N. and BALAKRISHNAN, H., "Detecting BGP Configuration Faults with Static Analysis," in *Proc. 2nd USENIX NSDI*, (Boston, MA), May 2005.

[23] FEAMSTER, N., REXFORD, J., and ZEGURA, E., "The Road to SDN," *Queue*, vol. 11, pp. 20:20–20:40, Dec. 2013.

[24] FEIGENBAUM, J., GODFREY, B., PANDA, A., SCHAPIRA, M., SHENKER, S., and SINGLA, A., "Brief announcement: On the resilience of routing tables," in *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, (New York, NY, USA), pp. 237–238, ACM, 2012.

[25] FELDMANN, A. and REXFORD, J., "IP network configuration for intradomain traffic engineering," *IEEE Network*, Sept. 2001.

[26] "Floodlight OpenFlow Controller." `http://floodlight.openflowhub.org/`.

[27] FOSTER, N., FREEDMAN, M. J., GUHA, A., HARRISON, R., KATTA, N. P., MONSANTO, C., REICH, J., REITBLATT, M., REXFORD, J., SCHLESINGER, C., STORY, A., and WALKER, D., "Languages for Software-Defined Networks," *IEEE Communications*, vol. 51, pp. 128–134, Feb. 2013.

[28] FOSTER, N., HARRISON, R., FREEDMAN, M., MONSANTO, C., REXFORD, J., STORY, A., and WALKER, D., "Frenetic: A network programming language," in *International Conference on Functional Programming (ICFP)*, Sept. 2011.

[29] "Frenetic, Pyretic and Resonance." `http://blog.sflow.com/2013/08/frenetic-pyretic-and-resonance.html`. Note: Kinetic was previously known as Resonance.

[30] "GNU Linear Programming kit." `https://www.gnu.org/software/glpk/`.

[31] GOTTLIEB, J., GREENBERG, A., REXFORD, J., and WANG, J., "Automated Provisioning of BGP Customers," *IEEE Network*, 2003.

[32] GUPTA, A., VANBEVER, L., SHAHBAZ, M., DONOVAN, S. P., SCHLINKER, B., FEAMSTER, N., REXFORD, J., SHENKER, S., CLARK, R., and KATZ-BASSETT, E., "SDX: A Software Defined Internet Exchange," in *ACM SIGCOMM*, (Chicago, IL), pp. 579–580, 2014.

[33] HINRICHS, T. L., GUDE, N. S., CASADO, M., MITCHELL, J. C., and SHENKER, S., "Practical declarative network management," in *ACM/USENIX Workshop on Research on Enterprise Networking (WREN)*, (Barcelona, Spain), pp. 1–10, 2009.

[34] HONG, C.-Y., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., and WATTENHOFER, R., "Achieving high utilization with software-driven wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, (New York, NY, USA), pp. 15–26, ACM, 2013.

[35] HOPCROFT, J. and TARJAN, R., "Algorithm 447: Efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, pp. 372–378, June 1973.

[36] "HP SDN Dev Center: VAN Controller." `http://h17007.www1.hp.com/us/en/networking/solutions/technology/sdn/devcenter/index.aspx`.

[37] HUANG, D. Y., YOCUM, K., and SNOEREN, A. C., "High-fidelity switch models for software-defined network emulation," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, (New York, NY, USA), pp. 43–48, ACM, 2013.

[38] "Internet Engineering Task Force." https://www.ietf.org.

[39] "Internet2." http://www.internet2.edu.

[40] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HÖLZLE, U., STUART, S., and VAHDAT, A., "B4: Experience with a globally-deployed software defined wan," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, (New York, NY, USA), pp. 3–14, ACM, 2013.

[41] KAGDI, H., COLLARD, M., and MALETIC, J., "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 2, pp. 77–131, 2007.

[42] KANG, N., LIU, Z., REXFORD, J., and WALKER, D., "Optimizing the one big switch abstraction in software-defined networks," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pp. 13–24, ACM, 2013.

[43] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., and WHYTE, S., "Real-time Network Policy Checking using Header Space Analysis," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[44] KAZEMIAN, P., VARGHESE, G., and MCKEOWN, N., "Header Space Analysis: Static Checking for Networks," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2012.

[45] KHURSHID, A., ZHOU, W., CAESAR, M., and GODFREY, P. B., "Veriflow: Verifying network-wide invariants in real time," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, (Lombard, IL), Apr. 2013.

[46] KIM, H., BENSON, T., AKELLA, A., and FEAMSTER, N., "The Evolution of Network Configuration: A Tale of Two Campuses," in *ACM SIGCOMM Internet Measurement Conference (IMC)*, (Berlin, Germany), pp. 499–514, 2011.

[47] KIM, H., REICH, J., GUPTA, A., SHAHBAZ, M., FEAMSTER, N., and CLARK, R., "Kinetic: Verifiable dynamic network control," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 59–72, USENIX Association, May 2015.

[48] KIM, H., SANTOS, J., TURNER, Y., SCHLANSKER, M., TOURRILHES, J., and FEAMSTER, N., "Coronet: Fault tolerance for software defined networks," in *Network Protocols (ICNP), 2012 20th IEEE International Conference on*, pp. 1–2, Oct 2012.

[49] "Kinetic source code." https://github.com/frenetic-lang/pyretic/tree/kinetic Note: Evaluation specific data & scripts are in the kinetic_test branch.

[50] "Kinetic Project Website," 2015. http://kinetic.noise.gatech.edu/.

[51] "Kinetic survey." https://github.com/hyojoonkim/Resources/tre/master/Example_Survey, 2015.

[52] "Kinetic tutorial." https://github.com/hyojoonkim/Resources/tree/master/Kinetic, 2015.

[53] KOPONEN, T., CASADO, M., GUDE, N., STRIBLING, J., POUTIEVSKI, L., ZHU, M., RAMANATHAN, R., IWATA, Y., INOUE, H., HAMA, T., and SHENKER, S., "Onix: A distributed control platform for large-scale production networks," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.

[54] KREUTZ, D., RAMOS, F. M., VERISSIMO, P., ROTHENBERG, C. E., AZODOLMOLKY, S., and UHLIG, S., "Software-defined networking: A comprehensive survey," *proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[55] KROTHAPALLI, S. D., SUN, X., SUNG, Y.-W. E., YEO, S. A., and RAO, S. G., "A toolkit for automating and visualizing vlan configuration," SafeConfig '09, (New York, NY, USA), pp. 63–70, ACM, 2009.

[56] KRUSKAL, J. B., "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[57] KVALBEIN, A., HANSEN, A., CICIC, T., GJESSING, S., and LYSNE, O., "Fast recovery from link failures using resilient routing layers," in *Computers and Communications, 2005. ISCC 2005. Proceedings. 10th IEEE Symposium on*, pp. 554–560, June 2005.

[58] KVALBEIN, A. and LYSNE, O., "How can multi-topology routing be used for intradomain traffic engineering?," in *Proceedings of the 2007 SIGCOMM Workshop on Internet Network Management*, INM '07, (New York, NY, USA), pp. 280–284, ACM, 2007.

[59] LE, F., LEE, S., WONG, T., KIM, H., and NEWCOMB, D., "Minerals: using data mining to detect router misconfigurations," in *Proc. MineNets '06*, (Pisa, Italy), pp. 293–298, Sept. 2006.

[60] LE, F., XIE, G., PEI, D., WANG, J., and ZHANG, H., "Shedding Light on the Glue Logic of the Internet Routing Architecture," pp. 39–50, 2008.

[61] LEHMAN, M., "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[62] "List of Real Campus Network Acceptable Use Policies." http://goo.gl/pdR6Sd, 2014.

[63] LIU, H. H., KANDULA, S., MAHAJAN, R., ZHANG, M., and GELERNTER, D., "Traffic engineering with forward fault correction," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, (New York, NY, USA), pp. 527–538, ACM, 2014.

[64] LIU, H. H., WU, X., ZHANG, M., YUAN, L., WATTENHOFER, R., and MALTZ, D., "zupdate: Updating data center networks with zero loss," *SIGCOMM Comput. Commun. Rev.*, vol. 43, pp. 411–422, Aug. 2013.

[65] LIU, J., PANDA, A., SINGLA, A., GODFREY, B., SCHAPIRA, M., and SHENKER, S., "Ensuring connectivity via data plane mechanisms," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, (Lombard, IL), pp. 113–126, USENIX, 2013.

[66] LIU, V., HALPERIN, D., KRISHNAMURTHY, A., and ANDERSON, T., "F10: A fault-tolerant engineered network," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, (Lombard, IL), pp. 399–412, USENIX, 2013.

[67] MAHAJAN, R., WETHERALL, D., and ANDERSON, T., "Understanding BGP misconfiguration," in *Proc. ACM SIGCOMM*, (Pittsburgh, PA), pp. 3–17, Aug. 2002.

[68] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., and KING, S. T., "Debugging the data plane with anteater," in *ACM SIGCOMM*, (Toronto, Ontario, Canada), pp. 290–301, 2011.

[69] MALTZ, D., XIE, G., ZHAN, J., ZHANG, H., HJÁLMTÝSSON, G., and GREENBERG, A., "Routing Design in Operational Networks: A Look from the Inside," in *Proc. ACM SIGCOMM*, (Portland, OR), Aug. 2004.

[70] MARSHINI CHETTY, HYOJOON KIM, S. S. S. B. N. F. K. E., "ucap: An internet data management tool for the home," in *ACM CHI*, April 2015.

[71] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., and TURNER, J., "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[72] MCMILLAN, K. L., *Symbolic model checking*. Springer, 1993.

[73] MÉDARD, M., FINN, S. G., and BARRY, R. A., "Redundant trees for preplanned recovery in arbitrary vertex-redundant or edge-redundant graphs," *IEEE/ACM Trans. Netw.*, vol. 7, pp. 641–652, Oct. 1999.

[74] MENTH, M. and MARTIN, R., "Network resilience through multi-topology routing," in *Design of Reliable Communication Networks, 2005. (DRCN 2005). Proceedings.5th International Workshop on*, pp. 271–277, Oct 2005.

[75] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., and WALKER, D., "Composing Software-Defined Networks," in *USENIX NSDI*, 2013.

[76] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., and WALKER, D., "Composing Software-Defined Networks," in *USENIX NSDI*, 2013.

[77] "North American Network Operators' Group." `https://www.nanog.org`.

[78] NELSON, T., FERGUSON, A. D., SCHEER, M. J., and KRISHNAMURTHI, S., "Tierless programming and reasoning for software-defined networks," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, (Seattle, WA), pp. 519–531, Apr. 2014.

[79] "NOX: An OpenFlow controller." `http://www.noxrepo.org`.

[80] NUNES, B., MENDONCA, M., NGUYEN, X.-N., OBRACZKA, K., and TURLETTI, T., "A survey of software-defined networking: Past, present, and future of programmable networks," *Communications Surveys Tutorials, IEEE*, vol. 16, pp. 1617–1634, Third 2014.

[81] "Open Networking Foundation." `https://www.opennetworking.org`.

[82] "Open Daylight." `http://www.opendaylight.org/`.

[83] "OpenDaylight." `http://www.opendaylight.org/`.

[84] "OpenFlow Specification v1.1.0." `http://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.1.0.pdf`.

[85] "Open vSwitch." `http://openvswitch.org/`.

[86] "POX OpenFlow controller." `http://www.noxrepo.org/pox/about-pox/`.

[87] "Puppet." `http://puppetlabs.com/solutions/juniper-networks`.

[88] RAGHAVAN, S., ROHANA, R., LEON, D., PODGURSKI, A., and AUGUSTINE, V., "Dex: A semantic-graph differencing tool for studying changes in large code bases," 2004.

[89] "Really Awesome New Cisco ConfIg Differ (RANCID)." `http://www.shrubbery.net/rancid/`, 2004.

[90] REITBLATT, M., CANINI, M., GUHA, A., and FOSTER, N., "Fattire: Declarative fault tolerance for software-defined networks," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, (New York, NY, USA), pp. 109–114, ACM, 2013.

[91] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., and WALKER, D., "Abstractions for Network Update," in *ACM SIGCOMM*, (Helsinki, Finland), pp. 323–334, 2012.

[92] "Ryu SDN Framework." `http://osrg.github.io/ryu/`, 2014.

[93] SCHLANSKER, M., TURNER, Y., TOURRILHES, J., and KARP, A., "Ensemble routing for datacenter networks," in *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '10, (New York, NY, USA), pp. 23:1–23:12, ACM, 2010.

[94] SCHMIDT, J. M., "A simple test on 2-vertex-and 2-edge-connectivity," *Information Processing Letters*, vol. 113, no. 7, pp. 241–244, 2013.

[95] SETHI, D., NARAYANA, S., and MALIK, S., "Abstractions for model checking sdn controllers," in *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pp. 145–148, 2013.

[96] SINGLA, A., HONG, C.-Y., POPA, L., and GODFREY, P. B., "Jellyfish: Networking data centers randomly.," in *NSDI*, vol. 12, pp. 17–17, 2012.

[97] STEPHENS, B., COX, A. L., and RIXNER, S., "Plinko: Building provably resilient forwarding tables," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, (New York, NY, USA), pp. 26:1–26:7, ACM, 2013.

[98] SUNG, Y., RAO, S., SEN, S., and LEGGETT, S., "Extracting Network-Wide Correlated Changes from Longitudinal Configuration Data," in *Proc. PAM*, (Seoul, South Korea), Apr. 2009.

[99] "The Internet Topology Zoo." `http://www.topology-zoo.org/dataset.html`.

[100] VOELLMY, A. and HUDAK, P., "Nettle: Taking the sting out of programming network routers," in *Practical Aspects of Declarative Languages (PADL)*, pp. 235–249, Springer, 2011.

[101] VOELLMY, A., KIM, H., and FEAMSTER, N., "Procera: a language for high-level reactive network control," in *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networks (HotSDN)*, (Helsinki, Finland), pp. 43–48, 2012.

[102] WANG, Y., WANG, H., MAHIMKAR, A., ALIMI, R., ZHANG, Y., QIU, L., and YANG, Y. R., "R3: resilient routing reconfiguration," *SIGCOMM Comput. Commun. Rev.*, vol. 41, pp. –, Aug. 2010.

[103] WEN, X., LI, L., DIAO, C., ZHAO, X., and CHEN, Y., "Compiling Minimum Incremental Update for Modular SDN Languages," in *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pp. 193–198, ACM, 2014.

[104] XIE, G., ZHAN, J., MALTZ, D., ZHANG, H., GREENBERG, A., HJALMTYSSON, G., and REXFORD, J., "On static reachability analysis of IP networks," in *IEEE INFOCOM*, vol. 3, pp. 2170–2183, 2005.

[105] YANG, B., LIU, J., SHENKER, S., LI, J., and ZHENG, K., "Keep forwarding: Towards k-link failure resilient routing," in *INFOCOM, 2014 Proceedings IEEE*, pp. 1617–1625, April 2014.

[106] YENER, B., OFEK, Y., and YUNG, M., "Design and performance of convergence routing on multiple spanning trees," in *Global Telecommunications Conference, 1994. GLOBECOM '94. Communications: The Global Bridge., IEEE*, pp. 169–175 vol.1, Nov 1994.