© 2018 Gohar Irfan Chaudhry

NETWORK ANALYSIS, INFERENCE AND VERIFICATION

BY

GOHAR IRFAN CHAUDHRY

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Associate Professor Matthew Caesar

ABSTRACT

Securely operating large-scale networks is a non-trivial task involving interactions between various hardware devices, protocols, and configurations, all of which need to work in tandem for the network to be secure and in the desired state that the network administrators want it to be in. Misconfigurations or malicious activities in the network can disrupt it resulting in dire effects including but not limited to outages of critical applications and breach of sensitive information.

In this work, we propose a robust framework for diagnosing such anomalies across enterprise networks, and study their impact in terms of changes in routing behavior and reachability. To study the network as closely as possible to its actual behavior we perform analysis on data plane features as they govern the journey of a packet during its life-cycle across the network. We perform temporal analysis of the network as a whole and inspect the evolution of various properties. We then determine the deviation of the network relative to its previous states and identify as accurately as possible if the current state is anomalous. Given the historic states of the network over some time, we also try to infer high-level policies and invariants in the network. These allow for running various verification techniques on the network. Finally, we propose a network verification tool designed to verify the network as a dynamic, multi-layer distributed system. The richness of this tool's network model allows it to find network issues that are not detectable using state of the art tools which work solely on either data plane states or control plane states without examining the interaction of the two among themselves and temporally with the network environment. Building on this verification tool, we propose a technique for high-coverage testing of end-to-end network correctness using the real software that is deployed in these networks; our design is effectively a hybrid, using an explicit-state model checker to explore all network-wide execution paths and event orderings, but executing real software as subroutines for each device. We show that this approach can detect correctness issues that would be missed both by existing verification and testing approaches, and a prototype implementation suggests that the technique can scale to larger networks with reasonable performance.

Thus, our framework provides an end to end solution for network analysis, inference and verification.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I want to thank my advisor, Dr. Matthew Caesar, for encouraging me to work on challenging problems and providing me with constant help and guidance whenever I encountered roadblocks. I would also like to thank the University of Illinois Computer Science Department for providing me with access to invaluable resources enabling me to complete this project and avail countless learning oppurtunities on the way.

I am also thankful to my peers and professors (especially Dr. Brighten Godfrey, Santosh Prabhu and Hassan Shahid Khan) who helped me on this journey and contributed towards various aspects of this project. Finally, I would not have been able to come this far without the unconditional love and support from my parents, my sister, my fiancé, all my friends and family.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 ANALYSIS & INFERENCE

Given the enormous amount of network data that is obtained every day in production networks, it becomes a challenging task to anlayze all that data in a meaningful way to ensure that the network meets the desired properties at all times. The challenge is two-fold; firstly, it becomes difficult to statistically view how the network evolves over time and whether any given state of the network is anomalous relative to previous seen states and secondly, its a challenge to infer what the underlying invariants/policies in the network are given the tremendous amount of historic network snapshots/logs. There has been a lot of work done to study historic network data and find anomalies and a lot of work has been done to infer network policies for specific kinds of devices.

However, what the state of the art does not address is a wholesome anlaysis and inference of the network as one entity. To this end, we propose:

1. A set of statistical analysis methods that do not require specific knowledge of the network and consume the data plane state at different times to study network evolution and identify unusual states.

2. A policy inference mechanism which finds the most frequently observed kinds of packets exchanged between pairs of devices.

3. A summarization technique to find the minimum amount of data units to describe network policies extracted by the inference mechanism which can be further fed into network tools like automated invariant checkers and verifiers.

1.2 VERIFICATION & TESTING SOFTWARIZED NETWORKS

Beyond statistical methods, we also provide formal verification techniques for ensuring correctness of networks which is a difficult yet critical task. A growing number of network verification tools are targeted towards automating this process as much as possible, thereby reducing the burden on the network operator. Verification platforms have improved steadily in the recent years, both in terms of scope and scale. Starting from offline data plane verification tools like Anteater [1] and HSA [2], the state of the art has evolved to support real-time data plane verification [3, 4], and more recently, analysis of configurations [5, 6, 7, 8].

Existing network verification techniques center around analysis of one or more individual data plane states to verify policies of interest. In the case of data plane verifiers, a single data plane state, often collected directly from the network, is presented as the network abstraction, and is analysed for policy violations. While helpful, this approach does not verify the effect of configuration prior to deployment. Configuration verification tools such as ERA, ARC and Minesweeper start from the network configuration, but their approach is to encode the control plane in models such as graphs or logical formulae, essentially designed to generate one or more data plane states. These data plane states are computed as representation of the *outcome* of the control plane execution. The actual analysis of the policy happens on the individual data plane states that are generated in this manner.

But in reality, networks involve many dynamically moving parts. At the most basic level, does the control plane converge? If so, can it converge to multiple states? How is traffic affected as the system is being modified, either by a distributed control plane, or by SDN-style software control of the data plane? Perhaps due to the sheer complexity of this network system, verification tools have either ignored it altogether and focused on the results of the control plane's execution [6, 7, 8], or modeled specific forms of the dynamic behavior, such as individual protocols [9, 10] or only the data plane [11].

As part of the final stage in the pipeline and as a solution to the aforementioned problems, we propose **Plankton**, the first network verification platform that models the network as a dynamic, multi-layer distributed system. This is a nontrivial goal. The system will have to understand control plane dynamics, rather than simply computing a data plane output of the control plane. However, the control plane does not exist in isolation; it must incorporate data plane events, including perhaps sequences of multiple packets relevant to policies that depend on actions across time. All of these cross-layer events may be woven together in arbitrary order, creating exponential complexity as we scale to large networks.

Moving one step further, we realize that for a network incorporating extensive software elements, assuming a network model becomes a serious limitation, for several reasons:

1. Implementation bugs: Given the highly specialized nature of middleboxes, there is both a high likelihood of bugs, and also the risk of them being undiagnosed for a significant length of time. These bugs (or simply implementation quirks) may cause network policy violations even if the network operator has configured the middlebox fully correctly. Writing a bug-for-bug faithful model of the software would be close to impossible.

2. Lack of a model: Part of the point of building a softwarized network is to be able to code custom features, behaviors, and even whole distributed systems. As such, we may

lack a starting point for a model, unlike data plane and control plane elements that typically operate with standardized protocols (BGP, spanning tree, etc.).

To address these shortcomings, we pose the question: how close can we come to getting the best of both worlds, with the faithfulness of real software and the high coverage of verification?

We initiate an exploration of that question with a hybrid approach. Intuitively, to retain high accuracy, we need to execute the actual running software with its actual configuration for each *individual* component. But we use those components, and partial executions of them, as building blocks to assemble and reassemble in an exhaustive exploration of network-wide execution paths driven by an explicit-state model checker. Our key contributions in this regard are as follows:

1. We propose a hybrid approach combining emulation-based testing and model-based verification, and show through examples that it can find intent violations that would be missed by each approach individually.

2. We design such a hybrid system, **Plankton-neo**, which builds on the Plankton network verification platform but inserts invokation of real software network elements in its execution loop, inserting packets and interpreting the results. Our design explores how to invoke these devices efficiently by doing both: running parallel instances and state restoration.

3. We implement and evaluate a prototype of Plankton-neo, showing that it can catch actual problems that rise with the `iptables` software firewall on Linux, and that it can validate a multi-tenant data center with 196 routers and 64 tenants under 80 minutes.

CHAPTER 2: RELATED WORKS

We use some of the techniques used for general purpose image similarity [12], video surveillance [13] and measuring network complexity [14] as basis for our analysis module discussed in Section 4.1. Some of the techniques used for network analysis and anomaly detection involve traffic inspection [15, 16, 17, 18] however, the aim for our proposed technique was to not require any form of packet inspection (headers or payload). There is recent literature that focuses on specific kind of attack detection; like the problem of detecting Denial-of-Service attacks has been modelled as a computer vision problem [19]. We aim to keep our approach general enough so that any deviations in network state from the norm can be identified.

In terms of network inference, as well, we aim to propose an all-encompassing technique to detect network-wide policies and not limit the scope to specific network device types (for example firewalls) as is done in some prior works [20, 21, 22].

Network verification has been thoroughly studied in the past body of works as well. The earliest offline network verification techniques, such as Anteater [1] and HSA [2], evolved to real-time tools such as Veriflow [3], NetPlumber [4] and DeltaNet [23]. However, these systems do not by themselves verify configurations prior to deployment. We compared Plankton with existing configuration verifiers and compared experimentally to Minesweeper in Section 6.2.3. There are also more specialized verifiers, such as Bagpipe [10] which verifies BGP configurations for a single AS. Bagpipe avoids multi-AS dynamics, and does not incorporate other control protocols or data plane behavior. CrystalNet [24] performs an emulation of actual device virtual machines, and its results could be fed to a data plane verifier. However, this composition would not verify nondeterministic control plane dynamics, cross-layer dynamics, or temporal policies. Simultaneously improving the fidelity of configuration verifiers in *both* dimensions (capturing dynamics as in Plankton and implementation-specific behavior as in CrystalNet) appears to be a difficult open problem. Libra [25] is a divide-and-conquer data plane verifier, which is related to our equivalence class-based partitioning of possible packets. Topological symmetry in networks has also been explored in the context of optimizing verification [26]. Plankton's optimizations, though philosophically similar, have been designed to accommodate protocol dynamics, unlike existing work which applies these principles to data plane verification. Past approaches that used model checking in the networking domain have focused almost exclusively on the network software itself, either as SDN controllers, or protocol implementations [9, 27, 28]. We show how Plankton-neo can help detect real issues in softwarized networks that would be missed by these existing

techniques. Work on verifying dataplane software [29, 30] has focused on the correctness of the software component itself, rather than end-to-end correctness, which Plankton-neo is designed to verify. Architecturally, it shares many similarities with NICE [9], but targets middleboxes instead of OpenFlow.

CHAPTER 3: DESIGN

In this section we describe the complete design of our framework including all the components in the pipeline. The first component of our framework is an analysis module which ingests data plane state of the network at different time intervals and compares it with previous states of the network to inspect the evolution of the network over time. This enables us to quantitatively identify if the network was in an anomalous state at a given time relative to previously witnessed states of the network. This module uses some approaches from algorithmic information theory [31] in order to compute *similarity* of one network snapshot with another to identify if one snapshot is deviating significantly from previous ones, acting as a heuristic that signals that the network may need administrator attention. An in depth discussion of how this is done is provided in Section 4.1.

Following this module, we present the policy inference engine, which given the data plane state of the network at different time intervals, can output a *minimum* set of policies that define the network. The policies describe which set of network devices mostly communicate with which other set of network devices and over which packet sets. A lot of modern networking tools, like Veriflow [3], require the user to input some policies that help in describing the network invariants before they can start performing any operations on the network, for example verification of properties etc. When the network grows, it may be hard to exhaustively describe the policies of the network for any such tools to optimally operate on the network, and our policy inference engine aims to solve this problem. A detailed discussion of the techniques applied in the inference and summarization of policies is done in Section 4.2.

Finally, we present our network verification tool. It uses principles of equivalence partitioning along with *explicit state model checking* to reason about concurrency which, to the best of our knowledge, the state of the art fails to address. Our tool does not treat a network data plane or control plane in isolation, but, in fact, takes into account all the dynamically moving parts of a network. It understands control plane dynamics rather than simply using it to output a data plane and models the cross-layer events that may be woven together in an arbitrary order creating exponential complexity of the network. As a result of applying a suite of novel optimizations, including the use of some data structure optimizations and parallelization of the system, it makes the process of dealing with such a complex system efficient which we discuss in greater detail in Section 4.3.

3.1 ANALYSIS

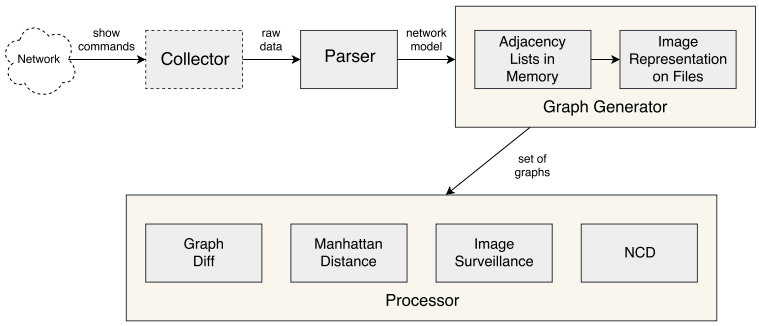


Figure 3.1: Analysis Module

The analysis module (an overview of which is shown in Figure 3.1) operates on a series of time ordered snapshots of the network. The snapshots can be of various forms and are processed so that we can obtain pair-wise reachability information between devices of the network. Having this reachability information, we perform temporal analysis on how reachability within the network changes over time and whether at some point in time the reachability of the network was deviating from the statistical norm.

3.1.1 Collector

As an entry point, we need to collect the network data over a period of time for which we use a collector. The implementation of this component was not in the scope of this work (hence shown in dotted-lines in Figure 3.1) and we design our pipeline such that various third-party collectors can be hooked and invoked from our pipeline. We omit details on the workings of these collectors but provide a high level overview of how this component fits within our analysis module and the overall pipeline. Depending on the network environment and context, (for example enterprise networks, Internet ASes etc.) the job of this component is to periodically gather information from the network. The nature of this information varies between different types of networks which may provide a different perspective on reachability within the network.

For instance, working in an enterprise network it would query all routing devices and collect the output of various *show* commands (for example *show interface*, *show ip interface*, *show access-list*, *show ip route* etc.) from each device in the network. Each of these commands describes a particular data plane feature from the stand-point of that particular device and this information is aggregated from all devices into a single *raw* network snapshot.

Another example of the collection phase is of gathering global BGP peering information from various vantage points including backbone routers on the Internet. This is done by using data collected by Route Views [32] which provides an archive of a vast amount of historical BGP RIB (Routing Information Base) dumps and BGP UPDATE messages exchanged between ASes.

3.1.2 Parser

After the collection of data is done from various heterogeneous network devices by different vendors, running potentially different versions of the firmware and operating system, we transform that information into a device agnostic format that we can use for our purposes. Here again, we do not completely reinvent the wheel and choose not to develop our own parsing component from scratch; instead we use the available tools and build on top of those. We used a third-party parser that outputs a standardized device model representation for each network feature of a routing device which works for most enterprise networks. Once the data has been parsed and modeled, we use it to generate pair-wise reachability between devices. This gives us information about the *packet sets* which this pair of devices can exchange in the given state of the network and we use this information to generate an edge list of reachability representing the entire network at this state.

For working with BGP data, we write a basic parsing component which ingests BGP RIB format and outputs an edge list. We make use of `libbgpdump` [33] which is a C library designed by RIPE NCC [34] to help with analyzing dump files produced by Zebra/Quagga or MRT. Since the Route Views archives are in the Quagga format, this library is able to open the dumps in a human readable format. We use Unix tools including `cut` and `sed` along with some basic Python scripts to parse Route Views data after having passed it through `libbgpdump`. An example input and output for BGP RIB is shown below along with a minimal break down of the parsing steps:

1. Opening the BGP RIB dump through `libbgpdump`, a sample line of the output looks like:

   ```
   TABLE_DUMP|1201829525|B|149.20.65.198|1280|12.6.247.0/24
   |1280 2828 7018 26456|IGP|149.20.65.198|0|30||NAG||
   ```

2. After the first pass of parsing this line using the command shown below, it becomes:

```
cut -d '|' -f 6,7 | sed -e 's#{\(.*\)}#\\1#' -e 's## #'


12.6.247.0/24|1280 2828 7018 26456
```

3. Next, we interpret the output of the previous step and turn it into an edge list such that if some AS, say AS1 can reach AS2 either directly or through one or more hops, there would be a direct edge between AS1 and AS2 in the edge list.

```
1280;2828;12.6.247.0/24
1280;7018;12.6.247.0/24
1280;26456;12.6.247.0/24
2828;7017;12.6.247.0/24
2828;26456;12.6.247.0/24
7018;26456;12.6.247.0/24
```

3.1.3 Graph Generator

We implement a graph generation module which ingests the standardized edge list format from the file system and produces an in-memory weighted graph corresponding to each snapshot. As the number of snapshots can be very large in some cases (discussed in Sectionn 5.2.1), it presents a few interesting challenges. Firstly, sequential graph generation for each snapshot is a time consuming process so we leverage multiple cores to do this in parallel giving us significant improvement in time taken to perform this action. Secondly, storing all historic snapshots of a network in memory is impractical given memory constraints in most typical compute environments therefore we keep only a *window* of these graphs in memory in a fixed size queue and follow a FIFO ordering to discard the oldest graph and bring in a new one if the queue is full up to the specified *window size*. For our comparisons and analysis, discarding some previous data is acceptable and we later discuss the impact of doing this in Section 5.2.1. For some of the techniques we will later discuss in Section 4.1.3 we require an *image* representation of each graph object which is also handled by this module. It creates the images and stores them in the file system for later access.

3.1.4 Processor

The last stage of the analysis module is the processing of the generated graphs using various techniques which will be discussed in Section 4.1. Each of the specific approaches we use including Normalized Compression Distance (Section 4.1.3) on images and Graph Diff (Section 4.1.1) are submodules which consume a set of graphs and output numeric metrics corresponding to each snapshot.

Note that the analysis module is *online* in nature and any real time stream of network data can be attached to this module. It will store a past window of network states in memory and compare with the latest snapshot from the stream. This makes it highly practical for use in production network environments.

3.2 INFERENCE

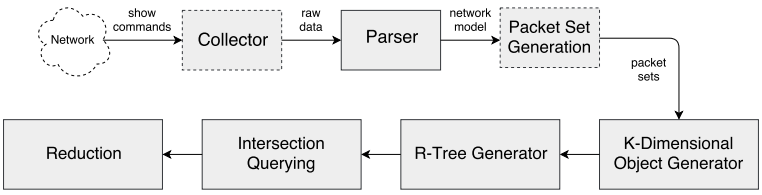


Figure 3.2: Inference Module

The inference engine is designed to load the data plane information from each snapshot of the network and eventually insert that information in an R-Tree (the reasons for using this particular data structure and how it is used are discussed in Section 4.2). The particular format of the data plane that we use here is a list of packet sets that each pair of devices in the network was able to exchange at a given snapshot. There are existing tools [3] that can generate the particular packet sets which can be exchanged at particular snapshots by processing the data plane state of the network so we do not discuss those specific details here and make our module extensible to be able to consume this particular information from other existing tools. We model the packet sets into **K-dimensional objects** (described later in Section 4.2) that are later inserted into an R-Tree which is then used to query for intersecting packet sets and eventually compute the frequently occuring packet sets in the network. We implement this module entirely in C++ in favor of efficient performance and make use of OpenMP [35] to parallelize various sections of the code including the loading of snapshots, inserting them into an R-Tree and finding the intersections for various packet sets.

After we have computed the frequent packet sets (which are the pair-wise low-level policies) we reduce this informaiton and aggregate it such that it becomes easier to comprehend and pass on to other tools. This is the final stage in our inference engine as can be seen in Figure 3.2.

3.3 VERIFICATION

3.3.1 Plankton

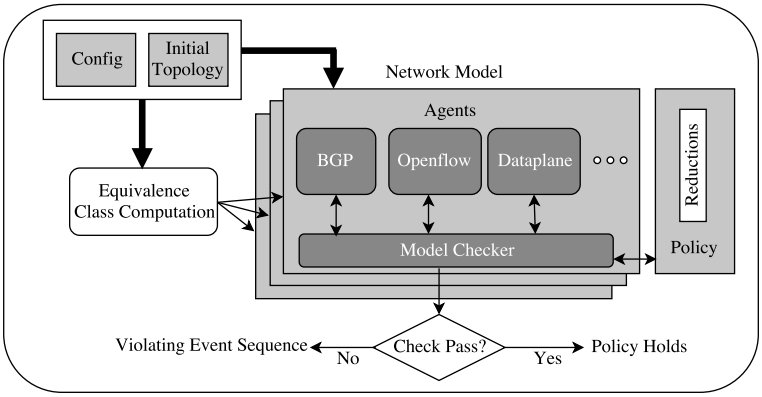


Figure 3.3: Plankton

Now we discuss the verification tool we implemented in this work, Plankton, and describe various design choices that were made while implementing it.

The design decisions made in Plankton are aimed at tackling the challenges of cross-layer dynamics, packet diversity, and event ordering. As illustrated in Figure 3.3, Plankton's model of the network is a collection of agents, each executing independently, making changes to a shared global state of the network. The agent-based model allows Plankton to reason accurately about the dynamics not only within the control plane, but also between the control plane and the data plane. Given a configuration, Plankton formally analyzes its correctness by interpreting it over the combined network model, and examining the behavior of the model. To do so, Plankton uses an explicit-state model checker. The model checker is designed to exhaustively search through the various interleaved executions of the agents, and find any potential policy violations. However, naively modeling the individual agents and passing them to a model checker does not suffice. Due to the large number of event orderings, even a well-designed and mature model checker cannot scale to networks the size of real world data centers or campus networks. In fact, past attempts to use model checking even in settings significantly more restrictive than Plankton have failed to scale

beyond networks of approximately 20 devices [9, 27]. Plankton tackles this problem through a collection of highly effective optimizations. These optimizations are designed to either reduce the overall number of event orderings to be checked without affecting correctness, or to increase the efficiency of the exploration so that more orderings can be covered.

3.3.2 High-coverage testing of softwarized networks

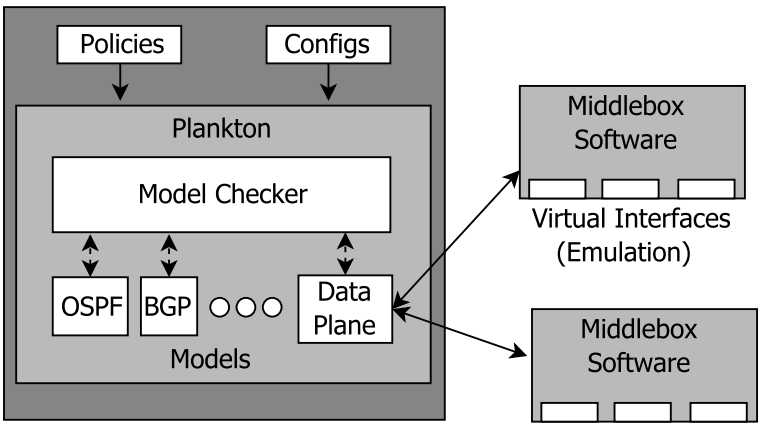


Figure 3.4: Plankton-neo

Figure 3.4 illustrates the organization of Plankton-neo. It builds upon Plankton, which uses model checking to formally verify network models. In Plankton, each logical component of the network, such as the data plane, control plane protocols and the environment are modeled as agents, which operate together to change the state of the overall network system. These agents are written manually to simulate the behavior of the actual network components, but with one key difference — they are designed to operate on *equivalence classes* of packets rather than a single packet at a time. For example, the model for a stateful firewall may define its behavior with respect to *request* and *reply* classes. The control plane models perform route computation for each equivalence class separately. The model for the dataplane forward a single *symbolic* packet through the dataplane, based on the forwarding rules. The overall state of the network is defined as the combination of the states of component models, and every time a step is taken within a model, the overall system is considered to have made a state transition. A step may be a link failing, a routing table change, forwarding of the symbolic packet from one device to the next etc. Assuming that the models are faithful to the actual network behavior, Plankton can be used to verify a wide range of correctness policies, including protocol convergence, failure tolerance etc. The actual verification of the policy done by an off-the-shelf explicit-state model checker, which exhaustively

explores every relevant execution of the overall network system, and locates any sequence of events that can violate network correctness. As the name suggests, the explicit-state model checker generates and checks each relevant state of the system separately.

The design we described in Section 3.3.2 uses a separate set of virtual interfaces for each middlebox in the network. Alternatively, we can also have multiple middleboxes sharing the same set of emulation setups. This is possible because the model checking algorithm only checks the behavior of one middlebox at any given time. When the packet needs to be injected into a particular middlebox, we run that middlebox over the virtual interfaces, and use update replay to put the system in the required state. The performance implications of this are discussed in Section 6.2.4.

We implemented a simple version of our technique over Plankton, with support for Linux-based middleboxes. For emulating *Virtualized Network Function* (VNF) devices, we create `tap` interfaces, and create separate routing tables in the kernel for the emulated middleboxes. When the verification algorithm invokes packet injection, the representative is sent into the appropriate `tap` interface. A special data payload is used to distinguish the injected packet from any other packets that may originate from the kernel. If the packet does not make it to any of the `tap` interfaces withing a configurable timeout, it is assumed to be dropped.

CHAPTER 4: TECHNIQUES & IMPLEMENTATION

4.1 ANALYSIS

This section describes several network analysis techniques that were explored and implemented in the framework we propose.

For analyzing networks over time, we model them as graphs of two types. First, we model the adjacency of network entities *as is* in the context of the network we are analyzing. For example, if we are running this framework on a BGP network (let's say on the AS level of the Internet) then the adjacency graph would represent the direct BGP peer relations between the ASes as we see them in the BGP RIB and BGP UPDATES. As another example, if we are working on enterprise networks then this would be direct hops between routers; this could be the result of static route configurations done by network administrators or running routing algorithms like OSPF.

Second, we model the reachability of network entities as a result of running basic graph algorithms on top of the adjacency graph. Initially we implemented Floyd-Warshall algorithm [36] for computing shortest paths between all possible pairs of nodes in the graph, however the time complexity for doing this is $O(N^3)$ in the number of nodes (N) in the graph which posed a lot of scaling challenges. Given the size of data we were dealing with, for example a set of AS level graphs with roughly 7716 nodes and 733 such graphs to process, this particular approach seemed unfeasible due to the enormously large amount of time it would require. Since we did not expect to have negative cycles in our graphs, we decided to go with using Dijkstra's algorithm [37] for computing the single-source shortest path to all nodes from each possible source. Each run of this takes $O(ElogN)$ time in the number of nodes (N) and the number of edges (E) in the graph. However, running this for all sources increases the time taken to complete significantly so this approach was also discarded. We finally decided to use Bread First Search [38] from all source nodes in the graph and used this to generate a reachability graph which had an edge between two nodes, say $\mathbf{A} \rightarrow \mathbf{B}$ if there was at least one path in the network to reach **B** from **A**. We do not model protocol specific reachability in this approach (for example we don't take into account BGP peering relationships like customer/peer/provider etc.) for now.

4.1.1 Graph Diff

As a baseline, we treat the generated graphs using existing techniques which work directly on these graph representations. We partially follow the approach proposed by Bunke et al. [39] in this regard. Given a time ordered series of graphs G_1, G_2, G_3, ... , G_n we compute pair-wise distances $D(G_i, G_j)$ between these graphs such that $i > j$ for all values of i and j that obey this constraint. This translates into computing distances between all (or a subset of all) graphs that precede graph G_i in time. The more this value turns out to be, the greater the difference between the two graphs. We define the distance metric as follows:

$$D(G_i, G_j) = \mid N_i \mid + \mid N_j \mid - 2 \mid N_i \cap N_j \mid + \mid E_i \mid + \mid E_j \mid - 2 \mid E_i \cap E_j \mid$$
$$\text{where } \mid N_x \mid \text{ is the number of nodes in graph x}$$
$$\text{and } \mid E_x \mid \text{ is the number of edges in graph x}$$

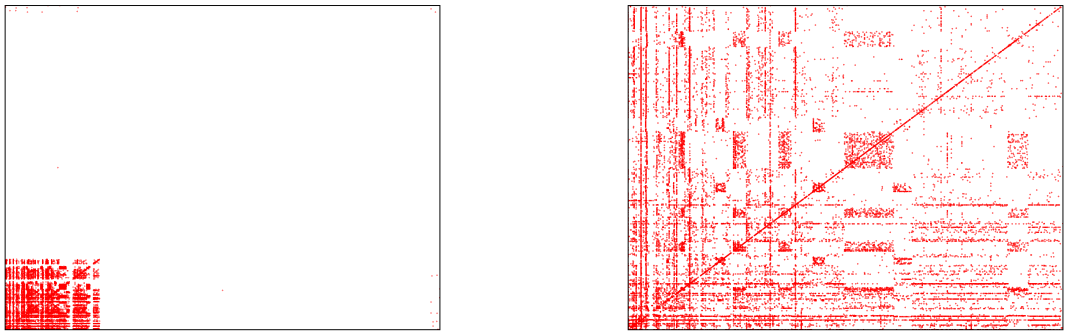
Computing this **graph diff** measure helps us visualize the evolution of networks over time and any aberrations in this trend are indicators of sudden changes in the network state which may need administrator attention. We compute this metric over a window of past network states and compute the average over all those comparisons.

4.1.2 Image Representation of Network Reachability

We generate an adjacency matrix for each graph object and turn this into an image by representing reachability between a pair of devices by the presence or absence of a pixel in the appropriate position of the image. Each node would have some coordinate on the x-axis of the image and some coordinate on the y-axis of the image. If two nodes, say **A** and **B**, have some notion of reachability in the graph such that **A** can "reach" **B**, then on the image we would produce a pixel to represent this information. Assuming that the position of **A** on the x-axis and y-axis is *posAx* and *posAy* respectively and the position of **B** on the x-axis and y-axis is *posBx* and *posBy* respectively then there would be a pixel on the coordinate (*posAx*, *posBy*) in the image. If the notion of reachability is bidirectional then there would also be a pixel in coordinate (*posBx*, *posAy*) in the image.

In some datasets, if we map the devices with their default identities on to our image space, the resulting images may be really sparse and may consume a lot of space on disk while storing the images causing more expensive I/O operations while processing them. For example, using default AS numbers poses this problem as the entire AS number range has

not been used but AS numbers lie all across the space of the 16-bit number. Therefore, the resulting matrix and the corresponding image is extremely sparse taking up more space than is necessary. We apply sparsity reduction on this by mapping this AS number space range on to a shrunk down version by re-assigning AS numbers to a smaller (*minimum*) range such that all numbers are used. In the case of doing retrospective analysis, we do this by taking a union of all the nodes present in the dataset. From this resulting set, we reassign each node's ID between 0 and the numbers of nodes in this union set, say N. This brings down the matrix size and image sparsity significantly. The result of this can be seen in Figure 4.1 where the network adjacency images are shown before applying this sparsity reduction optimization and after this. This has significant performance benefits as it brings down the image size per network snapshot considerably; for instance in the network snapshot shown the image size goes down from $2.3MB$ down to $1.7MB$ (26% decrease in file size). This results in a lower memory footprint of the application (if some images are kept in-memory) and also results in faster processing when comparing images and compressing images as will be discussed later. From our benchmarks (discussed in Section 6.2.1), it is evident that storing all images (network snapshots) in memory is not the bottleneck, it's in fact the time taken to compare that grows faster than memory consumption.



(a) (b)

Figure 4.1: Showing the image representation of network adjacency from AS-733 dataset as of 21st November 1999. (a) shows the sparse matrix version with some data points seen further away towards top right/bottom right/top left and (b) shows the optimized matrix version after sparsity reduction.

4.1.3 Image Similarity

As described in Section 4.1.2 we can represent the binary reachability information at each snapshot of the network using an image. We then explore some techniques widely used in computer graphics to compute similarity of images, which translate into similarity in the nature of reachability of devices in the network between a set of snapshots. This metric helps us identify how much the reachability properties of the network vary in the current snapshot being evaluated compared with the past set of snapshots that we are comparing with. It proves to be a good network-wide statistical indicator of how the network compares with a past state and given the usual nature of networks we explore, the variation in between snapshots does not adversely affect the inferences we make from employing this technique. The result of using this technique on some real world data will be discussed in Section 5.1.

Normalized Compression Distance

Lots of computer graphics problems have to use image similarity metrics for rendering, determining visual quality of frames and so on. Thus, a vast body of work is present which tries to address this but most of them work only for specific use-cases. An example is that of using Mean Squared Error (MSE) which compares pixel-to-pixel for computing a certain distance. This may not work in all cases, for example if the pixels are all shifted by a constant factor in one image. To counter such challenges and have a more general robust technique some work has been done in the field of algorithmic complexity and a new measure of similarity has been proposed named **Normalized Compression Distance** (NCD) [40] based on **Kolmogorov Complexity** [41]. We define Kolmogorov Complexity of a string w with respect to L, denoted $K_L(w)$ as the shortest computer program on a universal computer (such as a universal Turing Machine) written in language L which produces w as output. The conditional Kolmogorov Complexity, which is of interest to us for comparing objects, with respects to a string x, denoted $K_L(w \mid x)$ (spoken w given x, as in probability theory), is the length of the shortest program which, when given x as input, outputs w [42]. Although Kolmogorov Complexity is not computable, we will discuss in the following paragraph how it can be approximated leading to the NCD.

Bennet et al. [43] define **information distance** between two (potentially unequal sized) binary strings, as the length of transforming either string into the other (both ways). Li et al. [40] propose a normalized version of the same metric called **similarity metric** which like the previous metric, is also universal. As mentioned earlier, Kolmogorov Complexity can only be approximated, we use real-world compressors to compute the NCD and use it

to compare images. As evaluated by Vázquez et al. [12] NCD works well on images under certain constraints which we follow in our framework and discuss later on. Now we formally present the definition of **Normalized Compression Distance** as:

$$NCD(X,Y) = \frac{C(X,Y) - \min\{C(X), C(Y)\}}{\max\{C(X), C(Y)\}}$$

where $C(F)$is the size after compression of a file F

and XY is the concatenation of files X and Y.

Some other things to note about NCD (in context of our usage):

1. The values of NCD lie between [0,1.1]. Typically, similarity measure lies in the range [0,1] but due to real-world compressor imperfections, NCD observes values a little higher than this. Lower values signify a high similarity and higher values signify lower similarity between the two given input objects.

2. Before using a compressor for NCD computation we need to ensure that it obeys the properties analyzed by Cilibrasi et al. [44]. Most real-world compressors (like **BZIP** [45], **LZMA** (Lempel Ziv Markov chain algorithm) [46] etc.) obey these.

Now that we have defined what the similarity metric used in our technique precisely is, we discuss how we use this to analyze real-world networks. We operate on a time ordered set of network images as described above and compare each of these images with a *window* of previous images. For example, if we are at time T_{55} and the window size, which we denote as W is equal to 10, then the two metrics we compute for the network at time T_{55} are as follows:

$$\begin{aligned} NCD_{T_{55}} &= \max\{NCD(I_{T_{55}}, I_{T_{54}}), NCD(I_{T_{55}}, I_{T_{53}}), .., NCD(I_{T_{55}}, I_{T_{45}})\} \\ AVG\_NCD_{T_{55}} &= \text{mean}\{NCD(I_{T_{55}}, I_{T_{54}}), NCD(I_{T_{55}}, I_{T_{53}}), .., \\ &= NCD(I_{T_{55}}, I_{T_{45}})\} \end{aligned}$$

where I_{T_x} is the image corresponding to the network at time x

The first metric reports the maximum dissimilarity from among a window of previous network states and the second metric reports how much the current state is dissimilar from

among a window of previous network states on average. Having both these metrics, we are able to visualize, using trend lines, how the network evolves over time. We use various parameters for the window size and compare the results in Section 5.5. The assumption that is satisfied by most networks is that given an appropriate sampling period, the network would not significantly deviate its reachability state on the scale of that sampling period. If either of these metrics report otherwise (ignoring the minor deviations) then a potential abnormality may have occurred (or in the process of occurring) and thus, manual intervention and inspection may be required.

As part of this technique, we also employed a technique for computing Manhattan Distance between the images but the results of that were similar to the Graph Diff approach discussed earlier so we omit the details for that. We also implemented an approach which is a slight variation of NCD used in [13] the field of anomaly detection in surveilance of video feeds however, the general trend of the results obtained using this formulation and the implementation details are similar to the NCD technique therefore we also omit the details for that in the interest of space.

4.2 INFERENCE

In this section we describe how our policy inference engine works. When we talk of *network policies* we mean sets of conditions, constraints or properties that the network obeys at some given state. These policies dictate access control within network and control the flow of traffic between different entities.

As networks evolve over time, they undergo changes in terms of addition and removal of devices, rules, configurations etc. To keep track of all these events in the networks's history proves to be a cumbersome task because the amount of information associated with such events can be very large making processing that information a challenging problem. On the other hand, this information *is* required to some extent in order to perform crucial actions on the network like invariant verification and policy-based management to name a few. The underlying assumption behind the proper execution of all these actions is correct information about the network which we want to verify or manage. This brings us back to the problem of how we can retain this information as the network itself grows larger and keeps getting more complicated. In this section, we aim to propose a solution to inferring valuable information about underlying network policies from a series of snapshots which can be used for performing the aforementioned actions.

As was being done in the analysis stage, we ingest data plane state of the network at different points in time. However, the output from this stage of the pipeline is a set of policies

that define the network. This includes *low-level policies* that tell us what packet sets are exchanged between pairs/sets of devices and also *high-level policies* that are an accumulation of *similar* low-level policies. As a first step to our approach, we aim to identify the most frequently exchanged packet sets that were exchanged between pairs of devices and we do this for all pairs in the network. A packet set is modeled as a **K-dimensional shape** where each dimension is one field of the packet. For example, if we are considering the following fields in the packet (from different layers of the OSI Model [47]):

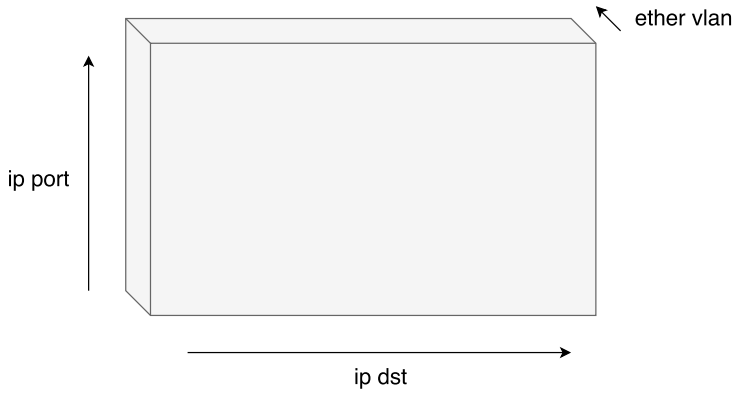


Figure 4.2: K-Dimensional Rectangle modeling three fields of a packet set

1. ip dst
2. ip port
3. ether vlan

then each of these fields would be one dimension of our K-dimensional shape (which we will call **KD-Rectangle**) that can be visualized as shown in Figure. 4.2.

Populate the R-Tree with all packet sets exchanged between devices D_1 and D_2 and call it R (4.1)

Initialize set Q equal to all packet sets in R (4.2)

$Q_{res} = \{\}(HashMap)$ (4.3)

Repeat steps 4.5 to 4.10 until Q is not empty (4.4)

$Q_{next} = \{\}$ (4.5)

For all packet sets $p \in Q$, query for the intersecting packet sets using the R-Tree R (4.6)

Assume that for packet set p the intersecting packet sets are stored in set S such that $S \subset Q$ (4.7)

Q_{res}.insert(p, length(S)) (4.8)

For each packet set s in S, find the intersecting region between s and p called q (another packet set); $Q_{next} = Q_{next} \cup \{q\}$ (4.9)

$Q = Q_{next}$ (4.10)

Sort Q_{res} by value (4.11)

The algorithm shown previously (Algorithm 4.1) is an iterative approach to computing the overlapping packet sets which appear most frequently across time between a pair of devices. We start with the actual packet sets observed and insert all of them into an **R-Tree** [48] data structure. The reason for using this particular data structure is that it is very efficient for spacial access methods, that is for indexing multi-dimensional information such as our K-Dimensional packet sets in space where time is an added dimension (signifying at what particular times each of those packet sets was observed as being exchanged between a given pair of devices.) The fundamental idea behind R-Trees is that they group nearby objects and represent them using a minimum bounding rectangle. These minimum bounding rectangles indicate whether or not a search query should search in a given subtree making this process efficient. Searching time complexity for our practical purposes is on average logarithmic in the number of nodes in the tree.

For each of the packet sets we have inserted in the R-Tree, we query the R-Tree to give us the overlapping packet sets with the querying packet set. This, in the context of the problem we are solving, gives us other packet sets that were seen at another time which have

at least some intersecting region with the current packet set being queried. Having access to these overlapping packet sets, we identify which region of each of these packet sets has an overlap with the queried packet set and use this overlapping part of the packet set (or the *inner packet set*) in the next iteration to look for further intersections. Since we are trying to find the most frequently exchanged packet sets, we keep narrowing down the size of our packet sets and search for more *dense* or highly overlapping regions in our K-Dimensional space and keep track of the number of intersections of each packet set in our HashMap called Q_{res}. Ultimately, when no more subsequent inner packet sets are found and we have scanned all intersections, the querying iterations end. At this point, we sort Q_{res} by value and report the top k packet sets. This terminates the algorithm and we have extracted the low-level pair-wise policies which have remained the most consistent in time. Of course, the most frequent policies may not be the most up-to-date policies (since the administrator may have just recently updated some policies which are still not frequent enough to be captured by this algorithm) in which case we can put constraints on how far back in time we need to query. We can also prevent the addition of the stale packet sets in the R-Tree at the beginning of the algorithm. We may also choose to apply a filtering pass on the frequent packet sets that the algorithm reports in the end to prune out those packet sets which have some intersection with our *blacklist* of packet sets.

4.2.1 Reduction

Next, we use the pair-wise packet sets computed above (which we call the low-level policies) to aggregate them into a more succint form of information that can be fed into other tools or used for human inspection of the network policies.

The pair-wise policies computed earlier are of the following form (where A, B, C, ..., F are devices and PS1, PS2, ... are unique packet sets):
{(A $\rightarrow$ B: PS1), (A $\rightarrow$ B: PS2), (A $\rightarrow$ B: PS3)}
{(A $\rightarrow$ D: PS1), (A $\rightarrow$ D: PS2), (A $\rightarrow$ D: PS3)}
{(C $\rightarrow$ D: PS1), (C $\rightarrow$ D: PS3)}
{(E $\rightarrow$ F: PS1), (E $\rightarrow$ F: PS3)}

On this, we apply the first phase of aggregation which is kind of a like a reverse index, making a list of pairs for each unique packet set. The result is of the following form:
PS1 = {A $\rightarrow$ B, A $\rightarrow$ D, C $\rightarrow$ D, E $\rightarrow$ F}
PS2 = {A $\rightarrow$ B, A $\rightarrow$ D}
PS3 = {A $\rightarrow$ B, A $\rightarrow$ D, C $\rightarrow$ D, E $\rightarrow$ F}

Now we apply the second phase of aggregation which groups together similar pairs of

devices. This is done by searching for the *longest intersecting set of device-pairs* from the above lists. For each of these sets, we enumerate the packet sets that they were picked from. For example, considering the above list we see that the following set of pairs is the longest set that is present in at least two lists (namely PS1 and PS3, in the previous example):
{(A → B), (A → D), (C → D), (E → F)}

Thus, we obtain a grouping from the above set as follows:
{A → B, A → D, C → D, E → F: PS1, PS3}

We keep doing this until we are unable to find any intersecting sets at which point we just merge the remaining results into our grouping. Thus, the final groupinng of the previous example would look like:
{A → B, A → D, C → D, E → F: PS1, PS3}
{A → B, A → D: PS2}

This results in a minimum bundle of devices obeying the same policies and we have reduced the information (or data units) into a much smaller output size which is much more comprehensible and can describe the network policies using this minimum set of data units.

4.3 VERIFICATION

Now we discuss the network verification tool in detail including discussion of packet equivalnce classes, the explicit-state model checker used by the tool, the roles performed by the various agents which model the various events executed in the network and how the policies are verified as a result of all these components. We also discuss the hybrid approach of using explicit-state model checking along with emulation-based testing and how we implement that for our framework.

4.3.1 Packet Equivalence Classes

The first form of equivalence that Plankton relies on for scalable network verification is packet equivalence. Plankton slices the network into Packet Equivalence Classes (PECs), and explores the dynamics of each PEC (or a small number of PECs) separately. PECs have been defined in various ways in existing verification literature. Dataplane verifiers such as Veriflow [3] define them over data plane rules, whereas the configuration analysis tool ERA [6] defines them over control plane message paths. Since Plankton checks not just the current network state but also many possible future states with various changes, we define PECs to guarantee that two packets that are in the same class in the original network state continue to be so even after any topological or other changes that may be explored by Plankton. There

are multiple ways in which we can ensure this, and Plankton's design does not mandate a particular way of computation. One possible approach is to consider the prefixes described in the network-wide protocol configuration, and compute boolean combinations (through intersection & set difference) of these prefixes. This approach is similar to how a data plane veriifier would compute data plane equivalence classes. Indeed, such a computation produces a similar number of PECs as a data plane verifier such as Veriflow. Nevertheless, this set is finer than necessary for ensuring Plankton's correctness. In order to obtain a more compact set, we can unify those PECs which have the same configuration network-wide. For example, multiple prefixes originating from the same router can be treated as a single PEC, if their behavior is identical throughout the network. This approach produces a relatively small number of PECs, which are used to define the input to the explicit exploration component. Our experiments are carried out using PECs defined in this manner.

For each PEC, Plankton defines a software model that encompasses the entire network and its environment. It is these models that are checked by the second component of Plankton, the model checker.

4.3.2 Explicit-state model checker

The explicit state model checker SPIN [49] provides Plankton its exhaustive exploration ability. A model checker for software programs, SPIN verifies models written in the *Promela* modeling language, which has constructs to describe possible non-deterministic behavior. Plankton's network model is essentially a software copy of the network written in Promela. While the model is defined for each PEC, SPIN verifies only a small number n of them at any given time, depending on the policy under verification. For most policies, n is 1, whereas some policies may require two, or more. An example of a policy that requires more than one PEC to be modeled is **stateful reachability**, which enforces that if a request is delivered, the reply gets delivered too. The model consists of a collection of agents, each responsible for one component of the network system. An agent may represent a protocol such as OSPF or BGP, the data plane which forwards traffic, the failure agent that changes the topology etc.

Plankton's model of the network executes **iteratively**, making multiple non-deterministic choices in each iteration. First, an agent is picked non-deterministically from among those that are ready to execute. This is followed by non-deterministic choice of the network device or link where the agent shall make changes. Further non-deterministic choices may be made as applicable in the protocol. For example, the forwarding agent non-deterministically decides the next-hop to forward to, when multipath load balancing is in use. As we shall dis-

cuss later, for each non-deterministic selection, the possible set of choices is restricted by the policy being verified. While such non-deterministic choices occur in simulation based analysis also, simulation explores network evolution only along one non-deterministic execution path. The model checker in Plankton explores every possible non-deterministic execution path, checking for policy violations along each of these paths. As the name suggests, the explicit-state model checker performs this exploration one state at a time. This distinguishes it from *symbolic* model checkers, which explore sets of states simultaneously. Unlike symbolic model checkers, explict-state model checkers do not require the transition relation of the system to be computed a-priori. SPIN's exploration of program execution proceeds as a traversal of the state transition graph. In the case of plankton, this traversal is made practical through efficient state-keeping, and optimizations both inside SPIN (such as *Bit-state Hashing*) as well as our own additional ones. SPIN checks the specified policy over the paths that are observed while traversing the graph - an approach known as on-the-fly model checking [50].

Agents

Agents in Plankton's network model define the behavior of the various moving parts in the network. In addition to the environment (topology changes) and control protocols, Plankton defines an controller agent and a data plane agent. Each agent defines state variables that become part of the overall system state, and actions that will update these state variables. Protocol agents for standard control protocols are semantically quite straightforward. For instance, the BGP agent defines BGP tables, import and export of routes, and the BGP decision procedure. The OSPF agent performs shortest-path routing. The controller agent installs an ordered stream of updates from an external entity to each switch, chosen in non-determinsitc order. This agent is designed to capture the semantics of SDNs, manual installation of static routes etc.

The agent that allows Plankton to check forwarding issues, including middleboxes, is the data plane agent. For each equivalence class, the data plane agent moves exactly one packet through the network. This is essentially a symbolic packet, intended to capture how the data plane forwards and is affected by traffic. The forwarding agent adds two variables to the overall network state: the starting point of the packet, and its current location. When a packet reaches a dynamic data plane device, the data plane agent performs any updates that are made by the device to the actual data plane. By interleaving execution of the forwarding agent with others, Plankton can detect policy violations that are not present in any single data plane state, but are experienced by specific packets.

Policies

Plankton allows a wide range of policies to be checked over individual packets, data plane states or the control plane. Policies that describe the forwarding path of the packet are expressed as *Linear Temporal Logic* (LTL) formulae. LTL is a logic system for describing execution paths of finite state machines, and is natively supported by SPIN. LTL includes operators to describe temporal behavior, such as G for *Always*, F for *Future*, and U for *Until*. These operators are used to describe expected packet behavior in Plankton. For example, loop freedom is expressed as `G(packetLocation != startingPoint => (G (packetLocation != startingPoint)))` which stands for *Once the packet location is not equal to the starting point, it will always be not equal to the starting point.* Since the starting point is non-deterministically chosen by the forwarding agent, this policy covers any forwarding loop in the data plane. Policies that do not involve interleaving of the control plane and the data plane can be checked without a forwarding agent. The data plane state is part of the overall network state, and a graph algorithm over the current data plane is sufficient to check most of such policies. This approach, which is significantly faster than having a forwarding agent, is similar in spirit to Veriflow. The policy definition will include a callback that accepts the current network state as an argument, and after checking the policy, sets appropriate flags to represent the outcome of the check. However, checking the policy efficiently requires more than a fast analysis algorithm. Plankton relies on the policy definition to supply any optimizations that may be safely applied to the exploration process, without compromising the correctness of the check. In other words, when faced with a set of non-deterministic choices, the policy must convey to Plankton which of those must be necessarily checked, and which may be skipped, without missing a potential violation. This responsibility is assigned to the policy because in general, a policy may require that one specific network state is not reached by the network, and a fully policy-agnostic optimization scheme may eliminate that particular state from the exploration. In general, policy-agnostic optimization would put restrictions on the set of policies that may be checked using Plankton. Hence, we choose against that option.

Recall that Plankton's network model performs successive iterations of agent selection, location selection and agent execution. For each of the non-deterministic selection points in an iteration, the policy needs to supply to the model checker a set of options to choose from, through callbacks. Specifically, callbacks are defined for the following:

- **Protocol selection:** While a large number of protocols may be prepared to execute at a given point of time, only a subset of them may be relevant to producing an interesting execution path. The purpose of this callback is to prune the set of non-

deterministic choices to those that are indeed relevant. The callback is expected to set flags corresponding to individual protocols, indicating that they may execute. Plankton non-deterministically picks (in other words, exhaustively explores) one protocol from among them to execute.

- **Location selection:** Similar to protocol selection, this callback is expected to prune the set of nodes/links where the chosen protocol is allowed to make a change.
- **Agent-specific callbacks:** Individual agents may define additional callbacks, to optimize their execution. For example, the forwarding agent defines a callback to prune the possible set of next-hops when there are multiple possible next-hops are defined in the data plane (like in ECMP).

4.3.3 Hybrid: explicit-state model checking + emulation-based testing

Incorporating real software

In Plankton-neo, we leverage Plankton's explicit-state model checking framework, but adapt it for *Network Functions Virtualization* (NFV). While much of the network is still represented using models operating on symbolic packets, middlebox software execute in their original form. For each middlebox in the network, a "virtual device" is created, which consists of virtual interfaces in one-to-one correspondence with the actual interfaces on the middlebox, and running the same software. The configuration supplied to the middlebox is updated to operate over the virtual interfaces rather than the original physical interfaces. Since middlebox software can work only with concrete packets and not equivalence classes, a fully instantiated *representative* is picked for each equivalence class. The difference between a symbolic packet and a representative packet is subtle, but important. A symbolic packet is a logical entity, that merely denotes an equivalence class, whereas a representative packet is a real packet that can be processed by network devices, chosen from the many packets that constitutes the equivalence class. We elaborate on the computation of equivalence classes and the selection of the representatives later.

Similar to Plankton, the model checking algorithm exhaustively explores the various execution paths of system, for each equivalence class. The distinction from Plankton lies in behavior of the dataplane model. While it still defines a single symbolic packet and moves it through the network, when the symbolic packet reaches a middlebox, the representative packet for the equivalence class is injected into the appropriate interface of the emulated copy of the middlebox. Then, the fate of the injected packet is observed, and the dataplane

model interprets the observation as an action performed on the symbolic packet. In essence, each middlebox defines an "API" that allows verification algorithm to query for the outcome of packets reaching one of the interfaces. Using such a system, we can verify a variety of policies about end-to-end correctness of the network. Perhaps the most relevant are policies that pertain to how the network changes its behavior in response to traffic. For example, a network with a web-cache may state that *No HTTP requests are sent to the server more than once*. The model checker that we inherit from Plankton natively supports such *temporal policies* as was discussed in Section 4.3.2.

Saving and restoring middlebox state

As the model checking algorithm exhaustively searches through possible executions of the network looking for policy violations, it will be required to perform packet injection into various middleboxes many times, under various hypothetical scenarios. Each time such an injection happens, the intention of the algorithm is to observe the fate of the packet if it was to reach the middlebox under the specific circumstances that the algorithm has contrived. So, we require the virtual middlebox to match the algorithm's intended state before the packet can be injected. In order to do so, we implement the network model in the following way: In addition to the middlebox software running on the emulated interfaces, for each middlebox, we define a model within the verifier itself. The purpose of the model is to keep track of the state changes that occur to the middlebox during the execution of the verification algorithm. Specifically, the state of this model is defined as the initial state, plus a list of all updates that have been made to the middlebox. We define the model to track two types of updates: any changes to the routing table, and any packets that were previously injected, along with the interfaces where the injection happened. When a new packet needs to be injected into a middlebox, we first match the emulator state with the one in the model — by first restarting middlebox software with the initial configuration, and then replaying all the historical updates that have supposedly happened in the past. This approach of putting a middlebox into a desired state is attractive, because it does not require any knowledge of the internal workings of the software. It is also more practical than snapshotting the virtual memory of the middlebox in order to store its state. However, it does assume that the software is deterministic, and has no dependency on timing. In other words, starting from an initial state, replaying the same sequence of updates is guaranteed to put the middlebox in the same final state.

CHAPTER 5: EVALUATION

5.1 NETWORK EVOLUTION

We use our image representation of adjacency and reachability graphs of the networks to visualize how the networks evolve over time. These, of course, are not any quantitative measure of reasoning about the network over time but still prove to be an interesting by-product of the techniques we apply for further analysis. Figures 5.1 and 5.2 show how dense the AS connectivity gets over time showing the addition of more entities in the network.

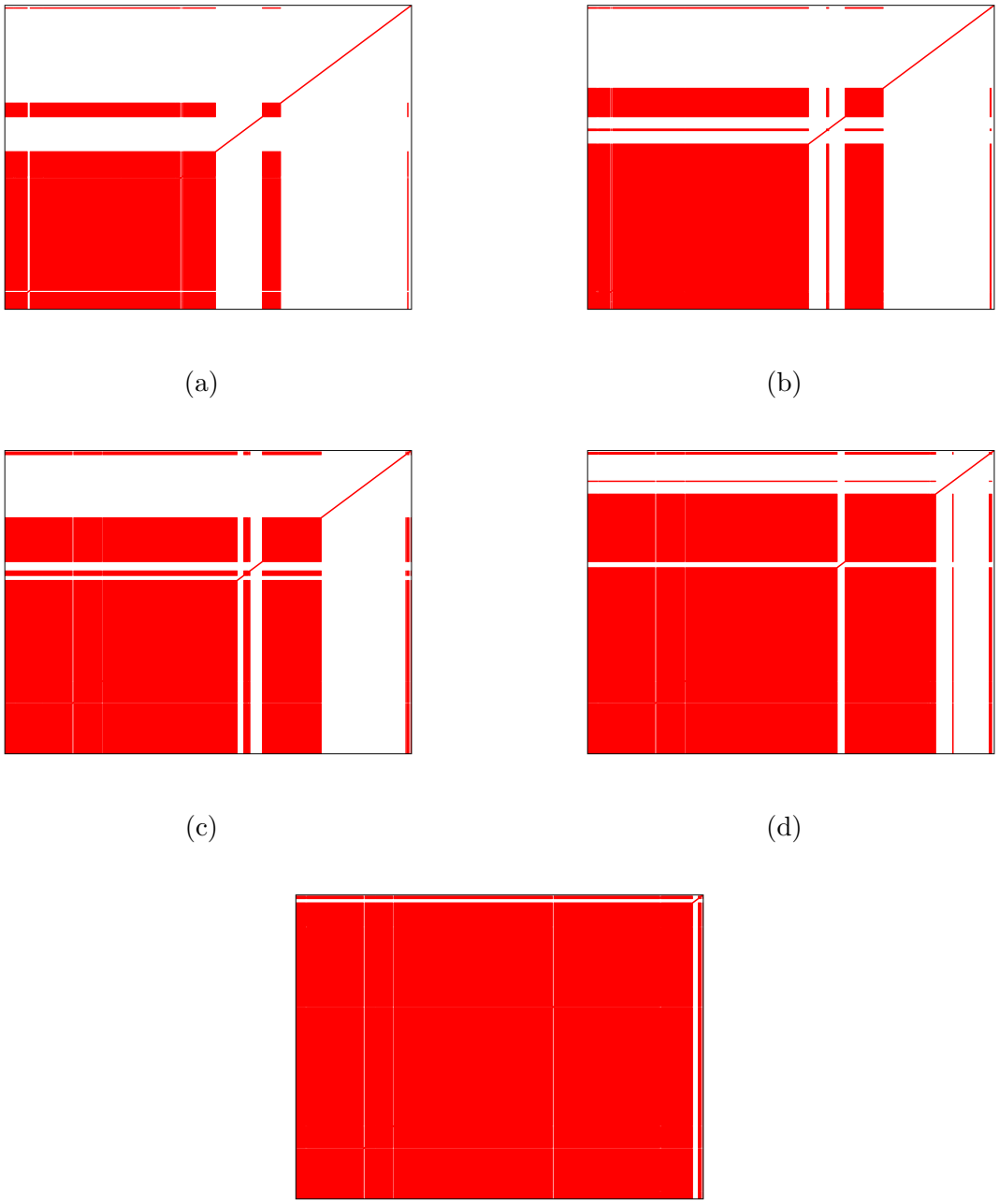
5.2 ANOMALY DETECTION

5.2.1 Normalized Compression Distance

One of the publicly available datasets that we use in this work is from the Stanford Large Network Dataset Collection [51] called **as-733** which is a dataset representing communication networks of who-talks-to-whom from the BGP (Border Gateway Protocol) logs. The data was collected from University of Oregon Route Views Project [32] data and reports. The dataset contains 733 daily instances which span an interval of 785 days from November 8 1997 to January 2 2000 mostly on 24 hour intervals.

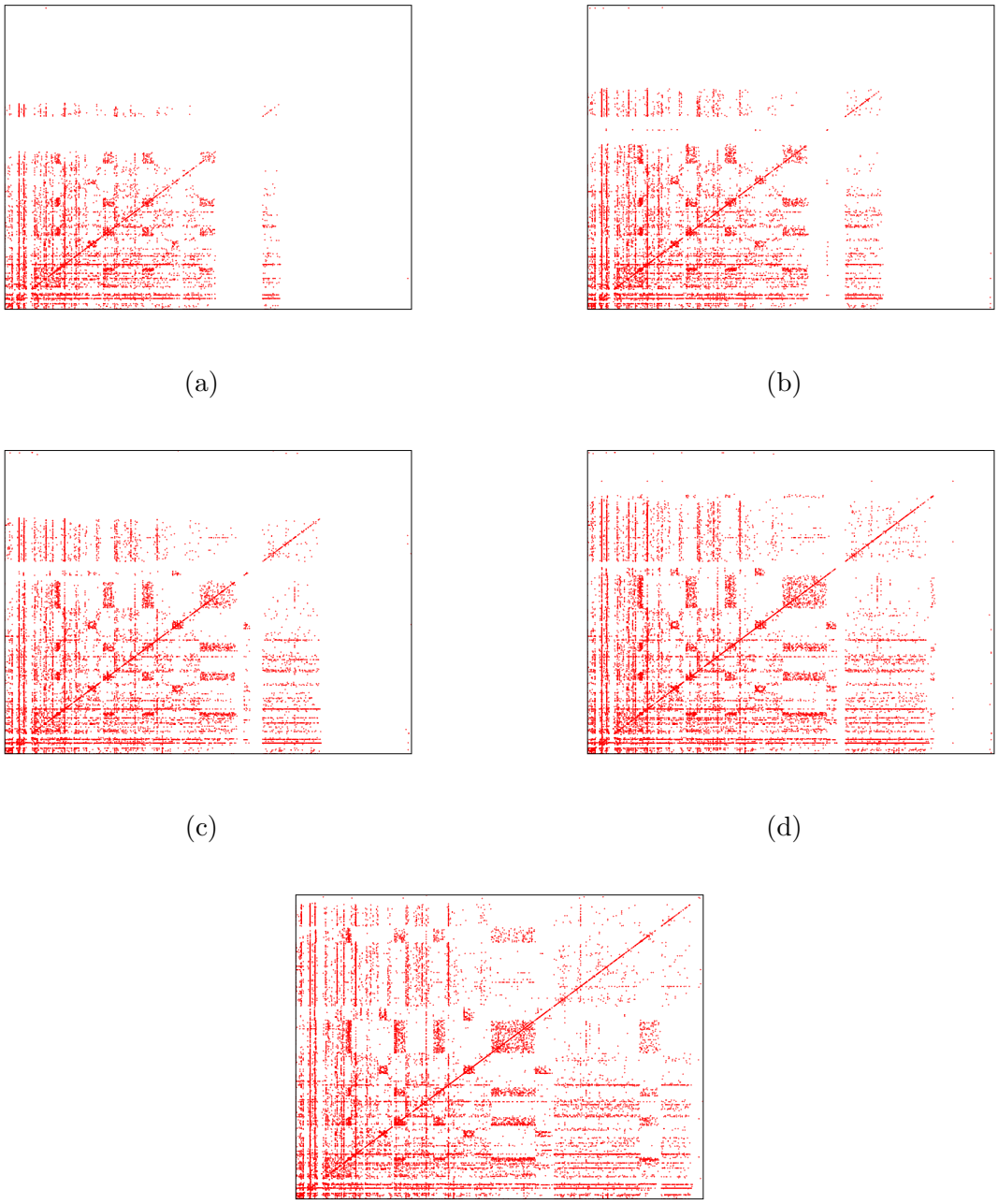
Figure 5.3 shows the NCD of the network adjacency graph. In the context of BGP, this shows the difference in the direct connections that can be inferred from the BGP RIB dumps of a given state relative to all previous states. As can be seen, there is a fair amount of churn in these values on the sampling granularity of 1 snapshot per 24 hours. Lots of peering relationships change. However, if the sampling is made more frequent and the number of comparisons or the *window size* is changed accordingly, one can tweak the granularity and the sensitivity at which one desires to see changes. The image also shows a few spikes where significant changes in the network were observed. For example, on 29th December 1998 there is a sharp decline in the number of nodes and edges that were observed, thus a large value of NCD can be seen. Previously, the nodes were roughly 4500 and edges were roughly 17000 but at this date, nodes fell down to 493 and edges fell down to 2379. It is also interesting to see the network slowly converge back to the desired state as the value of NCD goes back to the *normal* trend line. This was clearly an anomaly that this technique was able to capture.

Figure 5.4 shows the NCD of the network reachability graph. As we describe earlier in Section 4.1 that reachability is computed by performing Breadth First Search from all



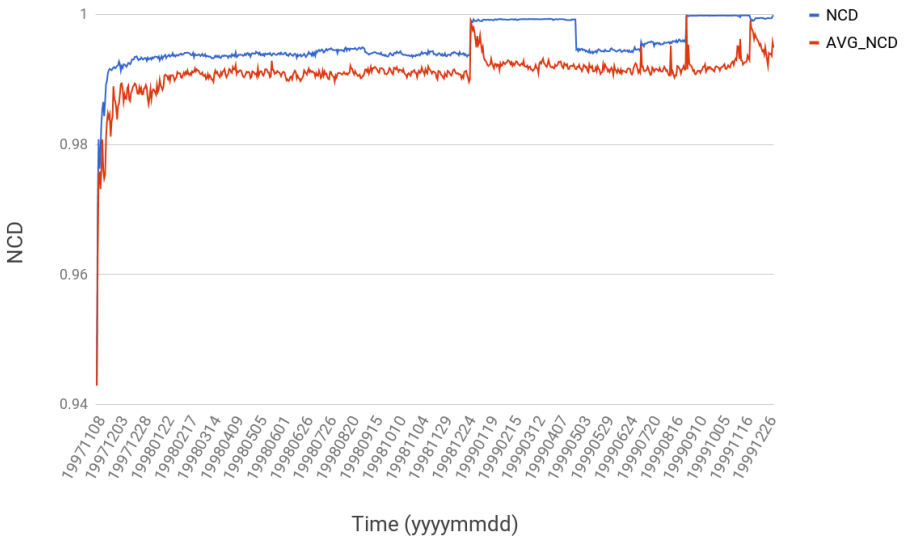
(a) (b) (c) (d) (e)

Figure 5.1: Showing the image representation of network reachability from AS-733 dataset at 6 month intervals from 8th November 1997 to 8th November 1999. Starting from top left to bottom right, each image is the reachability between ASes on the Internet. The images get more packed with pixels (i.e. more reachability relations are being added) with the passage of time.

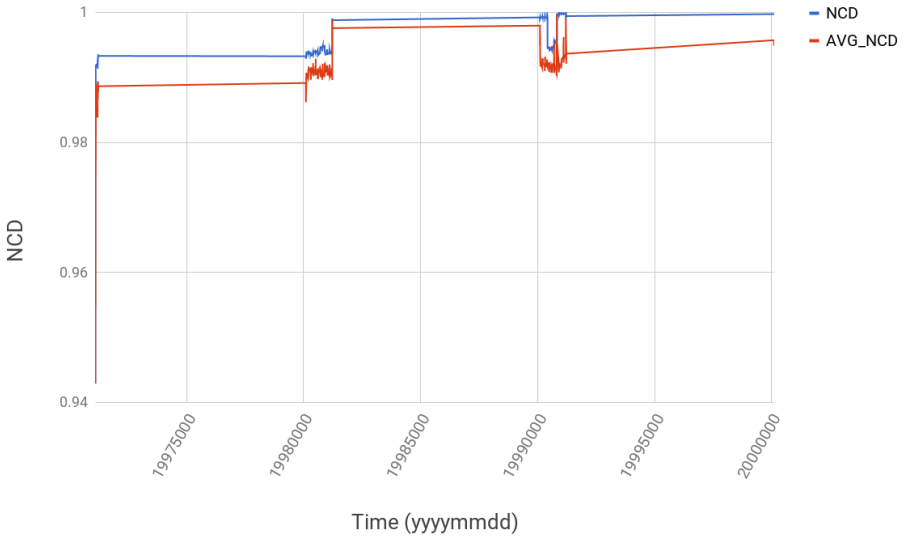


(a) (b) (c) (d) (e)

Figure 5.2: Showing the image representation of network adjacency from AS-733 dataset at 6 month intervals from 8th November 1997 to 8th November 1999. Starting from top left to bottom right, each image is the adjacency between ASes on the Internet. The images get more packed with pixels (i.e. more adjacency relations are being added) with the passage of time as more ASes join the network.



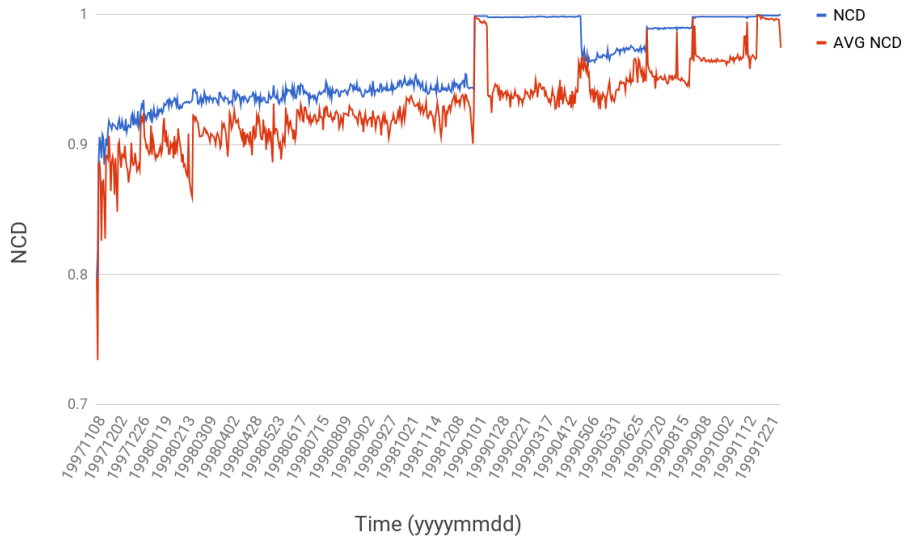
(a)



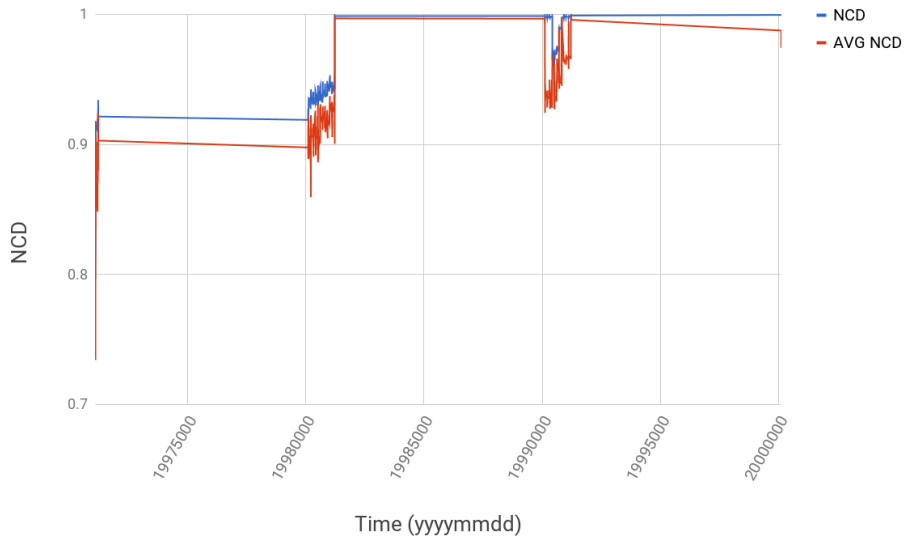
(b)

Figure 5.3: Showing the NCD computed for adjacency graphs of the AS-733 dataset where each network state is compared to 100 previous states (i.e. W = 100) (a) NCD vs Time. (b) NCD vs Time (denoised)

nodes on the graph. The NCD between reachability across snapshots also varies varies by some amount due to the constantly changing nature of the network but there are certain significant spikes that correspond to anomalous changes in reachability across ASes. This is also following the same trend as can be seen in the NCD for the adjacency graph from Figure 5.3; the spikes occur in the same times showing that both adjacency graph and reachability



(a)

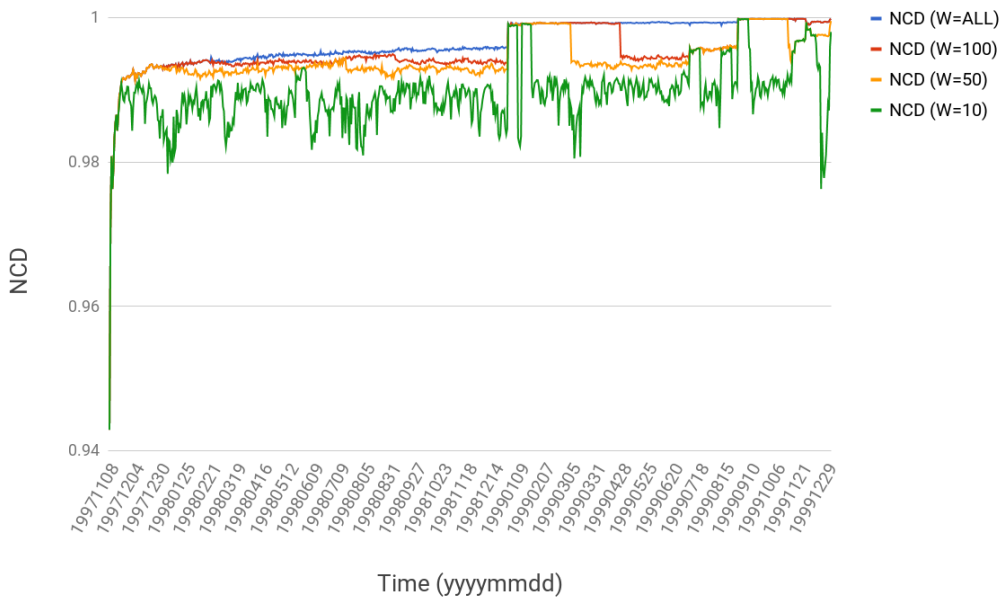


(b)

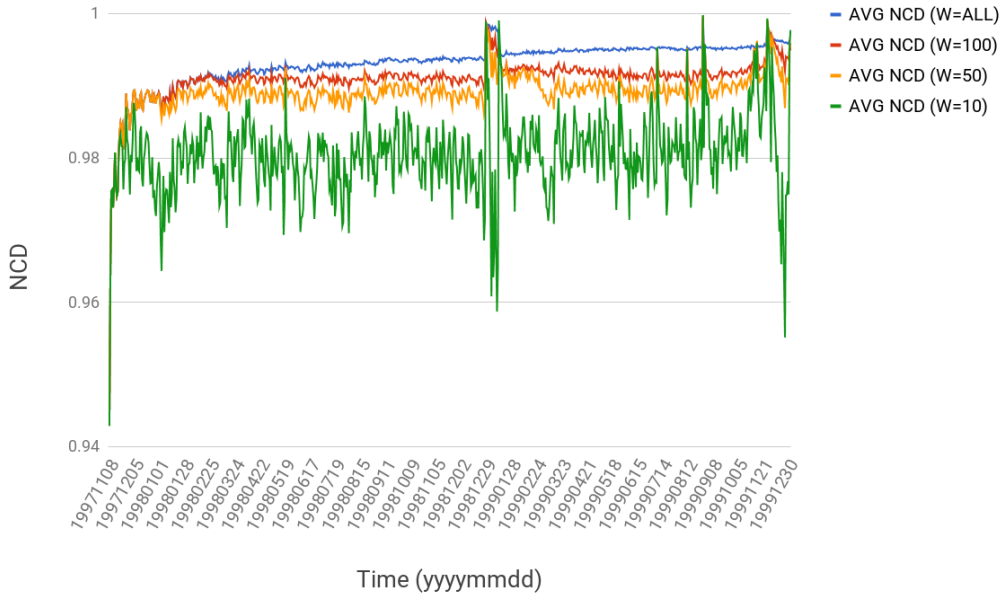
Figure 5.4: Showing the NCD computed for reachability graphs of the AS-733 dataset where each network state is compared to 100 previous states (i.e. W = 100) (a) NCD vs Time. (b) NCD vs Time (denoised)

graph can be analyzed from anomalies. However, one thing to note is the NCD values are lower for reachability graphs (shown in Figure 5.4) compared with the NCD values for adjacency graphs (shown in Figure 5.3) showing that there is less change in reachability even though adjacency changes more. This is because even if direct peering relationships change between ASes on the Internet, there can still be some other route through which an AS can

send traffic to another AS thus keeping reachability intact even if direct adjacency has now changed.



(a)



(b)

Figure 5.5: Comparing results obtained by using different values for the Window Size parameter

Window Size

Figure 5.5 shows the effect of varying the *window size* parameter which dictates how many comparisons have to be made to determine the NCD of each network snapshot. It is clear that a very small value (for example when $W = 10$) gives a lot of noise and variation in the value for NCD. This is because on the time scale that network state has been captured for this particular dataset (every 24 hours), the network state changes a lot due to a high traffic of BGP exchange messages. Thus, the value for window size is highly tied with the nature of the network as on the scale of the Internet, there would be a lot of churn expected as peering relationships change and the paths keep switching. On the scale of, say an enterprise network or a university network, the relationship between network devices and the reachability would remain fairly constant and less churn would be observed. However, as we average out the NCD with a larger previous window (for example $W = 100$ or $W = \text{ALL}$) the variation between the values is much less and an upward trend can be observed which shows how the network **evolves** over the years. The number of ASes increases, peering relationships grow thus making the graph much denser in the number of edges; this gradual growth can be visualized in the trend.

5.2.2 Graph Diff

Figure 5.6a shows the Graph Diff for two different window sizes on the same adjacency network. It can be observed that a larger window size (or even all previous network states in this particular demonstration) shows a growing trend in the network and hides the variations on a smaller time scale. A smaller window size (of 10 previous comparisons in this particular demonstrations) is able to capture much more accurately the difference of a state with some previous states and is more sensitive to changes. Again, it depends on the requirement what window size the user chooses to set as each poses its own nature of comparison and trade offs. We also compare the Graph Diff values for adjacency and reachability graphs of the same network (in Figure 5.6b) with the same window size (i.e. $W = 10$ in this example) and notice that reachability changes are higher in value by orders of magnitude. This is because the graph has more entities for reachability (more edges in particular) and the changes in those are higher in number as a consequence. The trend of both these metrics, however, is largely the same.

5.2.3 Comparing NCD with Graph Diff

It can be seen in Figure 5.7 that both NCD and Graph Diff techniques show the same general trend across time. Whenever there is a drastic spike, indicating anomalous snapshots, both of the techniques are able to convey this information. However, as can be noted that the sensitivity to changes in the networks are different for both techniques; graph diff is not able to capture the smaller changes in the networks because it only considers a change in the number of entities in the network (i.e. the number of unique and common devices and edges) whereas NCD is able to capture the difference in relation more closely by computing how much loss of information occurs in the network as a whole across snapshots. Therefore, depending on the sensitivity demands of the user, both techniques present valid use-cases.

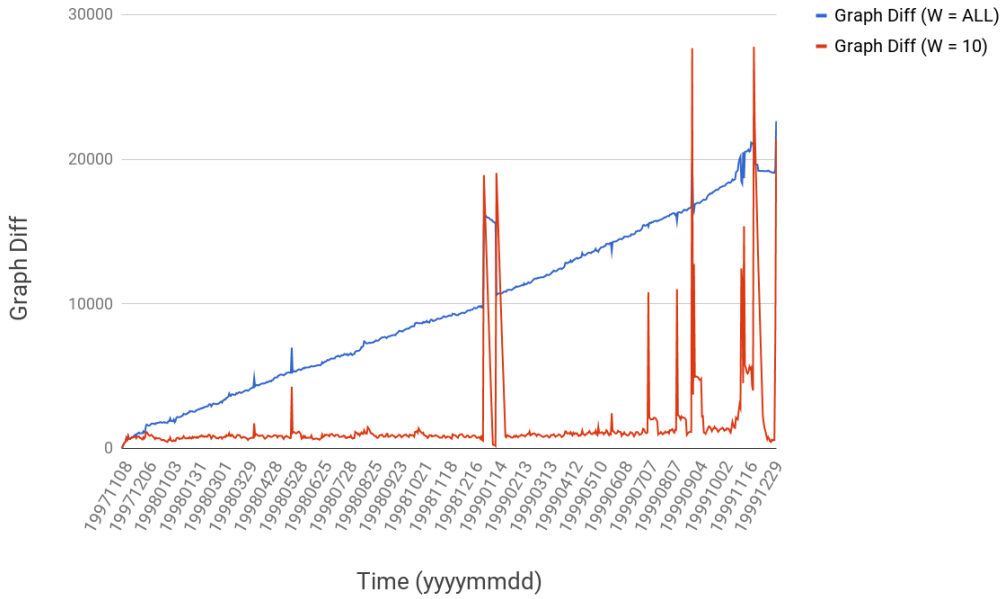
5.3 INFERENCE

We test our inference engine on private data collected from a major enterprise network. The dataset contained three years (2014-2016) worth of network snapshots with each snapshot consisting of approximately 200 - 400 routing devices belonging to a number of different vendors. Using some techniques by Veriflow, we were able to determine the packet sets exchanged on each snapshot. We use those packet sets to isnert into our R-Tree and then perform the techniques discussed in Section 4.2. The results are shown in Figure 5.8 which demonstrate the reduction in information size as a result of our technique showing the inferred low-level policies and high-level policies are much more easier to comprehend and feed to other tools.

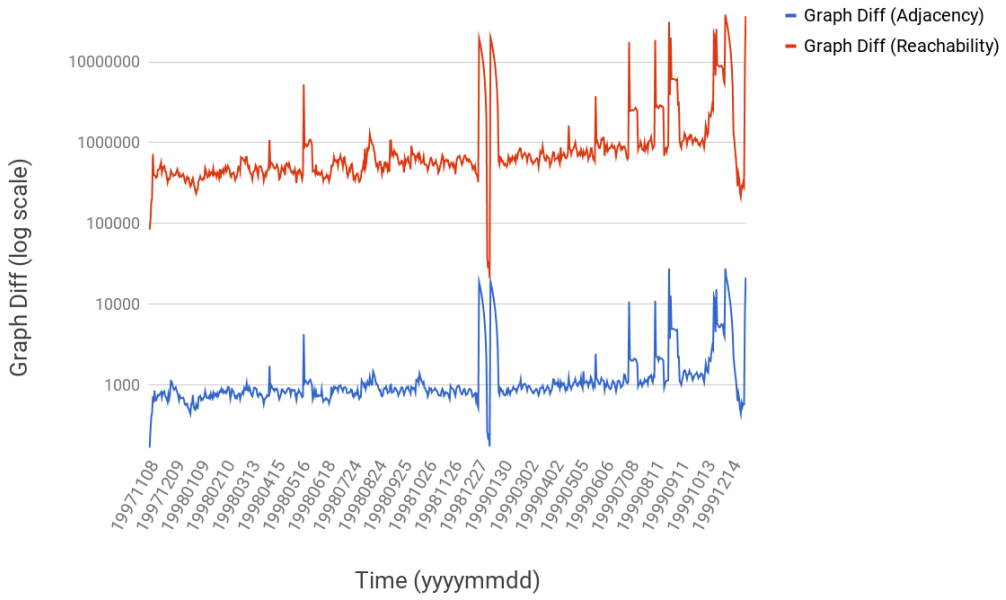
5.4 VERIFICATION

We implemented a prototype of Plankton including the equivalence class computation, agents, and policies in 495 lines of Promela and 2,854 lines of C++ code, excluding the SPIN model checker.

Using Plankton, we evaluated BGP in a data center setting, which is often employed to provide layer 3 routing down to the rack level in modern data centers [52]. We configure BGP as described in RFC 7938 [52] on fat trees of various sizes. Furthermore, we suppose that the network operator intends that traffic should pass through any of a set of acceptable *waypoints* on the aggregation layer switches. (This may be because the switches implement certain middlebox functionality; we pick a random subset of aggregation switches as the waypoints in each experiment.) However, we create a "misconfiguration" that prevents



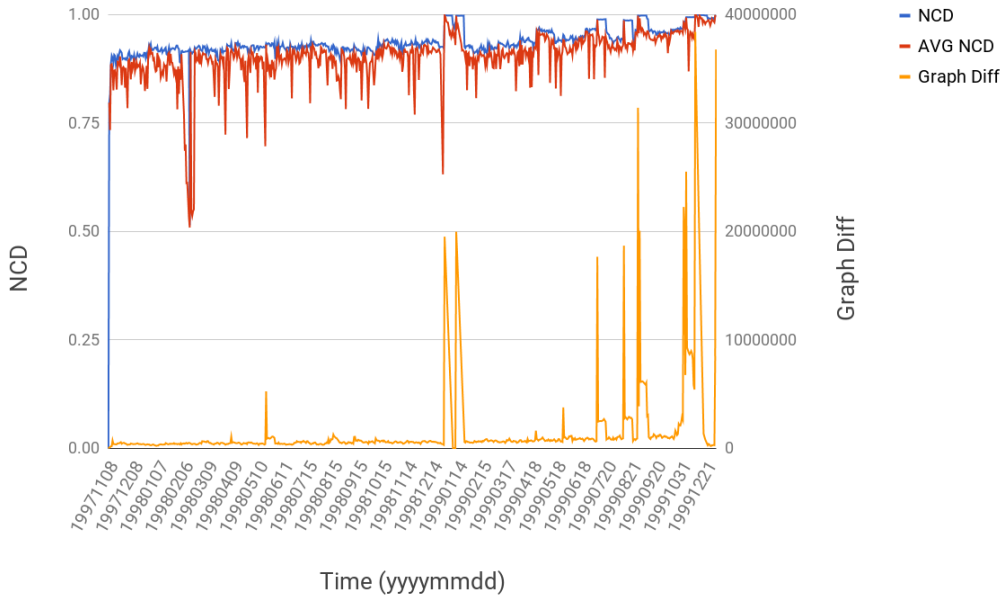
(a)



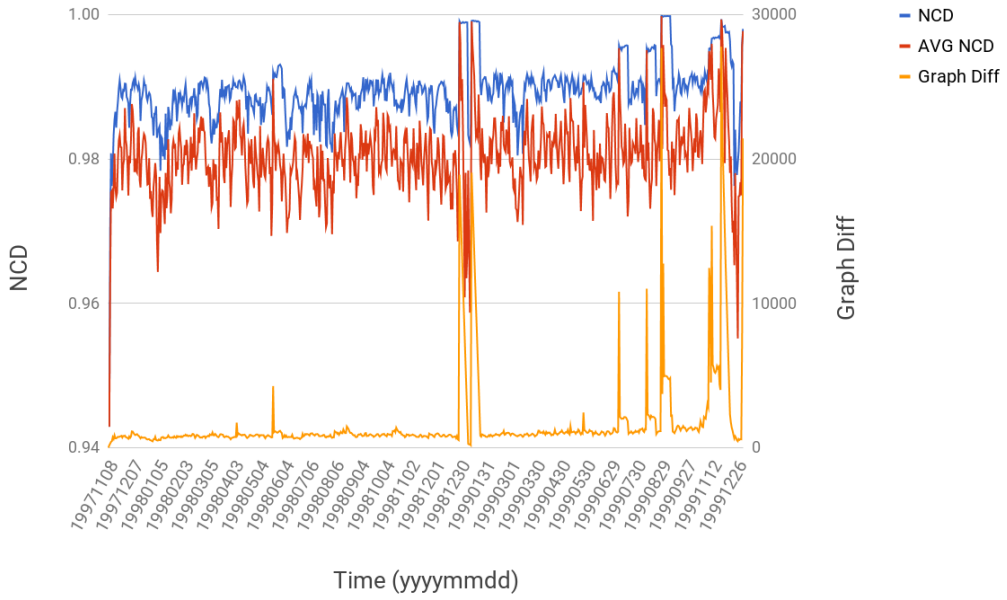
(b)

Figure 5.6: Showing Graph Diff computed for AS-733 dataset. (a) shows the effect of varying the *window size* and (b) shows the trend for Graph Diff on adjacency graphs and reachability graphs of the same networks.

multipath from being used and fails to steer routes through the waypoints. Thus, in this scenario, whether the selected path passes through a waypoint depends on the order in which



(a)



(b)

Figure 5.7: Showing NCD and Graph Diff computed for AS-733 dataset where each network state is compared to 10 previous states (i.e. $W = 10$) . (a) shows the metrics on the reachability graphs of the networks and (b) shows the same on the adjacency graphs.

updates are received at various nodes, due to age-based tiebreaking [53]. We check waypoint policies which state that the path between two edge switches should pass through one of the

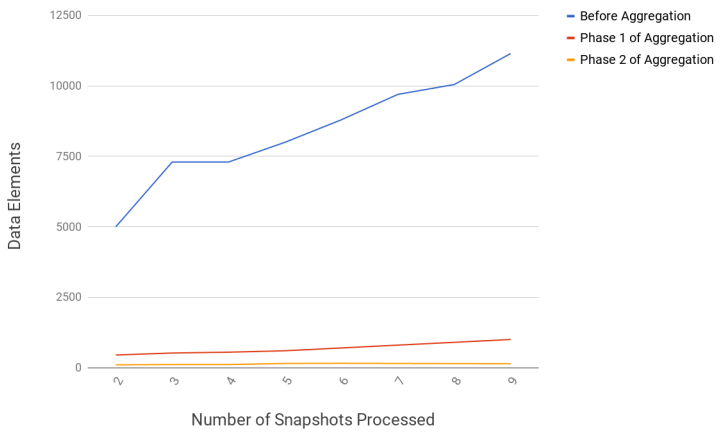


Figure 5.8: Information reduction using our inference engine

waypoints. Plankton evaluates various nondeterministic convergence paths in the network, and determines a violating sequence of events.

To further exercise Plankton's diverse policy capabilities in-real world settings, we run Plankton on fat trees and AS topologies, running OSPF. For fat trees, the link weights are assigned uniformly by us. Link weights for AS topologies were obtained from RocketFuel [54]. We assume that each device can advertise a set of prefixes. We check two types of policies - loop freedom for individual packets, and reachability in converged states. Both of these policies are checked under single link failures. While violations are found for loop freedom, in the case of reachability, due to the nature of the networks we chose and the fact that this policy queries only the converged states, no violation was reported in any of the runs. We perform each experiment both single threaded, as well as with 16 threads.

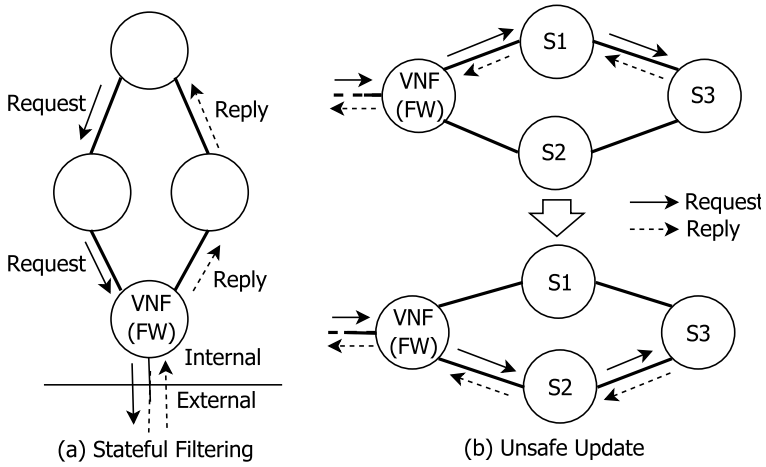


Figure 5.9: Policy violations in VNFs

As a test setup for Plankton-neo, we verify a network with 4 switches and a virtualized `iptables` firewall, as illustrated in Figure 5.9 a. It shows a virtualized firewall running on a server, configured to perform standard stateful filtering — block connection attempts from outside the organization, but allow replies to past requests. Additionally, the dataplane rules are installed such that requests and replies follow different paths, as shown. In this setup, one may expect that requests originating from inside, and their replies should not be blocked. An intuitive model-based verification platform may even declare that these conditions hold. However, in the real world, when running a virtualized firewall using `iptables` on Linux, the behavior of this setup depends on specific configuration variables in the kernel that runs the firewall. For example, when the `rp_filter` variable is enabled, the kernel performs reverse-path filtering — for each packet it forwards, it constructs a hypothetical reply and tries injecting it into the outgoing port for the current packet. If this hypothetical reply does not go out the incoming interface of the current packet, the current packet is dropped. This would cause the reachability requirements in our example to be violated. The policy we verify is that the request and reply traffic reach their respective destinations. When reverse path filtering is enabled, the firewall blocks the request traffic, causing the policy to fail. However, when we disable reverse path filtering in the kernel, the policy passed.

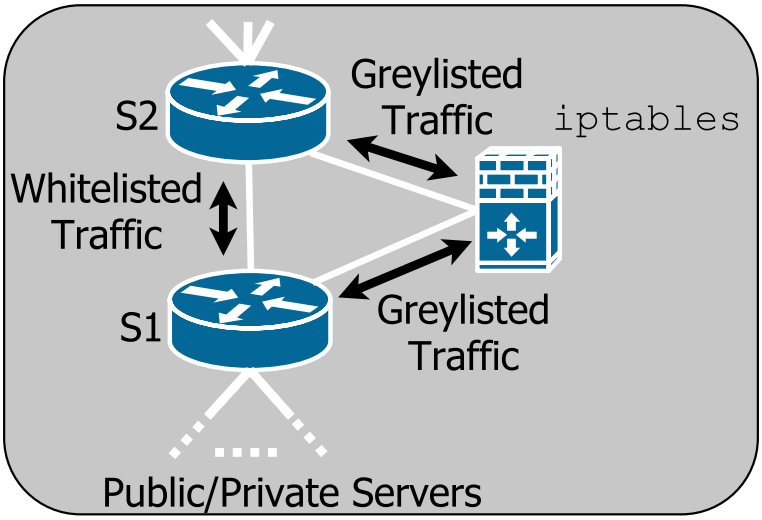


Figure 5.10: Policy enforcement in a multi-tenant data center

To test our approach at scale, we perform an experiment inspired by [55]. We consider a multi-tenant datacenter with varying number of tenants, with two types of traffic — whitelisted or greylisted. Each tenant also has two types of servers — public or private. Initially, each tenant allows all traffic to public servers, but for private servers, whitelisted traffic is always allowed, and for greylisted traffic, only replies to past requests are allowed.

We use `iptables` running on Ubuntu 14.04 to enforce the policies, as shown in Figure 5.10. This use of a real VNF implementation is what distinguishes our experiment from similar experiments in past work [55]. In our test, we assume that some of the tenants now wish to reclassify HTTP from greylisted to whitelisted . This change is implemented by updating two routers. Naturally, at any point during the transition, the correctness spec for the network states that no legitimate reply to a past request be dropped. In our experiments, Plankton-neo finds the following violation. Access switch $S1$ gets updated before the request packet reaches it. So, the request is forwarded directly to $S2$, without passing through the firewall. The reply packet reaches $S2$, before $S2$ is updated with the new policy. Abiding by the old policy, $S2$ forwards the reply to the `iptables` firewall. The firewall drops the reply, since it has not observed the request.

CHAPTER 6: PERFORMANCE

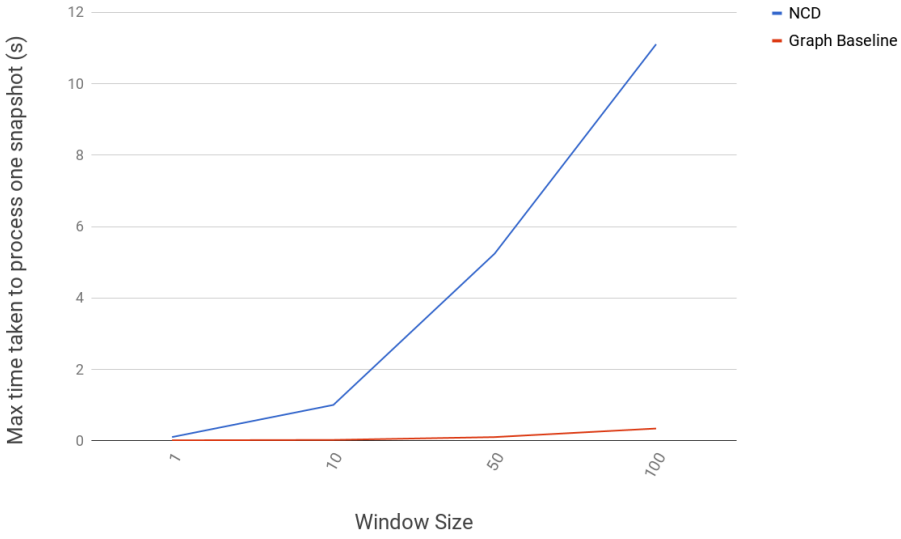
6.1 SETUP

We used a Dell Precision Tower 5810. The machine was equipped with an Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz CPU with 14 cores and Intel(R) Hyper-Threading Technology enabled to run up to 28 threads, a Seagate HDD (SATA 6.0Gb/s) and a Samsung 4 x 32GB DDR4 RAM. This machine was used to run all the experiments for the analysis and the verification modules. However, a different machine was used for running the inference module experiments with a 4 core Intel(R) i7 @ 3.40GHz and 32GB DDR3 RAM.

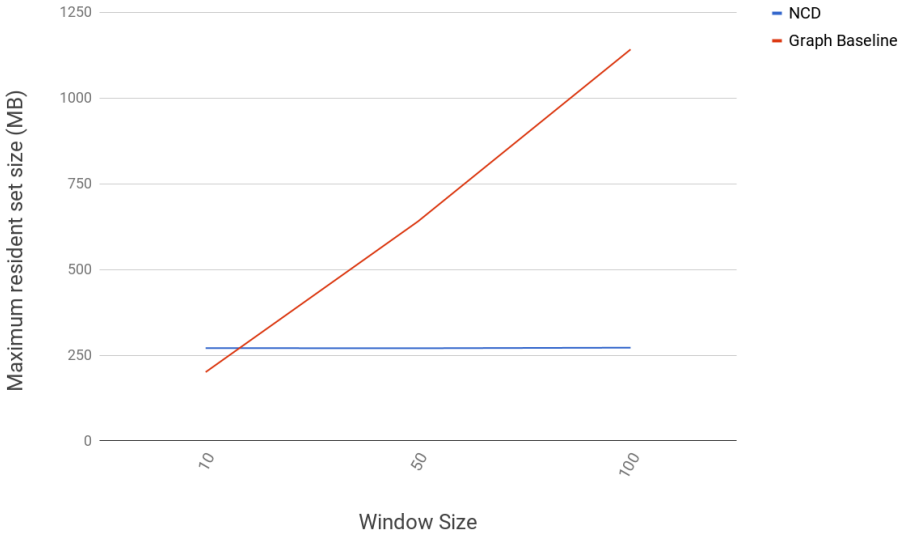
6.2 PERFORMANCE BENCHMARKS

6.2.1 Analysis

Figure 6.1 shows the time taken and memory usage to compute NCD and Graph Diff (for adjacency graphs) in our experiments run on the AS-733 dataset. It can be seen that the memory usage of Graph Diff approach is much higher because it keeps a window of graphs in-memory uncompressed whereas the NCD approach at the beggining of computations compresses all the image files and keeps them all in-memory. Also note that the time taken to process a snapshot using Graph Diff is much lesser compared to performing NCD operations (which involve a sufficient amount of computation when compressing the concatenation of various image pairs.) Also, the time taken to generate graphs (one graph with about 3500 vertices and 13500 edges takes approximately $0.02s$ to generate, another one with about 5500 vertices and 22000 edges takes about $0.04s$ to generate) is lesser than the time taken to generate the images (one image takes about $20s$ to generate) used by NCD. In order to generate images, first the graph is generated and then mappings are performed to output a 2D matrix with appropriate cordinates computed to represent network devices on the image. In practice, we make use of multiple cores to generate images in parallel as the nature of this computation is embarrassingly parallel with no dependency of one image generation to any other images in the dataset.



(a)



(b)

Figure 6.1: (a) Maximum time taken to process one AS-733 snapshot (b) Max resident memory usage while processing all AS-733 snapshots

6.2.2 Inference

Figure 6.2 shows the time taken to run the inference engine on the enterprise network mentioned in Section 5.3. It can be seen that using 4 cores through OpenMP gives a significant boost to performance. We develop our framework such that adding packet sets from multiple snapshots to the R-Tree can be done in parallel (with a couple of atomic

Figure 6.2: Time taken by the inference engine

operations) and the querying for intersections can be done in parallel as well. Since the process of inference is not something that a network operator would want to do in real-time, nor is it something that needs to be done frequently, the time taken for the process to finish given enough compute resources is acceptable.

6.2.3 Plankton

Figure 6.3 illustrates the average and worst-case time taken by Plankton to finish the BGP (in data center) check described in Section 5.4. Time and memory both depend somewhat on the chosen set of aggregation switches.

The time taken and memory consumed to carry out the RocketFuel experiments described in Section 5.4 are illustrated in Figures 6.4 and 6.5. For fat-trees of 1280 nodes, the reachability check consumes 28.5GB of RAM, running single threaded. Given the symmetry of the network, we expect a run with 16 threads to use close to 500GB of RAM. While this is not unreasonable for a network this large, we avoid running the experiment ourselves.

We compare Plankton with Minesweeper; the former has an expanded scope of functionality, but on scenarios both are able to verify, we show 143× single-threaded speed improvements, with both the basic architecture and the optimizations playing a significant role in the speedup. Our experiments so far have focused on the correctness and effectiveness of Plankton as a platform that enables new verification functionality (where, therefore, a direct comparison with past work was not possible). We also wish to evaluate how Plankton com-

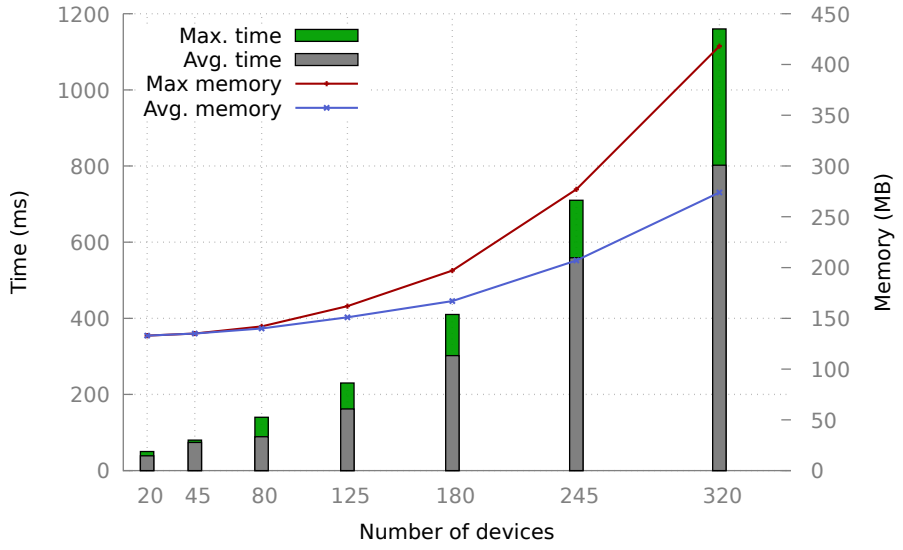


Figure 6.3: Time and memory overhead for waypoint query on BGP DCs

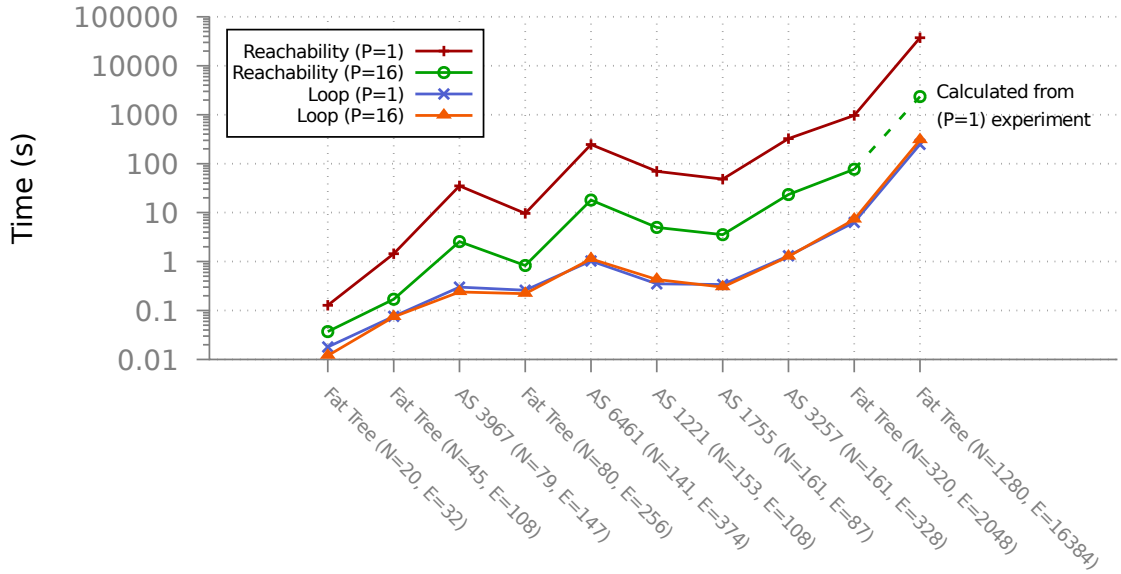


Figure 6.4: Time taken for various policies under single link failure, with $P = 1$ and $P = 16$ parallel threads.

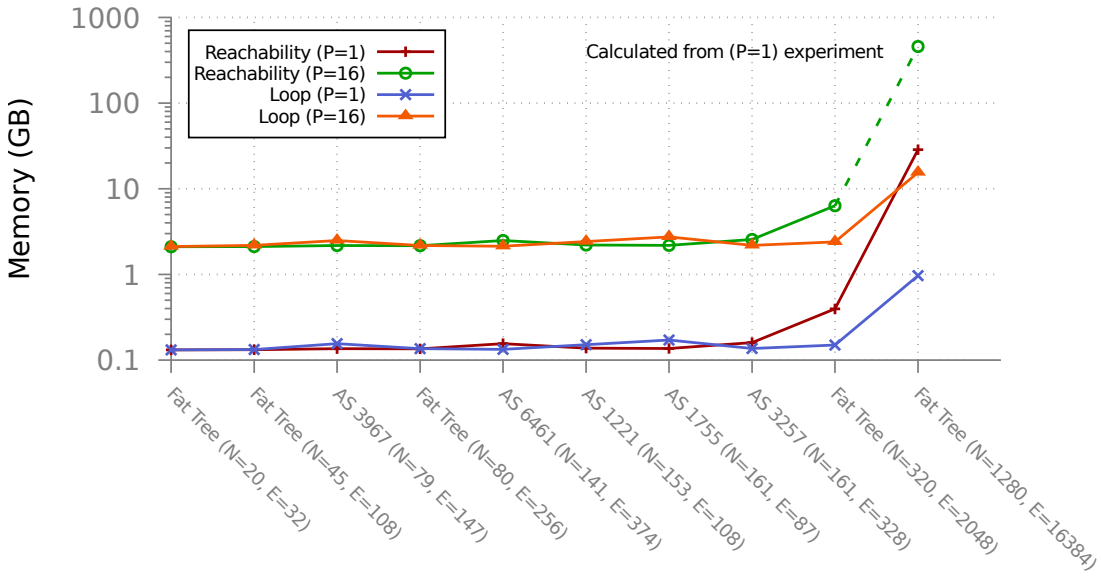


Figure 6.5: Memory consumed for various policies under single link failure, with $P = 1$ and $P = 16$ parallel threads.

2* Experiment	Plankton		2* Minesweeper
	16 threads	1 thread	
20-node Fat Tree	0.04 s	0.13 s	1.44 s
45-node Fat Tree	0.17 s	1.44 s	206.57 s
320-node Fat Tree	77.06 s	966.39 s	¿ 20 hours (stopped)
AS 1755	3.55 s	48.31 s	4046.74 s
AS 3967	2.56 s	35.08 s	6894.28 s

Figure 6.6: Comparison of Plankton with Minesweeper.

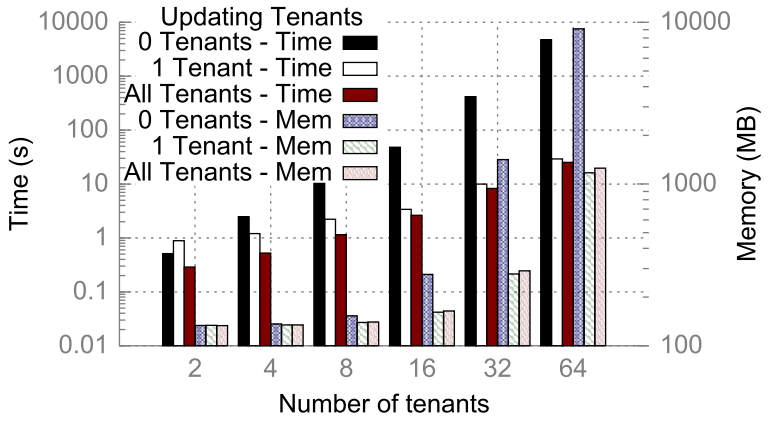
Experiment	Single emulation	Multiple Emulation
64 Tenants, no update	5835.52 s	4732.13 s
32 Tenants, no update	680.28 s	413.84 s
64 Tenants, all tenants update	29.22 s	27.73 s

Figure 6.7: Single emulated device vs. one per middlebox

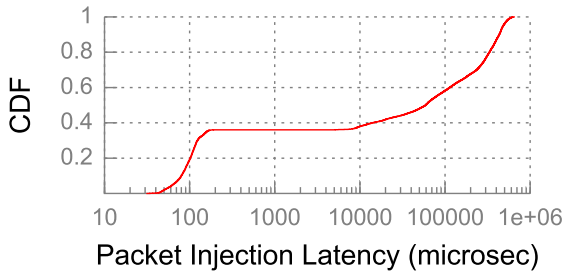
pares to prior approaches, for use cases that do not require the full capabilities of Plankton. To evaluate this, we solve the same verification problems using Plankton, and the current state-of-the-art, Minesweeper. We verify the reachability policy in the converged states of various networks under single-link failures. We choose the reachability policy because it is usually not violated due to single link failures, meaning that the verifier needs to explore all possible converged states of the system, and then conclude that the policy passes. Figure 6.6 illustrates the measurements we obtain from the various runs. In our experiments, we observed that Minesweeper ran single-threaded except for parsing the input, possibly due to a unified model for the entire network, unlike the partitioned model of Plankton. For a direct comparison, we perform Plankton's checks in single threaded mode also. Plankton performs around two orders of magnitude better in all cases. For networks with high symmetry, the larger the network, the higher the speedup, thanks to Plankton's device-equivalence optimization. However, the speedup in Plankton is not just due to symmetry. This is illustrated by the higher performance observed for the RocketFuel AS topologies, which do not benefit from the symmetry-based optimizations in Plankton, and indicating that our architectural approach has benefits.

6.2.4 Plankton-neo

As described in Section 4.3.3 we restore the state of the middleboxes by replaying the history of prior packets before injecting new packets. But doing so has an impact on performance, due to the inherently high latency incurred in update replay (See Figure 6.8 b). When using a separate emulation for each middlebox, there is a higher likelyihood that the emulated middlebox is in the same state as expected by the verifier, and hence, does not require additional state restoration. We empirically evaluate this design choice by repeating the multi-tenant data center experiment using a single emulation setup for all middleboxes. Figure 6.7 compares observations. For our test case, having separate emulation for each middlebox produces as much as 39% improvement in the time spent. This can be further improved by having *multiple* emulated devices to run the same middlebox. In general, the relationship between the middleboxes in the network and the emulation setups in Plankton-neo



(a) Time and Memory Overhead



(b) Latency CDF

Figure 6.8: Measurements from data center experiment

can be many-to-many. A full exploration remains to be done.

We perform the experiment mentioned in Section 5.4 with different number of tenants, with varying number of tenants performing the whitelisting change. Figure 6.8 a illustrates the time and memory used by a single run of the test (Time and number of tenants axes are logarithmic). When the fraction is 0, there are no changes being made, and hence, the policy passes. When at least one tenant updates its policy, a violation of the policy exists, and we correctly find it.

Figure 6.8 a provides some interesting insights into the nature of the problem. Intuitively, the verification problem is hardest when there is no update happening in the network. This is because the testing procedure has to exhaust every possible execution and finally declare that the policy holds. This is reflected in the time and memory consumption.

Figure 6.8 b illustrates the CDF of the latency incurred in performing packet injection and observing the outcome. It is clear that there exist two different groups of latency measurements. The faster one indicates packet processing at line rate, without having to restore middlebox state or wait for timeouts. This is the case where the emulated middlebox is in the same state as the one the verifier requires it to be, and the packet that is newly

injected does not get dropped. The slower set of measurements may be caused due to timeout or state restoration.

CHAPTER 7: CONCLUSION & FUTURE WORK

7.1 CONCLUSION

Modern day networks are evolving at a fast pace in terms of the number of devices, complexity of interaction and the sophistication of policies enforced. This, though making the networks extremely efficient and highly performant, poses high risks in terms of security of the network infrastructure and the applications using it. As the complexity grows, the attack surface along with the likelihood of unintentional misconfigurations of the network also grows dramatically as can be seen in a lot of production networks in the recent past. This opens up room for working towards efficient mechanisms of analysing the networks states over time, making sure the network is obeying the specified invariants and no anomalous events are occuring in the network. To achieve this end, we propose an end-to-end framework for analysing existing networks, inferring from them the policies that have been enforced without explicitly having to enumerate them all. Finally, we present a formal verification tool that performs explicit-state verification of a network model defined over equivalence classes that is capable of verifying control plane and cross-layer network dynamics, including properties that are themselves temporal in nature.

7.2 FUTURE WORK

Most of the implementation and experimentation performed for the proposed techniques has a lot of room for improvement before they can be used in production settings. This involves making all the modules interact with each other efficiently, distribution of non-dependent tasks to execute simultaneously and reducing the memory footprint to be able to scale the system to larger networks. We also intend to evaluate the proposed framework on more real-world datasets or production networks to better understand its effectiveness and usefulness.

REFERENCES

[1] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11, New York, NY, USA, 2011. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018470 pp. 290–301.

[2] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian pp. 113–126.

[3] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 15–27.

[4] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, IL, 2013. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/kazemian

[5] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789770.2789803 pp. 469–483.

[6] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese, "Efficient Network Reachability Analysis Using a Succinct Control Plane Representation," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, GA, 2016. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/fayaz pp. 217–232.

[7] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, 2016. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934876

[8] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098834 pp. 155–168.

[9] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/canini pp. 127–140.

[10] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock, "Scalable verification of border gateway protocol configurations with an smt solver," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984012

[11] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-mutable-datapaths pp. 699–718.

[12] P.-P. Vázquez and J. Marco, "Using normalized compression distance for image similarity measurement: an experimental study," *The Visual Computer*, vol. 28, no. 11, pp. 1063–1084, Nov 2012. [Online]. Available: https://doi.org/10.1007/s00371-011-0651-2

[13] C. E. Au, S. Skaff, and J. J. Clark, "Anomaly detection for video surveillance applications," in *18th International Conference on Pattern Recognition (ICPR'06)*, vol. 4, 2006, pp. 888–891.

[14] "On measuring the complexity of networks: Kolmogorov complexity versus entropy," https://www.hindawi.com/journals/complexity/2017/3250301/, (Accessed on 04/03/2018).

[15] S. S. Kim and A. L. N. Reddy, "Statistical techniques for detecting traffic anomalies through packet header data," *IEEE/ACM Transactions on Networking*, vol. 16, no. 3, pp. 562–575, June 2008.

[16] S. S. Kim and A. L. N. Reddy, "Modeling network traffic as images," in *IEEE International Conference on Communications, 2005. ICC 2005. 2005*, vol. 1, May 2005, pp. 168–172 Vol. 1.

[17] S. S. Kim and A. L. N. Reddy, "Image-based anomaly detection technique: Algorithm, implementation and effectiveness," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 10, pp. 1942–1954, Oct 2006.

[18] S. S. Kim and A. L. N. Reddy, "A study of analyzing network traffic as images in real-time," in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 3, March 2005, pp. 2056–2067 vol. 3.

[19] Z. Tan, A. Jamdagni, X. He, P. Nanda, R. P. Liu, and J. Hu, "Detection of denial-of-service attacks based on computer vision techniques," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2519–2533, Sept 2015.

[20] A. Tongaonkar, N. Inamdar, and R. Sekar, "Inferring higher level policies from firewall rules," in *Proceedings of the 21st Conference on Large Installation System Administration Conference*, ser. LISA'07. Berkeley, CA, USA: USENIX Association, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1349426.1349428 pp. 2:1–2:10.

[21] T. Benson, A. Akella, and D. Maltz, "Mining policies from enterprise network configuration," in *Internet Measurement Conference*. Association for Computing Machinery, Inc., November 2009. [Online]. Available: https://www.microsoft.com/en-us/research/publication/mining-policies-from-enterprise-network-configuration/

[22] S. Kandula, R. Chandra, and D. Katabi, "Whats going on? learning communication rules in edge networks," in *ACM SIGCOMM*. Association for Computing Machinery, Inc., August 2008. [Online]. Available: https://www.microsoft.com/en-us/research/publication/whats-going-on-learning-communication-rules-in-edge-networks/

[23] A. Horn, A. Kheradmand, and M. Prasad, "Delta-net: Real-time network verification using atoms," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn-alex pp. 735–749.

[24] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "Crystalnet: Faithfully emulating large production networks," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132759 pp. 599–613.

[25] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/zeng pp. 87–99.

[26] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese, "Scaling network verification using symmetry and surgery," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837657 pp. 69–83.

[27] D. Sethi, S. Narayana, and S. Malik, "Abstractions for model checking SDN controllers," in *2013 Formal Methods in Computer-Aided Design*. IEEE, oct 2013. [Online]. Available: https://doi.org/10.1109/fmcad.2013.6679403

[28] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, ser. NSDI'04. Berkeley, CA, USA: USENIX Association, 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251175.1251187 pp. 12–12.

[29] M. Dobrescu and K. Argyraki, "Software dataplane verification," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014, pp. 101–114.

[30] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, "A formally verified nat," in *Proceedings of the SIGCOMM '17*. New York, NY, USA: ACM, 2017, pp. 141–154.

[31] M. Hutter, "Algorithmic information theory," *Scholarpedia*, vol. 2, no. 3, p. 2519, 2007, revision #90953.

[32] "University of oregon route views project," http://www.routeviews.org/, (Accessed on 03/22/2018).

[33] "Ripe ncc libbgpdump," https://bitbucket.org/ripencc/bgpdump, (Accessed on 03/22/2018).

[34] T. McGregor, S. Alcock, and D. Karrenberg, "The ripe ncc internet measurement data repository," in *Proceedings of the 11th International Conference on Passive and Active Measurement*, ser. PAM'10. Berlin, Heidelberg: Springer-Verlag, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1889324.1889336 pp. 111–120.

[35] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[36] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, pp. 345–, June 1962. [Online]. Available: http://doi.acm.org/10.1145/367766.368168

[37] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959. [Online]. Available: http://dx.doi.org/10.1007/BF01386390

[38] E. Moore, *The Shortest Path Through a Maze*, ser. Bell Telephone System. Technical publications. monograph. Bell Telephone System., 1959. [Online]. Available: https://books.google.com/books?id=IVZBHAAACAAJ

[39] H. Bunke, P. Dickinson, A. Humm, C. Irniger, and M. Kraetzl, "Computer network monitoring and abnormal event detection using graph matching and multidimensional scaling," in *Proceedings of the 6th Industrial Conference on Data Mining Conference on Advances in Data Mining: Applications in Medicine, Web Mining, Marketing, Image and Signal Mining*, ser. ICDM'06. Berlin, Heidelberg: Springer-Verlag, 2006. [Online]. Available: http://dx.doi.org/10.1007/11790853_45 pp. 576–590.

[40] M. Li, X. Chen, X. Li, B. Ma, and P. M. Vitanyi, "The similarity metric," *IEEE Trans. Inf. Theor.*, vol. 50, no. 12, pp. 3250–3264, Dec. 2004. [Online]. Available: http://dx.doi.org/10.1109/TIT.2004.838101

[41] A. Kolmogorov, "On tables of random numbers," *Theoretical Computer Science*, vol. 207, no. 2, pp. 387 – 395, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397598000759

[42] "Kolmogorov complexity a primer math programming," https://jeremykun.com/2012/04/21/kolmogorov-complexity-a-primer/, (Accessed on 03/22/2018).

[43] C. H. Bennett, P. Gács, M. Li, P. M. B. Vitányi, and W. H. Zurek, "Information distance," *CoRR*, vol. abs/1006.3520, 2010. [Online]. Available: http://arxiv.org/abs/1006.3520

[44] R. Cilibrasi and P. M. B. Vitányi, "Clustering by compression," *CoRR*, vol. cs.CV/0312044, 2003. [Online]. Available: http://arxiv.org/abs/cs.CV/0312044

[45] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Tech. Rep., 1994.

[46] "Lzma sdk (software development kit)," https://www.7-zip.org/sdk.html, (Accessed on 03/22/2018).

[47] H. Zimmermann, "Innovations in internetworking," C. Partridge, Ed. Norwood, MA, USA: Artech House, Inc., 1988, ch. OSI Reference Model&Mdash;The ISO Model of Architecture for Open Systems Interconnection, pp. 2–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=59309.59310

[48] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, June 1984. [Online]. Available: http://doi.acm.org/10.1145/971697.602266

[49] G. J. Holzmann, "The model checker spin," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997. [Online]. Available: http://dx.doi.org/10.1109/32.588521

[50] G. Holzmann, "On-the-fly model checking," *ACM Comput. Surv.*, vol. 28, no. 4es, Dec. 1996. [Online]. Available: http://doi.acm.org/10.1145/242224.242379

[51] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, June 2014.

[52] P. Lapukhov, A. Premji, and J. Mitchell, "Use of BGP for Routing in Large-Scale Data Centers," RFC 7938 (Informational), RFC Editor, Fremont, CA, USA, pp. 1–35, Aug. 2016. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7938.txt

[53] P. Lapukhov, "Equal-Cost Multipath Considerations for BGP," Internet Engineering Task Force, Internet-Draft draft-lapukhov-bgp-ecmp-considerations-00, Oct. 2016, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-lapukhov-bgp-ecmp-considerations-00

[54] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson, "Measuring isp topologies with rocketfuel," *IEEE/ACM Trans. Netw.*, vol. 12, no. 1, pp. 2–16, Feb. 2004. [Online]. Available: http://dx.doi.org/10.1109/TNET.2003.822655

[55] S. Ghorbani and P. B. Godfrey, "Coconut: Seamless scale-out of network elements." in *EuroSys*, 2017, pp. 32–47.