UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ENHANCING AUTOMATED NETWORK MANAGEMENT

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Huazhe Wang

December 2019

The Dissertation of Huazhe Wang
is approved:

Chen Qian , Chair

J.J. Garcia-Luna-Aceves

Ying Zhang

Quentin Williams
Acting Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Enhancing Automated Network Management

by

Huazhe Wang

Network management benefits from automated tools. With the recent advent of software-defined principles, automated tools have been proposed from both industry and academia to fulfill function components in the network management control loop. While automation aims to accommodate the ever increasing network diversity and dynamics with improved reliability and management efficiency, it also brings new concerns as it's becoming more difficult to understand the control of the network and operators cannot rely on traditional troubleshooting tools. Meanwhile, how to effectively integrate new automation tools with existing legacy networks remains a question. This dissertation presents efficient methods to address key functionalities within the control loop in the adaption of automated network management.

Identifying the network-wide forwarding behaviors of a packet is essential for many network management tasks, including policy enforcement, rule verification, and fault localization. We start by presenting AP Classifier. AP Classifier was developed based on the concept of atomic predicates which can be used to characterize the forwarding behaviors of packets. There is an increasing trend that enterprises outsource their Network Function (NF) processing to a cloud to lower cost and ease management. To avoid threats to the enterprise's private information, we propose SICS based on AP Classifier, a secure and dynamic NF outsourcing framework. Stateful NFs have become essential parts of modern networks, increasing the complexity in network management. A major step in network automation is to automatically translate high level network intents into low level configurations. To ensure those configurations and the states generated

by automation match intents, we present Epinoia, a network intent checker for stateful networks. While the concept of auto-translation sounds promising, operators may not know what intents should be. To close the control loop, we present AutoInfer to automatically infer intents of running networks, which helps operators understand the network runtime states.

To my parents and friends.

Acknowledgments

This dissertation was a wonderful journey of about five years, with many ups and downs. It would not have existed without all the amazing people who helped, supported and trusted in me during this journey, many of whom I list below.

I would like to first thank my advisor, Chen Qian. Chen was the one who introduced me to networking. I can still remember our very first overseas phone call when he patiently described to me what is his research about and what kind of topics we can work on. Beside research, Chen has also been a source of wisdom about things that are both related to work (how to tell stories, how to write/present) and life. I am also grateful for his support of my projects outside school, such as my internships at Hewlett Packard Labs and my job searching after graduation.

I would also like to thank my dissertation reading committee members, J. J. Garcia-Luna-Aceves and Ying Zhang, for their constructive feedback and the fruitful discussions on my defense.

I was very lucky to collaborate with very bright and nice people. I would like to express my gratitude to Simon S. Lam and Hongkun Yang. Simon is the PhD advisor of Hongkun and Chen at UT Austin. Simon and Hongkun worked with me on my first project and I would like to thank them for their trust, for all the knowledge they shared with me and for their constructive feedback. My gratitude also goes to Puneet Sharma and Joon-Myung Kang for having me interning at Hewlett Packard Labs in Palo Alto for two productive summers in 2017 and 2018. They have been mentors and collaborators throughout my last few PhD years. They taught me how to combine research with reality and have tirelessly helped me improve both my delivery and the focus of my presentations.

The life in Santa Cruz would not have been the same without many friends. I would like to express my gratitude to Christina Parsa and her family. Chris has been a teacher,

a close friend, a partner in getting lunch during workdays. She always made time to help read my papers, teach me how to write correct and meaningful English without ever asking for anything in return. Since we met, Chris has invited me over to many family dinners especially on holidays, which makes me regard them as my second family in the US. I was lucky to share most of my PhD journey with Ye Yu and Xin Li. We have become close friends since the beginning. We shared endless discussions, spring festival dumplings, camping trips, paper rejections and acceptances, and many other good and bad moments. They both graduated one year earlier than me. I hope our paths cross often. The culture of the lab became more active when many new members joined in the last two years. I would like to thank Minmei Wang, Haofan Cai, Shouqian Shi, Junjie Xie, Ge Wang, Xiaofeng Shi and Minghao Xie for your support and invaluable discussions which helped shape this research. I will miss the friendly environment in the lab and countless good times we had at every Saturday game night.

Finally, I would like to express my love and gratitude to my parents. Without their support, I probably would not have done a PhD. Starting from an early age, my parents always valued my education. My father introduced me to science and engineering and motivated me to study computer engineering. I would like to thank my mother for standing by me, taking care of me even through our great distance. While, I often disagree with them on issues, everything I do is influenced by them. I feel very lucy to have such nice and supportive parents.

Chapter 1

Introduction

Managing a large packet network is a complex task. The procedure of processing packets is prone to faults from configuration errors and unexpected network dynamics. The complexity also comes from the rapid growth in size and diversity of modern networks, which include cloud (e.g., Amazon Web Service [3], Azure [15], Google Cloud [8]), software-defined networks [24], mobile devices, Internet of Things as well as traditional physical topologies. Network Functions (NFs) are a vital part of modern networks. Compared with switches and routers, NFs implement more diverse functions and their packet processing behavior may depend on the packet history previously encountered. Examples of such stateful packet processing include firewalls that allow inbound packets if they belong to established connections and web proxies that cache popular content etc. NFs are becoming increasingly prevalent in today's network, complicating the problems encountered in network management [102]. Furthermore, network configurations need to be continuously updated to serve the ever evolving business goals to address both security and performance requirements. According to a study, 70% of network failures occurred during changing network configurations [69].

The increasing complexity of business rules and policies coupled with the increasing size of modern networks has made the tasks of network operators extremely dif-

ficult. With the advent of software-defined principles, automated platforms and tools are proposed from both industry and academia to fullfil each function component in the network management control loop. For instance, to translate diverse policies automatically, [4,5,30,36,37,105] designed languages and specifications that can synthesis network control/data plane configurations (e.g., BGP configurations). Towards more reliable networks, [35,49,62,63,65,118,126] proposed formal analysis methods to verify essential network properties (e.g., The network has no routing loop for all packets.). To gain awareness of the state of devices network wide, [39,83,107,108,120] proposed runtime monitoring schemes to automatically collect network telemetry data. While network automation eases tasks for operators, there are still many challenges to make those automated tools both efficient and reliable. In this dissertation, we identify some of key functionalities that are missing in existing automated platforms.

Network-wide packet behavior identification. Let a flow be an equivalence class of packets defined on a subset of fields in the packet header, e.g., the 5-tuple consisting of source address, destination address, source port, destination port, and protocol type. All packets of a flow have the same forwarding behaviors in a network (also referred to as the flow's behaviors) when there is no update. Network-wide packet behavior identification is a function that discovers the actual forwarding behaviors of the packets in a flow (or a set of flows) including their forwarding paths, where they stop or are dropped, and which boxes they traverse, by analyzing network state [59]. Packet behavior identification is essential for network management tasks such policy enforcement [96,125], verification of flow properties [34, 73] and network fault localization [123, 127]. A practical packet behavior identification method must satisfy three requirements. First, it provides a high throughput in responding to packet behavior queries. According to recent measurement results [38, 61], a large data center network may see hundreds of thousands of new flows per second. SDNs should support hundreds of data plane

updates per second [60] and each update may need to query multiple flows to verify correctness. Hence a desired throughput should exceed one million packet queries per second (1 Mqps). Second, the query structure should fit into a small and fast memory such as cache. Third, the query structure can be updated in real time under data plane changes to ensure that query results reflect the current network state.

Unfortunately, none of the existing solutions can meet all of the requirements stated above. A straightforward approach is to maintain copies of the flow tables for all boxes in the controller. However even for a medium-scale network used in [63], tens of GBs are required to store all rules [59]. Due to slow search speed among flow tables and disk I/Os, the resulting query throughput is very low. Very recently, [59] propose to use a multi-valued decision diagram (MDD) to classify flows to different sets of network-wide behaviors. However, one limitation is that an MDD cannot be updated in real time.[1]

Secure and dynamic middlebox outsourcing. While traditionally, middleboxes a.k.a network functions, have been deployed as dedicated hardware devices inside an enterprise, the introduction of the Network Functions Virtualization (NFV) technology [57] and the cloud services has opened a new opportunity to outsource middleboxes to third-party clouds. An initial effort [102] indicates that middlebox outsourcing can be achieved without significantly impacting performance. Recently, there are also some industrial companies and communities working on providing in-cloud traffic processing capabilities [20, 28, 29]. However, it brings up an obvious concern about privacy, because in the new model, both the cloud provider and the middlebox provider may see the user's traffic and the middlebox rules, which may contain sensitive user information. For example, rules of a firewall contain sensitive information such as what traffic

[1]The paper [59] claims that if a data plane update does not change the existing packet behaviors, MDD update can be finished in tens of milliseconds. However from examining update traces of the Route Views Project [26], it is unlikely that a data plane update does not change the existing packet behaviors.

is not welcome, and its leakage could expose a severe security hole. How to perform generic computing in the cloud while keeping the privacy of data has been studied extensively. The introduction of the hardware enclaves (e.g., Intel SGX [81]) provides a way to perform generic private computation; it can verify the binary before running it and can encrypt data before storing the data to enclave memory. However, this approach assumes one knows the hash of a correct binary [90] and thus cannot prevent a *curious* middlebox provider from leaving a backdoor in the middlebox. Moreover, current implementations of enclaves still suffer from side channel attacks [116]. In another approach, the user can encrypt packets before sending them to the cloud/middleboxes, and previous works have studied how middleboxes can perform computation over encrypted data. These solutions are usually not generic, but it turns out that most middlebox functionalities only need a limited number of operations. For example, keyword matching, which is widely used for intrusion detection, can be performed efficiently over encrypted data [104, 122].

One key challenge of the cryptographic approaches is how to handle packet headers. Headers are involved in both middlebox processing and traffic steering [96, 125] (e.g., route all HTTP traffic through firewall-IDS-proxy), which need to detect whether or not an address lies within a range of values (e.g., if a header belongs to a prefix). With traditional IP addresses, one can implement such a rule matching efficiently by aggregating IPs from the same subnet because they share the same prefix. When headers are encrypted, however, such prefix property is lost, and building a lookup table using keyword matching, though possible, will create a memory explosion. Moreover, because of the dynamic nature of the network, the matching rules may change at runtime, and an ideal solution should not incur high overhead when the network configuration is changed. In summary, an ideal mechanism to handle packet headers should achieve three properties. First, the cloud and middlebox should be able to fulfill

its functionalities without learning the user's packet headers. Second, the mechanism should incur low processing overhead at both the enterprise side and the middlebox side, so that they can process packets at high speed. The mechanism should not consume much extra bandwidth because cloud providers usually charge traffic redirected to the cloud by volume. Third, allow operators to frequently modify network configurations (e.g., rerouting traffic to backup middlebox instances; changing the Access Control Lists (ACLs) of a firewall) to perform tasks, ranging from traffic engineering to patching security vulnerabilities [98]. SDN/NFV provides the ability to update a middlebox instance or launch a new one and reroute traffic to the new instance in a matter of milliseconds [79]. To support frequent rule updates, an ideal secure middlebox outsourcing mechanism should be able to update incrementally, i.e. the overhead of performing such an update should be proportional to the number of rules to be changed. So far, none of the existing mechanisms can achieve all of the properties listed.

Checking network intents for stateful networks. Intent-Based Networking (IBN) [13] is the new advent in network automation. IBN aims to make networks more reliable and efficient by automatically converting network-wide objectives, called *intents* (e.g., all critical services in the data center are available to remote sites) into detailed network configurations that implement those intents. While IBN eases the configuration task for network administrators, it faces several challenges. The first challenge is handling undetected bugs and inaccuracies in the automation logic itself given that the need to dealing with the diversity of network devices and services effectively is hard. The second challenge is the subjective nature of *intents*, which cannot be completely fulfilled by automation and might need human intervention to provide input or make changes that are not supported by the automation framework.

Stateful networks refer to the networks that contain stateful NFs. According to a survey in [91], 43% of network intent violations involve NFs, and between 4% and

15% of them are the result of NF misconfiguration. However, recent work on network verification either only ensures correct NF traversal assuming all instances of each type of NFs are equally and correctly configured [35, 52, 63], or only checks NF configurations in a restricted scope that may lose end-to-end expressiveness and accuracy [88]. We have identified three key requirements of an intent checking system for stateful networks: First, vendor-agnostic model specifications to support diverse NFs and their configurations from different vendors. Second, completeness to support end-to-end intent checking, to handle packet header modifications by NFs and routing dynamics, and Third, Incremental checking to efficiently check correctness to avoid performing full checking for every change. Existing network verification work consists of two approaches: The **customized approaches**, such as HSA [63] and its real-time version, NetPlumber [62], identify the set of packets affected by the network changes and utilize customized path-based algorithms to calculate their new forwarding paths. This approach is unable to model extra packet sequences from other parts of the network and thus cannot be used for stateful networks. The **solver-based approaches**, such as Minesweeper [35] and VMN [88], encode all possible packet behavior within the network using first-order logic. To achieve scalability with modern solvers, such as SAT [78] and SMT (Satisfiability Modulo Theories) [44], they rely on optimizations to identify logically independent network slices. However, there is no guarantee that these slices will always have moderate size or even exist, especially when there are NFs that modify packet headers. Further, both Minesweeper and VMN solve all constraints as a whole, and cannot reuse previous checking results when the network changes.

Automated network intent inference. While the above intent statements appear promising and straightforward, creating or updating intents may be challenging for multiple reasons. Firstly, the operator may not be aware of what the intent is or to which endpoints the labels in an intent refer (e.g., where is the "secure zone", "conferenc-

ing application", etc?) because the operator may be a novice, or because the original architects of the network are no longer available and the documentation is poor. This covers a common scenario in the upgrade procedure of legacy networks to IBN since those networks were originally built without having formally defined intents stored in the system. Recently, some advanced platforms provide centralized interfaces for operators to ease the management of network-wide configurations [4, 5], but they still lack visibility of existing running intents implemented in underlying networks. Even if the operator knows what the intent should be, the network may be very large and have many goals, so it would take significant time to describe all intents, which may number in the hundreds or thousands.

Recent advances in network verification are able to formally verify network control and data plane to efficiently check for unwanted behavior efficiently. Therefore, an intuitive approach to infer intents is to look at those configurations as they are supposed to implement the original network intents. However, configurations may not reflect the network runtime states. Indeed, configurations are usually installed by multiple operators to address different business goals on a long time scale. In practice, configurations often contain a large amount of outdated snippets and mixed residues of obsolete policies [16]. Therefore, inferring network intents solely based on configurations may incur inaccurate and misleading results. To help operators learn the high level insights of runtime states, [39] proposed to run a sophisticated heuristic over the whole network-wide packet forwarding records to extract a summary of the network runtime state. This method may require a large storage capacity to store all forwarding behavior and is not suitable for online use. A more commonly adopted approach to understand runtime states by network operators is random packet sampling. By design, random sampling provides no guarantee on which traffic flows will be sampled, by which router and at what time. Except for a few heavy-hitters [124], even minutes-long collections of ran-

	Throughput	Minimum Overhead (per packet)	Incremental Update	Function Chain	Security Guarantee
Melis et al. [82]	very low	119 Bytes	✗	✗	high
Embark [71]	high	20 Bytes	✗	✓	Possible leakage of packet headers and rules
Splitbox [33]	medium	$> 2\times$ traffic	✗	✗	Possible leakage of packet headers and rules
SafeBricks [90]	$<$ Embark	-	✗	✓	Side-channel attack
SICS	$>$ Embark	4 Bytes	✓	✓	high

Table 1.1: Comparison of existing secure middlebox outsourcing schemes.

dom samples typically provide coarse-grained and inaccurate bandwidth estimations for the large majority of flows.

1.1 Overview of Dissertation

This dissertation presents methods to address the above challenges and missing functionalities. The dissertation is comprised of four parts:

- We present a network-wide packet behavior identification tool called AP Classifier, where AP stands for Atomic Predicates, a concept developed in [118]. Each atomic predicate specifies a set of packets that have the same forwarding behavior in the network. The motivation of using atomic predicates is stated as follows. Existing solutions of packet behavior identification that use forwarding table simulation or BDD-like structures [42] are slow in processing queries and memory-inefficient because every bit of the packet header is considered. The concept of atomic predicates [118] provides a way to compress ACLs and forwarding rules to a small set of equivalence classes that can be specified efficiently. We hence develop a novel data structure, called AP Tree, to classify packets into atomic predicates which allows us to eliminate the primary cause of inefficiency by using a BDD-like structure to analyze packet flow behavior. The packet behavior can then be easily computed using the atomic predicate. To further increase the

performance, AP Classifier employs optimized construction algorithms, so that the constructed AP Tree achieves higher query throughput. To deal with network dynamics, AP Classifier utilizes a real-time update to maintain query correctness and an AP Tree reconstruction method that periodically rebuilds the tree to optimize its performance. We evaluated the performance of AP Classifier using the data plane network state, including forwarding tables and ACLs, from two real networks: Internet2 [14] and a Stanford campus network [63]. Our results show that AP Classifier, running on a general-purpose desktop computer, uses only a few MBs memory and supports more than two millions of queries per second. In addition it can be updated in real time ($<$ 4 ms for 95% updates in Internet2 and $<$ 1 ms for 95% updates in Stanford).

- We present a middlebox outsourcing scheme SICS, short for Secure In-Cloud Service chaining. SICS protects the private information of packet headers and rules by only allowing packets with encrypted headers into the cloud. However, encrypted headers cannot be used for forwarding and middlebox rule matching. Inspired by the concept of forwarding equivalence classes in packet forwarding networks [112, 118], SICS assigns a label to each encrypted packet. Each label uniquely identifies the forwarding and rule-matching behavior of the packet. To apply forwarding equivalence classes for middlebox outsourcing, there are key domain-specific challenges. First, middlebox policies typically require a set of packets to go through a sequence of middleboxes, which is called a service function chain [23]. Those independently specified polices should be efficiently combined for packets that are subject to multiple requirements. Second, most middleboxes employ stateful processing and may modify packet headers (e.g., a source NAT converts internal addresses to external ones). However, forwarding equivalence classes can only analyze forwarding behavior of static networks [112] and

cannot be directly used to handle the complexity and dynamics in middlebox chaining. To address these challenges, we first logically group packets with the same middlebox processing chain and actions into policy equivalence classes and thus we eliminate the need to assign a unique label to every single flow. Second, building on configurations for header transformation, we propose a label-to-label replacement scheme. The new labels correspond to the new modified headers and are used for subsequent processing. Table 1.1 summarizes results from evaluations and compares SICS with the four recent secure middlebox outsourcing schemes in five desired properties: throughput, bandwidth overhead, incremental update, service function chaining, and security guarantee. SICS achieves all of the desired properties, while every other design contains several weaknesses. Note that SICS focuses on how to handle packet headers securely. Similar to previous work [71], SICS is compatible to existing secure Deep Packet Inspection (DPI) over encrypted traffic and can be perfectly combined with existing methods [104, 122] to handle the whole packet securely.

- We present Epinoia, an intent checker for stateful networks. Table 1.2 shows a comparison of Epinoia with other related work in terms of support for the key requirements described above. To the best of our knowledge, Epinoia is the only system that can fully support all the key requirements. Epinoia includes a novel configuration model for NF function units represented as vendor-agnostic extensions of OpenConfig YANG models [84] that can be combined to represent configurations of commercial advanced NFs [87]. We propose new techniques leveraging causality precedence relationships [97] between packet I/Os and NF states to represent stateful NF operating logic. Efficiency is achieved by the design of a scalable yet correct approach for intent checking based on intent decomposition and incremental checking using a novel causality graph memoization technique

for all checked results. We have conducted a comprehensive evaluation of Epinioa using a real-world dataset and topologies. Epinioa can perform incremental checking within a few seconds per network and/or intent update which reduces the time cost by up to a factor of 100x compared to an exhaustive checking for all intents.

- We present AutoInfer, a tool that infers intents of a running network automatically, and serves as a starting point for operators to update existing or create new network intents. AutoInfer exploits the possibility to augment state of the art configuration analysis techniques with runtime monitoring, by enabling fine-grained measurement of potential intents obtained from configurations. AutoInfer coordinates intent monitoring among switches and schedules small monitoring tasks for selected intents, within hundreds of milliseconds. This enables AutoInfer to efficiently capture accurate runtime intents. Endpoints sharing the same intents are grouped together, suggesting operators can create new labels and reorganize their networks. With AutoInfer intents are displayed in intent graphs following an existing intent specification called the Policy Graph Abstraction (PGA) [92]. The choice is motivated by the intuitive graph representation of network intents, support of network function boxes, and the fact that PGA ideas have been included in the OpenDaylight (ODL) Network Intent Composition (NIC) project [18]. Even with a reduced scope after analyzing configurations, determining which intent to monitor, where and when, is still challenging. Biased strategies can lead to poor coverage and inaccurate results (e.g., if all resources are used to monitor intents with popular destinations). Conversely, strategies for strict fairness can lead to wasted resources and slow answers (e.g., if many intents are inactive). These are the challenges AutoInfer tackles on behalf of operators. Given the intents obtained from configurations, it derives a set of aggregated intents and adds

Table 1.2: Epinoia vs. other network verification tools
(○unsupported ◐partial support ●support)

	Vendor-agnostic NF models	Header transformation	Incremental checking
HSA/NetPlumber [63] [62]	○	●	●
Minesweeper [35]	◐	○	○
VMN [88]	●	◐	○
Epinoia	●	●	●

them to the potential intents for monitoring. Then, it utilizes an adaptive monitoring refinement scheme following three key ideas. First, AutoInfer decides what (which intent) and where to monitor all potential intents, with objectives to maximize both intent coverage and switch resource utilization. Second, AutoInfer adaptively refines the monitoring plan based on previous monitored results and quickly narrows the monitoring tasks down to only active intents at the moment. Third, for each monitoring plan change, AutoInfer runs a greedy heuristic that utilizes intent information to minimize the number of monitoring spot changes, reducing the communication overhead and time delay to reset monitoring tasks on switches. Our evaluation results based on an AutoInfer prototype show that AutoInfer quickly captures all active intents and increases monitoring efficiency by at least 3.7x compared with a fixed schedule. We also show that AutoInfer computes monitoring schedules for large networks, with hundreds of nodes in tens of milliseconds and is therefore suitable for online use.

Chapter 2

Practical Network-wide Packet Behavior Identification by AP Classifier

2.1 Model and Background

We model a network as a directed graph of boxes, each of which has a forwarding table as well as input and output ports guarded by access control lists (ACLs). Each packet has a fixed-size header, including all fields that are evaluated by forwarding tables and ACLs in the network. A flow is then a sequence of packets that have the same values in the evaluated header fields.

Following the concepts in [118], forwarding tables and ACLs are all packet filters. Each ACL can be specified by a *predicate*. The set of packets that are allowed by the ACL are evaluated to true by the predicate. Similarly, by analyzing a forwarding table, each output port can be specified by a forwarding predicate. The set of packets that

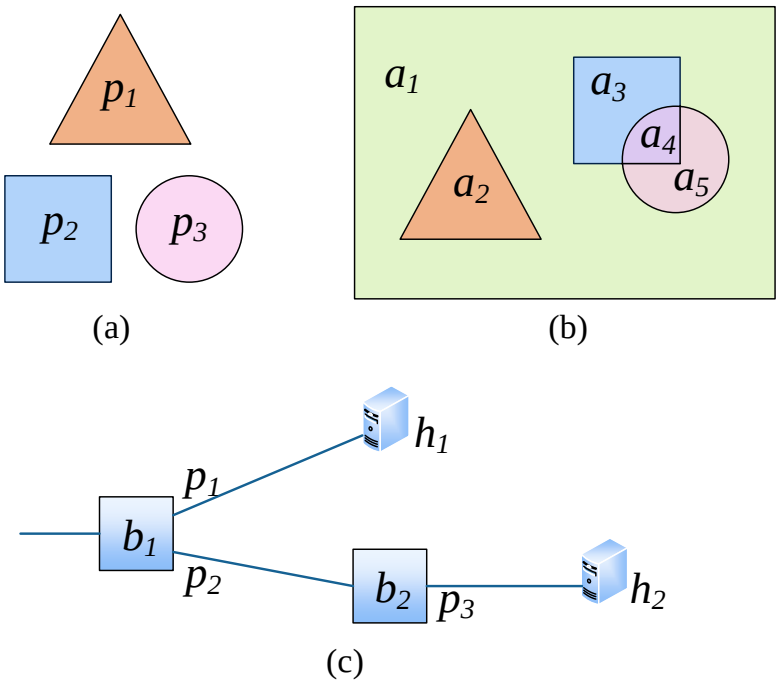


Figure 2.1: (a) Three predicates. (b) The packet header space and five atomic predicates. (c) A sample network including the three predicates.

can be forwarded to the port are evaluated to true by the predicate.[1] Forwarding tables and ACLs can be converted to predicates using the algorithms in [118]. A predicate P specifies the set of packets for which P evaluates to true. Hence if a packet can travel through a sequence of packet filters, it is evaluated to true by the conjunction of predicates corresponding to the packet filters.

Given a set of predicates, we can compute a set of atomic predicates. Due to space limitation, we do not repeat the formal definition of atomic predicates, which can be found in [118]. A proved property of the set of atomic predicates is that they specify the minimum set of equivalence classes in the set of all packets. The packets that are evaluated to true by the same atomic predicate have identical behaviors at all boxes. For a set of predicates $P = \{p_1, p_2, ..., p_k\}$, each atomic predicate a_i is in the form

[1] All predicates are represented by binary decision diagrams (BDDs) [42] in our implementation of AP Classifier.

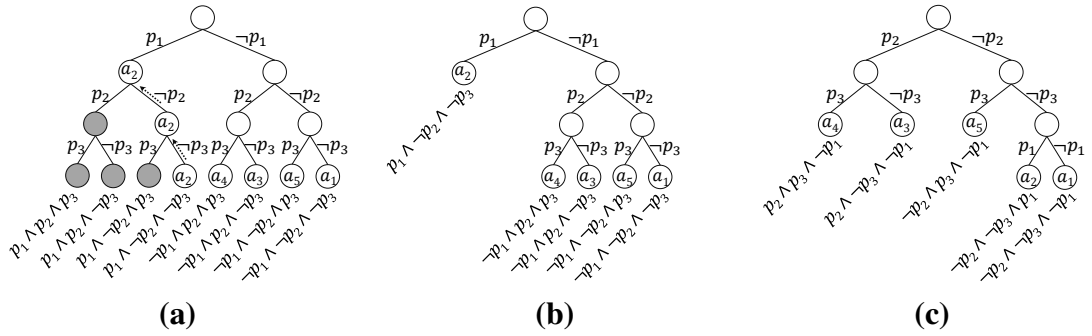


Figure 2.2: AP Tree of predicates in Fig. 2.1(b). (a) Original AP Tree. (b) Pruned AP Tree. (c) Optimized AP Tree.

$a_i = q_1 \wedge q_2 \wedge ... \wedge q_k$, where $q_j \in \{p_j, \neg p_j\}$. (Note that a_i in the previous sentence is an atomic predicate only if it is not false.) Every predicate is equal to the disjunction of a subset of atomic predicates. Every packet is evaluated to true by one and only one atomic predicate.

As an illustration, Fig. 2.1(a) shows three predicates p_1 (triangle), p_2 (square), and p_3 (circle), each of which represents a set of packets that are evaluated to true by a predicate. Each predicate specifies a set of packets that can pass the corresponding packet filter. Fig. 2.1(b) shows the three predicates in the packet header space. All packets in this example can be classified into five equivalence classes specified by five atomic predicates, a_1 to a_5. Each predicate is equal to the disjunction of a subset of atomic predicates. For example, $p_2 = a_3 \vee a_4$. Also, $a_4 = \neg p_1 \wedge p_2 \wedge p_3$. All packets evaluated to true by a_4 have identical behaviors: they can pass the filters of p_2 and p_3 but cannot pass p_1.

In the network shown in Fig. 2.1(c), Let p_1 specify the set of packets that can be forwarded at box b_1 to its output port to host h_1, p_2 specify the set of packets that can be forwarded at box b_1 to its output port to box b_2, and p_3 specify the set of packets that can be forwarded at box b_2 to its output port to host h_2. A packet specified by $a_4 = \neg p_1 \wedge p_2 \wedge p_3$ is forwarded at b_1 by the path $b_1 -> b_2 -> h_2$. A packet specified

by $a_5 = \neg p_1 \wedge \neg p_2 \wedge p_3$ is forwarded to h_2 if it is at b_2, but will be dropped if it is at b_1. An atomic predicate characterizes the behaviors of all packets it evaluates to true.

2.2 Design Framework of AP Classifier

AP Classifier is a program designed for a SDN controller. It computes the network-wide behaviors for an input packet (or flow). AP Classifier performs two-stage processing for a packet. First, using the AP Tree, it classifies the packet to the atomic predicate that evaluates to true for the packet. Second, AP Classifier determines all forwarding paths for the packet by using the atomic predicate, network information, and ingress box of the packet.

2.2.1 AP Tree

Using the algorithms presented in [118], the controller first converts each ACL to a predicate and the forwarding table of each box to m predicates, where m is the number of output ports of the box. Let $P = \{p_1, p_2, ..., p_k\}$ be the set of predicates of all boxes in the network. The controller constructs an *AP Tree* which is a binary tree. The root is labeled by p_1. At level i, the 2^i internal nodes are each labeled by p_i. Starting from the root, at each internal node, the input packet is evaluated by the predicate in the label. If the result is true, the packet continues to be evaluated in the left sub-tree. Otherwise it goes to the right sub-tree. An AP Tree with $(k + 1)$ levels can be constructed from evaluating each of the k predicates at each level of internal nodes. A leaf node is then labeled by $q_1 \wedge q_2 \wedge ... \wedge q_k, q_i \in \{p_i, \neg p_i\}$, which specifies the set of packets reaching the leaf. Fig. 2.2(a) shows the AP Tree of the three predicates in Fig. 2.1(b). Shaded circles indicate leaf labels that are false. We will show that two sub-trees in an AP tree do not necessarily have a same predicate order in Section 5.3.

To classify a packet to an atomic predicate, AP Classifier simply searches the AP Tree by evaluating the packet until the leaf labeled by the atomic predicate is found. At each node, the packet is evaluated by checking the BDD of the predicate. Since predicates on sibling nodes are disjoint, for a given packet, the path from the root to the leaf is exclusive and determinate.

In the worst case, there could be 2^k atomic predicates and finding a leaf needs to evaluate all k predicates. However, it is found that the number of atomic predicates is surprisingly small for real networks [118]. Hence many leaves specify empty sets of packets. For example, in Fig. 2.2(a), $p_1 \wedge p_2 \wedge p_3$, $p_1 \wedge p_2 \wedge \neg p_3$, and $p_1 \wedge \neg p_2 \wedge p_3$ are all false according to the relationships in Fig. 2.1(b). Hence no packet can reach any of these three leaves. We use the following rule to "prune" the AP Tree: If no packet reaches a sub-tree, i.e., all leaves in the subtree are labeled by false predicates, the sub-tree is removed from the AP Tree. If an internal node has only one child, it is removed from the AP Tree as there is no need to check the predicate. We define the depth of a leaf to be the number of predicates evaluated to reach the leaf. After pruning, the average depth of all leaves in the AP Tree can be reduced and each node has either 0 or 2 children. Fig. 2.2(b) shows the pruned AP Tree has average depth $(1 + 3 + 3 + 3 + 3)/5 = 2.6$.

An important observation is the following: If predicates are placed at the levels in a different order, the average depth of the AP Tree may be different. In Fig. 2.2(c), the predicates are placed at three levels in the order of p_2, p_3, p_1. The average depth of all leaves in the pruned AP Tree is 2.4. An important contribution of this work is an algorithm to find an order of predicates that substantially reduces the average depth of an AP Tree.

For examples, each of the Internet2 and Stanford networks includes hundreds of thousands of forwarding rules, which can be converted to 161 (Internet2) or 507 (Stan-

ford) predicates. Using our AP Tree construction algorithm, the average depth of the AP Tree is only 10.6 (Internet2) or 16.8 (Stanford). In an unpruned AP Tree, a packet needs to be evaluated by 161 or 507 predicates. AP Classifier only requires it to be evaluated by 10.6 or 16.8 predicates, on average, thus improving the query throughput by more than an order of magnitude. The detailed algorithm design of AP Tree construction is presented in Section 2.3.

2.2.2 Computing packet behaviors

The second stage of AP Classifier determines the network-wide behaviors of the queried packet from the network information, the ingress box, and the atomic predicate determined in the first stage.

Since the atomic predicate is in the form $q_1 \wedge q_2 \wedge ... \wedge q_k, q_i \in \{p_i, \neg p_i\}$, for any predicate p_j, AP Classifier can easily check whether the predicate evaluates to true or false for the packet. Recall that p_j represents a packet filter of an ACL or output port. Hence AP Classifier can determine at any box whether the packet is dropped and which port it is forwarded to. Starting from the ingress box, i.e., the box that sees the packet first in the network, AP Classifier finds the output port to which the packet is forwarded and then determines the next-hop box. If the packet is a multicast packet, it may be forwarded to multiple ports. AP Classifier continues to find the forwarding ports on the next-visited boxes until the packet reaches the destination or is dropped. The packet behaviors are thus obtained.

Fig. 2.3 shows an example to illustrate how to compute network-wide forwarding paths for a given packet. Consider a packet which arrives at the ingress box b_1 and it is classified to atomic predicate a_4 by searching the AP Tree. The representation, $\neg p_1 \wedge p_2 \wedge p_3$, of a_4 shows that the packet is forwarded to b_2 because p_1 is false and p_2 is true for the packet. Similarly at b_2, the packet is forwarded to h_2 because p_3 is true

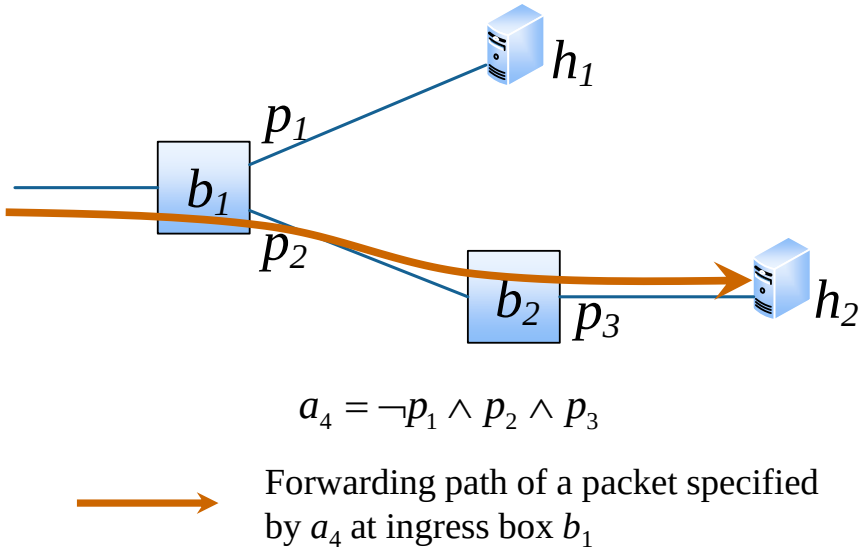


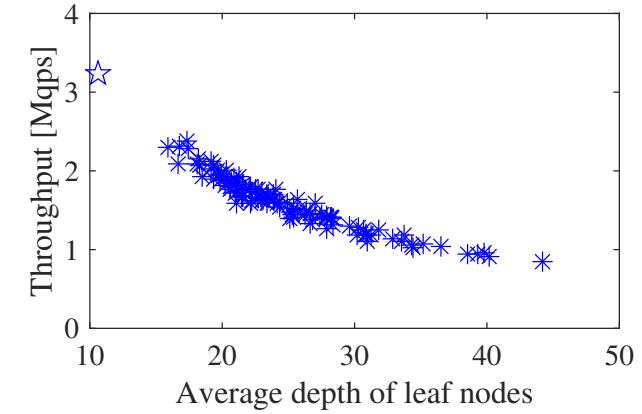
Figure 2.3: Computing forwarding path for a packet in a_4

for the packet.

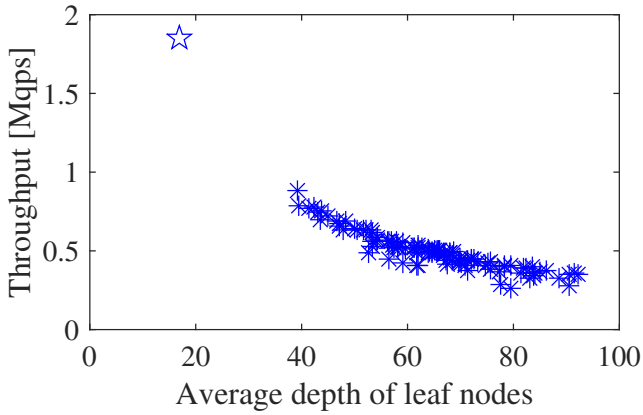
We ran experiments to evaluate the speed of the above approach on a general-purpose desktop computer. We found that, for the Internet2 and Stanford datasets, the throughput is greater than 15M and 10M packets per second, respectively. Note that this throughput is much higher than the throughput in the first stage. Therefore, the main effort of this work is to optimize the construction, search, and update of the AP Tree.

2.3 AP Tree Optimization

The most challenging problem in designing AP Classifier is to construct an AP Tree with minimized average depth, which can support dynamic updates.



(a) Internet2



(b) Stanford

Figure 2.4: Query throughput versus average depth of leaves

2.3.1 Query throughput versus average depth

To reduce the query time and improve the query throughput, the optimization goal of AP Tree construction is to reduce the average depth of leaves. We conduct a set of experiments to justify the correlation of reducing the average depth and improving the throughput. We use the Internet2 network containing 161 predicates and the Stanford network containing 507 predicates. In each experiment, we randomly order the k predicates for placement at levels of the AP Tree. Then we query the generated tree using sample packets and measure the query throughput. In Fig. 2.4, we show the relationship between query throughput and average depth for 100 random generated AP Trees

for each network. After pruning, the average depth of the AP Tree of Internet2 varies from 15.9 to 44.2, and the average depth of the AP Tree of Stanford varies from 39.1 to 92.5. From the two sub-figures in Fig. 2.4, it is obvious that an AP Tree with smaller average depth provides higher query throughput. The star in each figure represents the performance of the AP Tree constructed by AP Classifier. The query throughput of AP Classifier is 3.35 Mqps (Internet2) and 1.82 Mqps (Stanford), substantially higher than any random construction.

2.3.2 Quick-Ordering algorithm

The number of atomic predicates for a network is determinate if there is no update. That is, for a network, its AP Tree has a fixed number of leaves. A more balanced binary tree results in smaller average leaf depth. Compare the two AP Trees in Fig. 2.2(b) and (c) whose average depths are 2.6 and 2.4, respectively. The one in Fig. 2.2(c) is more balanced and hence has less average depth. The reason for the imbalance in Fig. 2.2(b) is that p_1 is placed at a higher level of the tree. According to properties of atomic predicates, every predicate is equal to the disjunction of a subset of atomic predicates. The number varies from one to the number of all atomic predicates. In this example, p_1 is a predicate that is equal to a single atomic predicate. Hence the left child of the node labeled as p_1 must be a leaf representing the atomic predicate. However, the right sub-tree may include more levels, causing the imbalance.

In fact, an analysis of the two real network data planes shows that many predicates are equal to a single atomic predicate. One fast yet effective ordering of predicates is to place those predicates at lower levels. For example, in Fig. 2.2(c), p_1 is placed at the lowest level.

Notation. Let $R(p)$ denote the subset of atomic predicates whose disjunction is p. $|R(p)|$ denotes the cardinality of $R(p)$.

In the Quick-Ordering algorithm, $|R(p_i)|$ is counted for each predicate p_i. Then the AP Tree is constructed by placing all predicates onto the tree in descending order of $|R(p_i)|$.

2.3.3 Optimized AP Tree construction

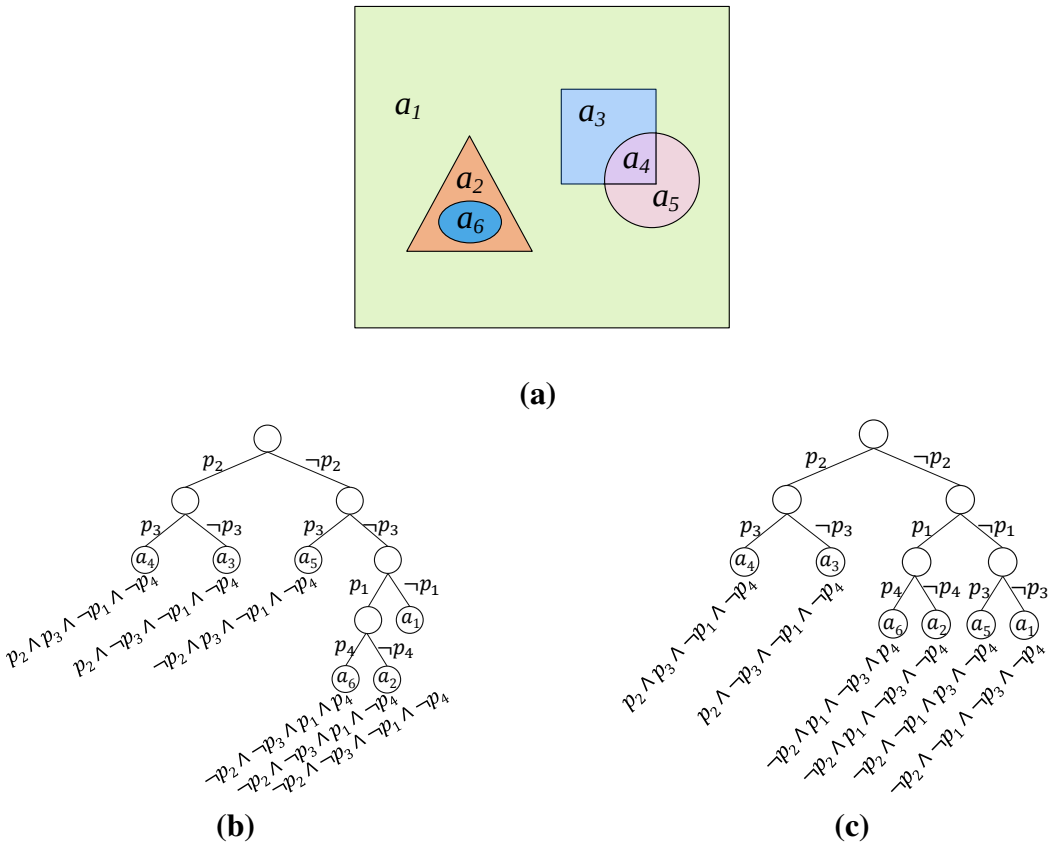


Figure 2.5: Additional example. (a) Five predicates. (b) Pruned AP Tree. (c) Optimized AP Tree.

To develop a more sophisticated ordering method, one important observation is that, for two sub-trees whose roots are siblings, their predicate orders can be different. In the example of Fig. 2.5(a), we now have four predicates p_1 (triangle), p_2 (square), p_3 (circle), and p_4 (ellipse), which determine six atomic predicates, a_1 to a_6. If the

predicates are added in the order p_2, p_3, p_1, p_4, the pruned AP Tree is shown in Fig. 2.5(b). However, for the sub-tree rooted at the right child of the root, its subtree is more balanced if the predicate order is p_1, p_3, p_4, as shown in Fig. 2.5(c).

For a given set of predicates $P = \{p_1, p_2, ..., p_k\}$, the atomic predicates $A = \{a_1, a_2, ..., a_n\}$ is determined. The number of leaves of the AP Tree is n, because each leaf corresponds to an atomic predicate. We define $F(Q, S)$ as the *minimal sum of leaf depths* of the subtree (which is a part of the AP Tree) whose nodes include the set of predicates Q and leaves are the set of atomic predicates S. In the example of Fig. 2.5(c), let $Q = \{p_1, p_3, p_4\}$ and $S = \{a_1, a_2, a_5, a_6\}$, $F(Q, S) = 8$. $F(Q, S)$ can be calculated recursively using the following equations. Let $H(Q, S, p)$ be the minimal sum of leaf depths if the root of the sub-tree is p. If $S \cap R(p) \neq \varnothing$ and $S \cap R(\neg p) \neq \varnothing$, $H(Q, S, p)$ is the sum of three components: $F(Q - \{p\}, S \cap R(p))$ and $F(Q - \{p\}, S \cap R(\neg p))$ are recursive computing for the left and right sub-trees and extra $|S|$ needs to be added because the depth of every leaf increments by 1. We have

$$H(Q, S, p) = F(Q - \{p\}, S \cap R(p)) + F(Q - \{p\}, S \cap R(\neg p)) + |S|$$

If $S \cap R(p) = \varnothing$, the left sub-tree will be pruned. The internal node with only one child is also removed and the leaf depths do not increase. Hence,

$$H(Q, S, p) = F(Q - \{p\}, S \cap R(\neg p))$$

Similarly, if $S \cap R(\neg p) = \varnothing$, we have,

$$H(Q, S, p) = F(Q - \{p\}, S \cap R(p))$$

In addition, we have the following recursive equation.

$$F(Q,S) = \begin{cases} 0 & \text{if } |S| = 1 \\ \min_{p_i \in Q} H(Q,S,p_i) & \text{otherwise} \end{cases} \tag{2.1}$$

When $|S| = 1$, it is easy to see that the sub-tree contains only one leaf, hence $F(Q,S) = 0$. Otherwise, the predicate $p_i \in Q$ is selected as the root of the sub-tree such that p_i minimizes $H(Q,S,p_i)$.

Using the above formula, it is possible to compute $F(P,A)$. By recording the selection of p_i at each recursion, the optimized AP Tree can also be constructed.

However, the time complexity of solving this recursion is as high as $O((2^k) * k!)$, where k is the cardinality of P. We need to propose an efficient heuristic algorithm to simplify the recursion. At a level of recursion, we need to find the predicate p_i that minimizes $H(Q,S,p_i)$. Instead of trying all predicates, we propose an easier way to decide which predicate to select.

We define a pair-wise relation between two predicates that implies which one is better to select. If $H(Q,S,p_i) < H(Q,S,p_j)$, we say that p_i is *superior* to p_j and p_j is *inferior* to p_i, denoted as $p_i \overset{S}{\rightarrow} p_j$. If $H(Q,S,p_i) = H(Q,S,p_j)$, we say p_i and p_j are in the same order, denoted as $p_i \overset{S}{\sim} p_j$.

We compare two predicates in four cases based on their logical relationships, as shown in Fig. 2.6. Here, p_i and p_j refer to predicates which are equal to union of atomic predicates in $S \cap R(p_i)$ and $S \cap R(p_j)$ respectively. $H(Q,S,p)$ is calculated based on the first three equations of section 2.3.3 for all four cases as follows:

1) Packets specified by p_i intersect with those of p_j (Fig. 2.6(a)). If we place p_i to the root and p_j to the children of the root, we get a full sub-tree since $R(p_i) \cap R(p_j)$, $R(p_i) \cap R(\neg p_j)$, $R(\neg p_i) \cap R(p_j)$ and $R(\neg p_i) \cap R(\neg p_j)$ are all non-empty. Hence, we have

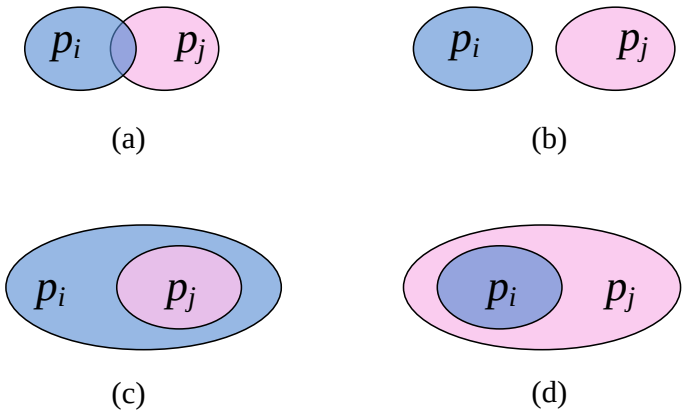


Figure 2.6: Relationships of two predicates. (a) Neither $P_i \wedge P_j$ nor $\neg P_i \wedge \neg P_j$ is false. (b) $P_i \wedge P_j$ is false. (c) $\neg P_i \wedge P_j$ is false. (d) $Pi \wedge \neg Pj$ is false.

$$
\begin{aligned}
H(Q, S, p_i) = |S| &+ F(Q - \{p_i\}, S \cap R(p_i)) \\
&+ F(Q - \{p_i\}, S \cap R(\neg p_i)) \\
&= |S| + F(Q - \{p_i, p_j\}, S \cap R(p_i) \cap R(p_j)) \\
&+ F(Q - \{p_i, p_j\}, S \cap R(p_i) \cap R(\neg p_j)) \\
&+ |S \cap R(p_i)| \\
&+ F(Q - \{p_i, p_j\}, S \cap R(\neg p_i) \cap R(p_j)) \\
&+ F(Q - \{p_i, p_j\}, S \cap R(\neg p_i) \cap R(\neg p_j)) \\
&+ |S \cap R(\neg p_i)|
\end{aligned}
$$

If we place p_j to the root and p_i to the children, we can get $H(Q, S, p_j)$ similarly. Since $|S \cap R(p_i)| + |S \cap R(\neg p_i)| = |S \cap R(p_j)| + |S \cap R(\neg p_j)| = |S|$, $H(Q, S, p_i) = H(Q, S, p_j)$. We have $p_i \overset{S}{\sim} p_j$.

2) Packets specified by p_i disjoint with those of p_j (Fig. 2.6(b)). $p_i \wedge p_j$ is false. If we place p_i to the root and p_j to the children of the root, the sub-tree representing $R(p_i) \cap R(p_j)$ will be pruned. The child representing $R(p_i) \cap R(\neg p_j)$ will replace

its parent node and leaf depths do not increase. However, the sub-tree representing $R(\neg p_i) \cap R(p_j)$ and $R(\neg p_i) \cap R(\neg p_j)$ are both non-empty, so the total leaf depths increase by $|S \cap R(\neg p_i)|$. Hence

$$\begin{aligned} H(Q, S, p_i) = |S| &+ F(Q - \{p_i, p_j\}, S \cap R(p_i) \cap R(\neg p_j)) \\ &+ F(Q - \{p_i, p_j\}, S \cap R(\neg p_i) \cap R(p_j)) \\ &+ F(Q - \{p_i, p_j\}, S \cap R(\neg p_i) \cap R(\neg p_j)) \\ &+ |S \cap R(\neg p_i)| \end{aligned}$$

Similarly, if we place p_j to the root and p_i to the children,

$$\begin{aligned} H(Q, S, p_j) = |S| &+ F(Q - \{p_i, p_j\}, S \cap R(p_j) \cap R(\neg p_i))) \\ &+ F(Q - \{p_i, p_j\}, S \cap R(\neg p_j) \cap R(p_i)) \\ &+ F(Q - \{p_i, p_j\}, S \cap R(\neg p_j) \cap R(\neg p_i)) \\ &+ |S \cap R(\neg p_j)| \end{aligned}$$

Despite of the same terms, if $|S \cap R(\neg p_i)| < |S \cap R(\neg p_j)|$, $p_i \xrightarrow{S} p_j$. If $|S \cap R(\neg p_i)| = |S \cap R(\neg p_j)|$, $p_i \overset{S}{\sim} p_j$. Otherwise $p_j \xrightarrow{S} p_i$.

3) Packets specified by p_j are a subset of those of p_i (Fig. 2.6(c)). $\neg p_i \wedge p_j$ is false. If we place p_i to the root and p_j to the children of the root, the sub-tree representing $R(\neg p_i) \cap R(p_j)$ will be pruned. The child representing $R(\neg p_i) \cap R(\neg p_j)$ will replace its parent node and leaf depths do not increase. The sub-tree representing $R(p_i) \cap R(p_j)$ and $R(p_i) \cap R(\neg p_j)$ are non-empty, so the total leaf depths increase by $|S \cap R(p_i)|$.

Hence

$$\begin{aligned} H(Q,S,p_i) = |S| &+ F(Q-\{p_i,p_j\}, S\cap R(p_i)\cap R(p_j)) \\ &+ F(Q-\{p_i,p_j\}, S\cap R(p_i)\cap R(\neg p_j)) \\ &+ F(Q-\{p_i,p_j\}, S\cap R(\neg p_i)\cap R(\neg p_j)) \\ &+ |S\cap R(p_i)| \end{aligned}$$

If we place p_j to the root and p_i to the children of the root, the sub-tree representing $R(p_j)\cap R(\neg p_i)$ will be pruned.

$$\begin{aligned} H(Q,S,p_j) = |S| &+ F(Q-\{p_i,p_j\}, S\cap R(p_j)\cap R(p_i))) \\ &+ F(Q-\{p_i,p_j\}, S\cap R(\neg p_j)\cap R(p_i)) \\ &+ F(Q-\{p_i,p_j\}, S\cap R(\neg p_j)\cap R(\neg p_i)) \\ &+ |S\cap R(\neg p_j)| \end{aligned}$$

Therefore if $|S\cap R(p_i)|< |S\cap R(\neg p_j)|$, $p_i \xrightarrow{S} p_j$. If $|S\cap R(p_i)|= |S\cap R(\neg p_j)|$, $p_i \overset{S}{\sim} p_j$. Otherwise $p_j \xrightarrow{S} p_i$.

4) Packets specified by p_i are a subset of those of p_j (Fig. 2.6(d)). Similar to the above cases, we can get if $|S\cap R(\neg p_i)|< |S\cap R(p_j)|$, $p_i \xrightarrow{S} p_j$. If $|S\cap R(\neg p_i)|= |S\cap R(p_j)|$, $p_i \overset{S}{\sim} p_j$. Otherwise $p_j \xrightarrow{S} p_i$.

We then design the key criterion of predicate selection for each level of recursion, namely: We select a predicate that is not inferior to any other predicate. The algorithm is presented as follows: For each level of recursion, a predicate p_s is maintained, initially being p_1. A linear scan is performed from p_2 to p_k. For a predicate p_i, if $p_i \xrightarrow{S} p_s$, then p_s is set to p_i. At the end, p_s is selected as the root node of the subtree for this level of recursion.

To prove the correctness of the above algorithm, we need to show that p_s is indeed

not inferior to any other predicate. A sufficient condition is that the superior/inferior relation is acyclic, i.e., there are no three predicates p_a, p_b, p_c such that $p_a \xrightarrow{S} p_b$, $p_b \xrightarrow{S} p_c$, and $p_c \xrightarrow{S} p_a$. We have proved the acyclic property by exhaustion. Our proof is not shown herein due to space limitation.

Time efficiency of AP Tree construction. In the AP Tree construction algorithm presented above, we avoid the time-intensive operation of computing the conjunction of two predicates represented as BDDs. Instead, our algorithm computes the intersection of two sets of integers that are identifiers of atomic predicates, as suggested in [118]. Intersections of integer sets can be computed much more quickly than conjunctions of BDDs. Each predicate is represented as a set of integers, so the time complexity of determining relationship between two predicates is $O(n)$, where n is the number of atomic predicates. For each level of recursion, a linear scan needs $O(k'n)$ time, where k' is the number of predicates in the current level. The overall complexity of building an AP Tree depends on the number of levels as well as the balance of the tree. Here we only provide the complexity analysis for a balanced AP Tree. For a balanced AP Tree, there are 2^l nodes at level l. For each node, $k' \leq (k - l)$. Hence at level l, the time complexity is at most $2^l(k - l)n$. Since $l \leq \log_2 n$, $2^l(k - l)n < kn^2$. Since there are $\lceil \log_2 n \rceil$ levels, the overall time complexity is upper-bounded by $O(kn^2 \log n)$.

2.3.4 Optimization for packet distribution

In the proposed algorithms, we assume that, for a packet query, leaf nodes (atomic predicates) have equal probability to be visited. Therefore minimizing the average depth of leaf nodes maximizes the query throughput. However, practical network flows may not be distributed uniformly with respect to the set of atomic predicates. For example, if many queried packets may eventually visit a leaf in a very deep position and leaves close to the root are rarely visited, the throughput decreases. To improve the

query throughput for uneven packet distribution, we assign weights to atomic predicates such that leaf nodes that are visited frequently will be placed relatively close to the root.

To estimate the packet distribution, AP Classifier maintains a counter for each leaf node (atomic predicate), which records the number of visits by queries in a past period of time. The value of a counter is then converted to the weight of the corresponding atomic predicate after reduction of a fraction. When using the optimized algorithm presented in Section 2.3.3, every occurrence of $|R(p_i)|$ is replaced by the sum of weights of all atomic predicates in $R(p_i)$, rather than its cardinality.

For example, suppose AP Classifier is choosing the root of a subtree by comparing two predicates p_i and p_j whose relationship is as shown in Figure 2.6(c). If the atomic predicates in set $R(p_j)$ have been queried by many packets, we prefer to place p_j before p_i in order to get smaller depths for the leaf nodes labeled by the atomic predicates in $R(p_j)$. Higher weights help to get $H(Q, S, p_j) < H(Q, S, p_i)$ and make p_j superior to p_i.

2.3.5 Dealing with packet header changes.

Today's networks rely on a wide range of middleboxes (e.g., firewalls, intrusion detection and prevention systems, and proxies) which achieve performance and security benefits. Some middleboxes may modify packet headers of incoming traffic. When middleboxes modify packet headers, the forwarding behaviors of these packets on downstream boxes must be determined by the new header fields. For example, when a Network Address Translation (NAT) middlebox translates an external address to an internal one, AP Classifier must be aware of such translation and compute the remaining packet behaviors using the internal address.

We consider three types of packet header changes by middleboxes, namely 1) deterministic based on packet headers, 2) deterministic based on packet payload, and 3)

probabilistic.

For Type 1 changes, a change is completely determined by the header of an incoming packet. In AP Classifier, we model these middlebox operations as a flow table. Each packet that enters a middlebox passes through a flow table. A flow table contains entries consisting of three components: match fields, instructions, and a new atomic predicate. Match Fields are used to select packets that match the predicates in the fields. Instructions specify new packet headers if a match occurs. The atomic predicate fields store atomic predicates calculated for new packet headers.

For Type 1 changes, given the packet header before a change, the atomic predicate after the change can be easily determined based on the flow table. Therefore when AP Classifier finds that a packet passes a middlebox, at the behavior computing stage (second stage of AP Classifier), it checks the flow table whether the packet header has been modified based on the middlebox policies. If the packet has a new header, AP Classifier will read a new atomic predicate and compute forwarding behaviors for the new header based on the new atomic predicate. Such process may repeat multiple times until the packet is dropped or the forwarding path ends at the packet's destination.

To see how this works, we use an extensional version of the example from 2.2.2 in Fig. 2.7. The topology in the figure is a part of the whole network. Packets passing box b_1 are firstly processed by the flow table at middlebox MB_1 and then by b_1's forwarding table. The flow table of MB_1 contains three entries that modify packet headers and one default entry. Consider a packet enters box b_1 and matches the third entry of the flow table at MB_1. Its corresponding packet header fields are changed to 172.16.146.2 and its atomic predicate is changed to a_4. The yellow line, in Fig. 2.7, shows that the packet is forwarded to box b_2 and then host h_1 after header modification.

For Type 2 changes, the packet header after a change can be determined only after the packet payload is known. Hence it is not possible to pre-compute a flow table that

The flow table at MB_1

Match fields	Instructions	New atomic predicates
10.10.50.0/24	172.16.178.230	$a_2 = p_1 \wedge \neg p_2 \wedge \neg p_3$
10.10.60.0/24	172.16.158.49	$a_3 = \neg p_1 \wedge p_2 \wedge \neg p_3$
10.10.70.0/24	172.16.146.2	$a_4 = \neg p_1 \wedge p_2 \wedge p_3$
Others	None	Unchanged

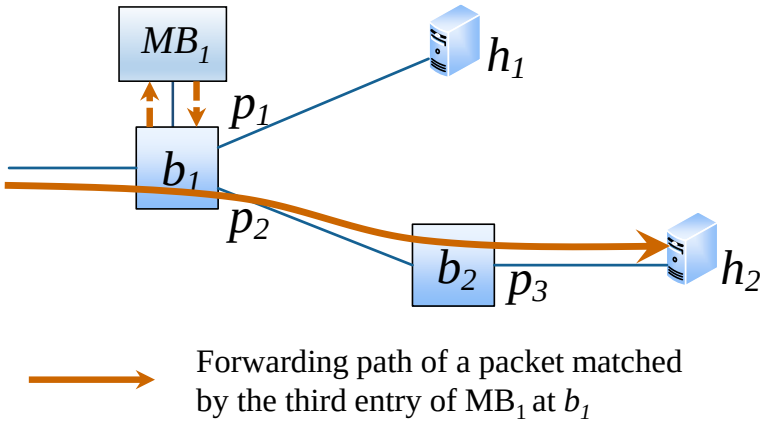


Figure 2.7: Computing forwarding path with header modifications

stores the atomic predicate after packet header changes. AP Classifier needs to search the AP Tree again using the new header to find a new atomic predicate. This process may repeat multiple times. Probabilistic changes (Type 3) can be treated similarly. However, AP Classifier may output multiple possible network-wide behaviors for a given packet.

2.4 AP Tree update and reconstruction

An important requirement of practical packet behavior identification is to support dynamic network changes, including link and rule changes, both of which require addition and deletion of predicates. We design fast AP Tree update methods for adding a predicate and deleting a predicate while maintaining tree correctness. However, after a large number of updates, an AP Tree will experience performance degradation. Hence

we also design an AP Tree reconstruction method that periodically rebuilds the tree to optimize its performance while performing packet query processing at the same time. In this section, we assume that each atomic predicate is equally weighted.

2.4.1 Real-time update of an AP Tree

The SDN data plane of a network is frequently updated by rule installation and deletion. When a rule is inserted into or removed from a forwarding table or an ACL, it may change one or more predicates. The set of atomic predicates may change as well. We use the method presented in [117] to convert a rule insertion or deletion to predicate change. If there is no predicate change after a rule update, AP Classifier does not need to update the AP Tree. Otherwise, AP Classifier performs the methods presented below to remove the old predicate and add the updated predicate in the AP Tree. These methods are also used after addition/deletion of a network link which requires addition/deletion of predicates.

Add a predicate. When a new predicate p is added, for each leaf node representing an atomic predicate a in the current AP Tree, AP Classifier computes $a \wedge p$ and $a \wedge \neg p$. If none of them is false, two children are added to the leaf node, representing $a \wedge p$ and $a \wedge \neg p$ respectively. If one and only one of the two conjunctions is false, the label of the leaf node is replaced by the other conjunction. If both conjunctions are false, AP Classifier does nothing to this leaf node.

Delete a predicate. To delete an existing predicate p from the AP Tree, AP Classifier does not remove all internal nodes labeled by p. This is because after the removal of a node, merging the two sub-trees rooted at its children is very difficult. Instead, we still keep p in the AP Tree, but mark it as "deleted" in the list of all predicates. A query packet is still processed by the AP Tree to find its leaf node representing its atomic predicate. It is still evaluated by the deleted predicates to determine which sub-

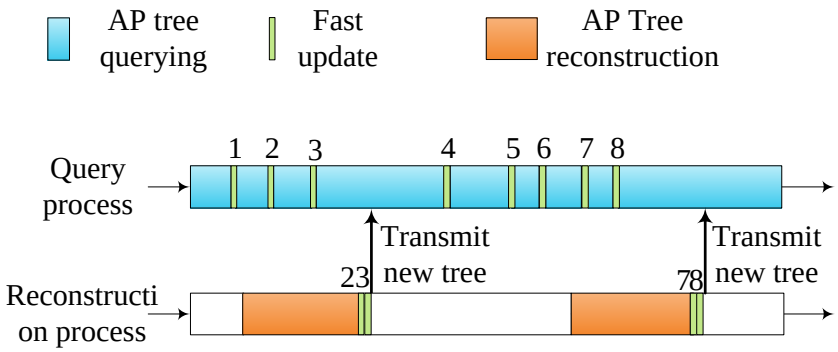


Figure 2.8: Real-time update and query processing

tree to visit next. However, in the second stage of AP Classifier, i.e., computing packet behaviors, AP Classifier just ignores all predicates that have been deleted.

2.4.2 Parallel reconstruction of an AP Tree

Although, the AP Tree updates in AP Classifier are fast and maintain correctness of packet behavior identification, the AP Tree is no longer optimized and the query throughput will degrade over time. Hence AP Classifier also reconstructs the AP Tree to optimize it from time to time. To enable query processing at the same time as tree reconstruction, AP Classifier runs two processes in parallel, called the query process and reconstruction process, executing on two different cores. The start of a reconstruction is triggered by an event, e.g., query throughput is lower than a threshold or the number of updates on the current AP Tree is higher than a threshold. During reconstruction, the query process still maintains the old AP Tree by performing updates, and responds to queries. After the reconstruction process has built a new tree, the new tree needs to be updated for data plane changes that have occurred during the reconstruction period, if any. The updated new tree is then transmitted to the query process to replace the old tree.

Fig. 3.7 shows an example of the parallel reconstruction of an AP Tree. The query process performs AP Tree search to respond to queries as well as updates when data

Table 2.1: Statistics of the two real networks

	Stanford		Internet2
No. of rules	Forwarding	ACL	Forwarding
	757170	1584	126017
No. of predicates	507	71	161
No. of atomic predicates	494	21	216

plane changes happen. In this example, the first reconstruction starts shortly after the change that requires update 1, which is included in the construction of a new tree. However, when the new tree is finished, two changes that require updates 2 and 3 have occurred during the reconstruction period. The new tree does not reflect these two updates. Thus the reconstruction process also applies these two updates to the new tree. Then the updated new tree is sent to the query process to replace the old AP Tree. Similarly the second reconstruction begins after changes that require updates 4, 5, and 6. The new tree constructed needs to be updated for changes (that require updates 7 and 8) which occur during the reconstruction period, before it can be sent to the query process. Note that if there is no data plane change during a reconstruction period, the new AP Tree is optimized.

If network dynamics change weights of atomic predicates, current AP Tree constructed using previous configurations should be rearranged to provide the best performance. It is hard to adjust AP Tree in the real time update process which should be finished very quickly. However, rearranging AP Tree needs to compare relationships of several predicates which may cost beyond the time scale of milliseconds. To regain the optimized performance of AP Tree, AP Classifier reconstructs AP Tree with the new weights of atomic predicates periodically.

2.5 Experimental Evaluation

We have implemented and evaluated AP Classifier on a general purpose desktop computer with quadcore@3.2G and 16GB memory. Our implementation and evaluation include all functional components for packet behavior identification from scratch, including computing atomic predicates, classifying packets using the AP Tree, and computing packet behaviors. (In comparison, prior work on this problem only implements and evaluates a single function, namely: classifying packets to equivalence classes [59].) For our experimental evaluation, we use forwarding tables and ACLs from two real networks: Internet2 [14] and Stanford network [63]. As shown in Table 2.1, Internet2 includes 126,017 forwarding rules and the Stanford network includes 757,170 forwarding rules and 1,584 ACL rules. The predicates and atomic predicates are computed using the method in [118]. We compare AP Classifier with possible solutions by utilizing two state-of-art tools, namely Header Space Analysis (HSA) [63] and AP Verifier [118]. We do not compare AP Classifier with MDD [59] because it relies on a special method for MDD construction and the source code is not publicly available. Furthermore, its method does not support dynamic updates.

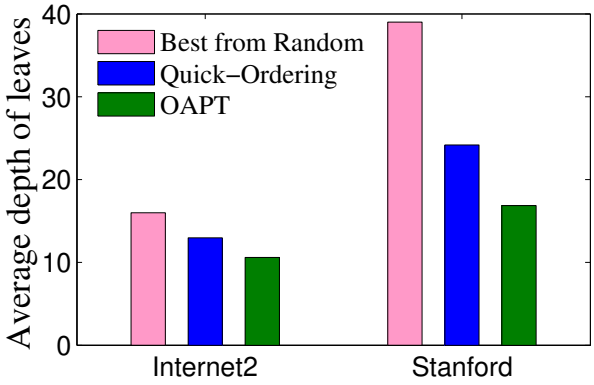
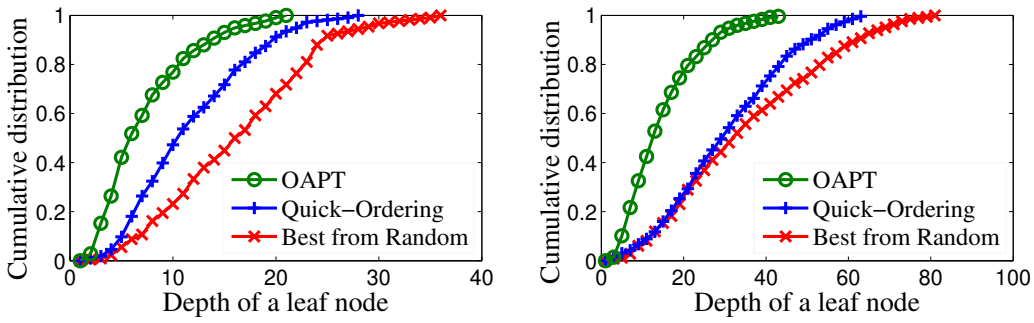


Figure 2.9: Average depth of leaves



(a) Internet2 **(b)** Stanford

Figure 2.10: Cumulative distribution of the depths of leaf nodes in AP Trees

2.5.1 Depths of leaf nodes

In this set of experiments, we show the depths of leaf nodes in an AP Tree, which can demonstrate effectiveness of the proposed tree construction algorithms. We evaluate and compare three methods, Best from Random, Quick-Ordering, and Optimized AP Tree construction (OAPT), for both Internet2 and Stanford networks. The Best from Random method generates a random order of predicates for placement on levels of an AP tree and performs pruning. It constructs 100 AP trees and chooses the tree with the minimal average depth of leaf nodes. Quick-Ordering is presented in Section 2.3.2 and OAPT is presented in Section 2.3.3.

Fig. 2.9 shows the average depth of of leaf nodes in an AP tree. For Internet2, the average depth of Best from Random is 16.0, worse than those of Quick-Ordering (13.0) and OAPT (10.6). OAPT reduces the average depth by 34% compared to Best from Random and 19% compared to Quick-Ordering. For the Stanford network, Best from Random also has the highest average depth (39.0), followed by Quick-Ordering (24.2) and OAPT (16.9). OAPT shows significant improvement: It reduces the average depth by 57% compared to Best from Random and by 30% compared to Quick-Ordering.

Fig. 2.10 shows the cumulative distribution of depths of leaf nodes in an AP Tree. For Internet2, the leaf depths of Quick-Ordering are clearly smaller than Best from

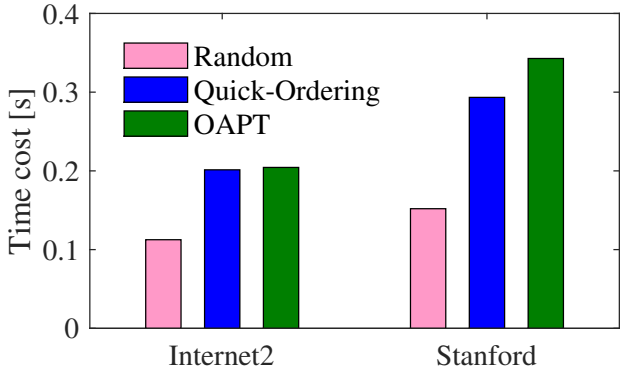


Figure 2.11: Overall construction time cost of AP Classifier

Random. However for the Stanford network such improvement is not very significant. OAPT has clearly smaller depths for all percentiles compared to the other two methods. For Internet2 80% of the leaf nodes in the OAPT tree have a depth less than 11 and for Stanford this number is 21. The maximum depths are 24 and 46 for Internet2 and Stanford, respectively.

2.5.2 Memory Usage

After construction, AP Classifier only stores one copy of all predicates and atomic predicates as BDDs and also, for each predicate, a set of integer identifiers of atomic predicates. In the AP Tree a node only stores a pointer to the labeled predicate or atomic predicate. Since pointers use very little memory, the memory costs of different methods are very close. Hence we only show the memory cost of AP Classifier using OAPT. In our implementation, we use JDD library [111] to construct BDDs and their logical operations. Each node in a BDD has a fixed size. The memory consumption of a BDD is determined by the number of nodes in the BDD. It is interesting to observe that more rules in a network do not always mean more BDD nodes. When there exist much more similarities among rules of a network, a BDD of the network is more likely

to be simple with a smaller number of nodes. The memory cost for the network is prone to be lower.

The total memory cost of AP Classifier for Internet2 is 4.79 MB and that for Stanford is 2.15 MB. Although Internet2 has fewer predicates than Stanford, it requires more memory because BDDs of the Internet2 predicates are more complex than those of Stanford. Unlike the results of [59] that only show memory cost of the search structure, our memory costs account for all components for packet behavior identification, including the network topology, predicates, atomic predicates, and AP Tree. We found that AP Classifier uses very small memory and can be stored in cache.

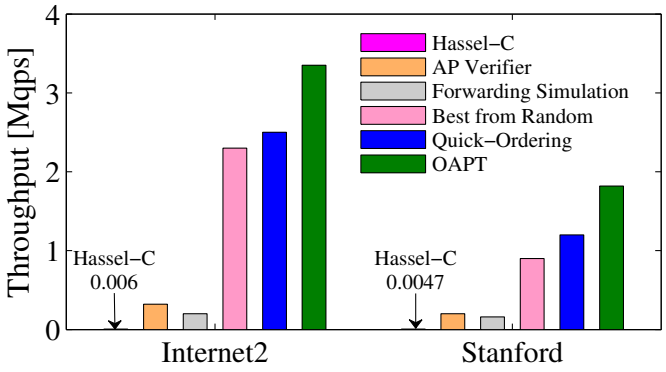
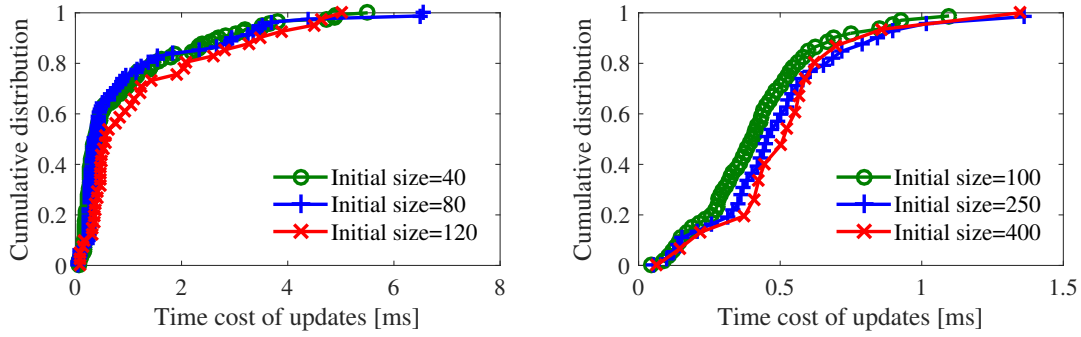


Figure 2.12: Query throughput for static networks



(a) Internet2 **(b)** Stanford

Figure 2.13: Cumulative distributions of time cost for adding a predicate.

2.5.3 AP Tree construction time

Fig. 2.11 shows times to construct AP Trees using the three methods for the two networks. Note that the time cost is the overall construction time that includes the times for computing atomic predicates as well as for AP Tree construction. The Random method costs the least time but it is only for one random construction. To find the best AP Tree from a large number of random constructions takes substantially longer time. Quick-Ordering and OAPT have similar time costs, 201.36 ms and 204.39 ms, for Internet2. For the Stanford network, OAPT requires 342.77 ms for Stanford, a little longer compared to Quick-Ordering (293.36 ms).

2.5.4 Query throughput for static networks

In this set of experiments, we measure the throughput of AP Classifier to process packet queries, in number of queries per second (qps). Packet headers used for queries in the experiments are generated randomly with respect to the atomic predicates. The throughput results for static networks are shown in Fig. 3.8. For Internet2, AP Classifier using OAPT can achieve 3.4 Mqps, higher than Best from Random by 102% and Quick-Ordering by 52%. For Stanford network, AP Classifier using OAPT can achieve 1.8 Mqps, higher than Best from Random by 46% and Quick-Ordering by 34%. For both networks, the throughput of AP Classifier is much higher than 1 Mqps, which is enough to satisfy most application requirements in SDN.

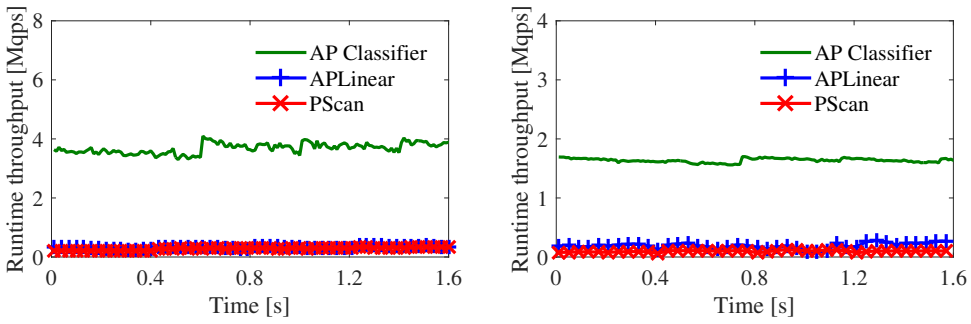
For static networks, we can use the open-source tool Hassel-C [11] that implements HSA [63] to perform packet behavior identification for a specific packet. By providing the input port and a specific query packet, Hassel-C computes the reachability tree of the query packet. (For a unicast packet, the reachability tree is a forward path to the packet's destination.) The query throughputs of using Hassel-C to perform packet behavior identification are 6 Kqps and 4.7 Kqps for Internet2 and Stanford, respectively,

which are about 1000 times slower than the query throughputs of AP Classifier. They are also plotted in Fig. 3.8 but they are very small and barely visible. We also compare AP Classifier with AP Verifier [118]. We first use AP Verifier to compute all atomic predicates, and perform a linear search of all atomic predicates for the query packet until the packet matches an atomic predicate. Results in Fig. 3.8 show that AP Verifier is also much slower, though its throughput is improved a lot compared to Hassel-C.

In addition we use a method of Forwarding Simulation, i.e., determining the forwarding behavior of the packet at a box, then checking the forwarding behavior on the next-hop box, until the packet stops. At each box, a packet is checked using the predicates at the box linearly until a match occurs. In our experiments using Forwarding Simulation, the average number of predicates checked is 96.8 and 232 for Internet2 and Stanford, respectively. The corresponding throughput is 0.2 Mqps and 0.16 Mqps as shown in Fig. 3.8. In contrast, only 10.6 and 16.8 predicates are needed to be checked on average using AP Classifier.

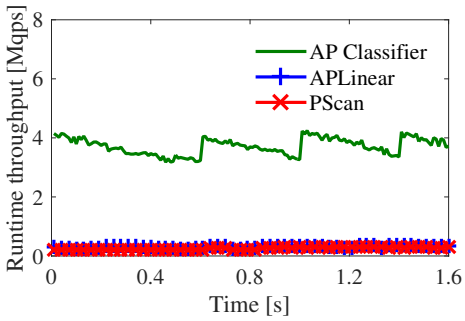
2.5.5 Dynamic Networks

In this set of experiments, we first construct the AP Tree using a number of predicates and then keep adding new predicates. We measure the time cost to add each new predicate and update the AP Tree. Fig. 2.13 (a) shows the cumulative distribution of time cost for adding a predicate in the Internet2 network. The initial number of predicates is set to 40, 80, and 120 for three different experiments. From the figure we find that about 80% of the predicate additions are finished in 2 ms. It may take 5-6 ms in worst cases. We do not observe obvious differences when the initial numbers of predicates are different. Fig. 2.13 (b) shows the results of similar experiments for Stanford. The initial number of predicates is set to 100, 250, and 400 for three different experiments. Over 90% of the predicate additions are finished in 1 ms. Deleting a predicate

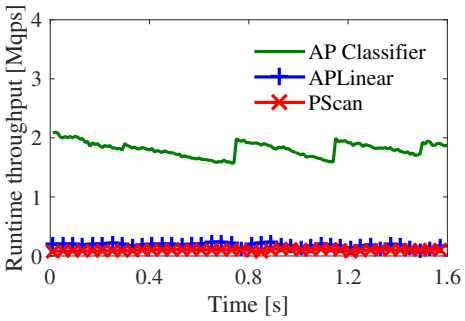


(a) Internet2, 100 updates/s

(b) Stanford, 100 updates/s



(c) Internet2, 200 updates/s

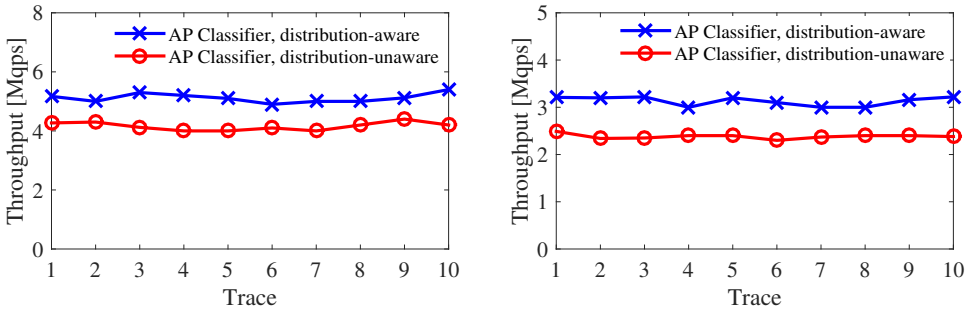


(d) Stanford, 200 updates/s

Figure 2.14: Query throughput for dynamic networks. The number of updates per second is 100 in (a) (b) and 200 in (c) (d)

does not require extra computation, hence there is no result for deletions.

Query throughput for dynamic networks. We also evaluate the throughput of AP Classifier in practical environments where additions and deletions of rules and predicates happen over time. At the beginning of each experiment, a number of predicates are chosen randomly from the set of predicates of a network to construct the initial AP Tree. Starting from time 0, the arrivals of change events requiring the addition or deletion of predicates are modeled by a Poisson process. Each update operation can be adding a new predicate or deleting an existing predicate. In all experiments, equal numbers of additions and deletions are inserted to the event queue. A reconstruction is triggered every 0.4 s. During every reconstruction, AP Classifier answers queries and performs updates as explained in Section 2.4.2. We compare AP Classifier with two possible methods, APLinear and PScan, APLinear utilizes AP Verifier [118] to com-



(a) Internet2 **(b)** Stanford

Figure 2.15: Query throughput of AP Classifier for different packet distributions

pute atomic predicates and performs a linear search for the query packet until the packet matches an atomic predicate. Note that BDDs of atomic predicates are more complex than those of predicates. Hence APLinear is not efficient. PScan performs a scan on all predicates using the query packet and decides whether the packet is filtered by the predicate. Both methods can be used to identify packet behaviors.

Fig. 2.14 shows the throughputs of AP Classifier, APLinear, and PScan in dynamic networks. The x-axis is time and the y-axis is throughput measured in Mqps. We conduct two sets of experiments whose update rates are 100 updates/s and 200 updates/s. From all subfigures in Fig. 2.14, we find that AP Classifier is faster than the other two methods by an order of magnitude. Note that starting from time 0, the throughput of AP Classifier slowly decreases as an increasing number of updates make the AP Tree less optimized. The first reconstruction starts at time 0.4 s and finishes at about 0.6 s in Fig. 2.14(a) and (c), and 0.7s in Fig. 2.14(b) and (d). When a reconstruction finishes, the throughput immediately goes back to a high value (4 Mqps in (a) and (c), and 2 Mqps in (b) and (d)). Furthermore, the throughput does not degrade in the long-term view. Comparing results of the two different update rates, we find that the average throughput of AP Classifier does not drop much even after the update rate is doubled. Hence AP Classifier is fast and robust for practical dynamic networks.

2.5.6 Impact of packet distribution

To evaluate the performance of AP Classifier under various packet distributions, we generate new sets of test traces which are unevenly distributed with respect to the atomic predicates. The number of packets corresponding to the atomic predicates are chosen by sampling from a Pareto distribution. The probability density function for the Pareto distribution can be expressed as:

$$f_X(x) = \begin{cases} \frac{\alpha {x_m}^\alpha}{x^{\alpha+1}} & x \geq x_m \\ 0 & x < x_m \end{cases} \tag{2.2}$$

Where x_m is the minimum possible value of X, and α is a positive parameter, which is known as the tail index. In our experiments, we chose $x_m = 1$, $\alpha = 1$. About half of atomic predicates have 1,000 packets, but some have more than 20,000 packets.

We generated 10 sets of traces for each network. If we still use the AP Trees constructed without the consideration of packet distributions (distribution-unaware), the average depth of all queries is 10.65 for Internet2 and 16.2 for Stanford network. Then we construct new distribution-aware AP Trees using the method described in Section 2.3.4. The average depth of all queries is reduced to 8.09 (Internet2) and 11.3 (Stanford). The corresponding values of throughput are shown in Fig. 2.15. We can see that, if AP Classifier measures the packet distribution and assigns different weights to atomic predicates, the throughputs in all cases have notable improvements compared to the distribution-unaware method. The average query throughput increases from 4.2 Mqps to 5.2 Mqps for Internet2 and from 2.4 Mqps to 3.2 Mqps for Stanford.

2.5.7 Dealing with packet header changes

In this set of experiments, we evaluate the throughput of computing packet behaviors when there exist middleboxes modifying packet headers. We use the topologies

of Internet2 and Stanford networks. In each experiment, one to three of switches are chosen as boxes connecting to middleboxes that may change packet headers. Due to lack of available middlebox policy data, we create ten entries for each flow tables of middleboxes. Match fields of flow tables are produced by dividing the packet header space into ten disjointed sets. We obtain match fields by grouping all atomic predicates into ten predicates. So every incoming packet can match an entry. When incoming packets match these entries, AP Classifier computes the remaining forwarding behaviors of packets using new atomic predicates. However for some packets, the new packet headers cannot be determined in advance. AP Classifier needs to search the AP Tree for the second time to find an atomic predicate for the new header. The process of computing packet behaviors ends until the packet is dropped or reaches the destination.

We measure the throughput of packet behavior computation under these circumstances. Packets used in the experiments are generated randomly with respect to atomic predicates.

Table 2.2: Throughput with packet header changes

	Throughput(Mpps)		
No. of middleboxes	One	Two	Three
Internet2	13	10.2	9.8
Stanford	10	8.6	7.4

(a) Deterministic ratio $= 0.9$.

	Throughput(Mpps)		
No. of middleboxes	One	Two	Three
Internet2	11.2	9.8	8.5
Stanford	8.9	7.9	7

(b) Deterministic ratio $= 0.5$.

	Throughput(Mpps)		
No. of middleboxes	One	Two	Three
Internet2	8.7	6.9	3.2
Stanford	7.1	4.9	2.1

(c) Deterministic ratio $= 0$.

Table. 2.2 illustrates throughput of computing packet behaviors for Internet2 and Stanford datasets in different scenarios. We define the deterministic ratio as the portion of middlebox rules that can determine the atomic predicates of packets after packet header changes. When the deterministic ratio is 0.9, the throughput does not downgrade much as number of middleboxes increases since most packets have new atomic predicates stored in the flow tables, as shown in Table. 2.2 (a). Compared with Table. 2.2 (a), the corresponding throughput values in Table. 2.2 (b) and (c) are lower since more packets passing through a middlebox require searching the AP Tree for a second time. In the worst case, the throughput of computing packet behaviors is still 3.2 M and 2.1 M packets per second respectively, which is much higher than using other methods.

2.6 Related Work

Network-wide packet behavior identification is equivalent to reachability computation for a specific packet. This problem is related to, but different from, network reachability analysis which has been studied for over a decade. Xie *et al.* [115] present a model for static reachability analysis of data plane network state. Quarnet [64] represents ACLs as firewall decision diagrams to compute network reachability. Header Space Analysis (HSA) [63] is custom-designed method to check network invariants but not in real time.

For real-time applications, NetPlumber [62] makes use of HSA to detect network invariant violations. Veriflow [65] stores all data plane rules in a multi-dimensional prefix tree (trie) and determines the Equivalence Classes (ECs) of packets. An EC is defined to be a set of packets that have identical forwarding actions in all boxes. Veriflow then checks network invariants by analyzing reachability graphs of ECs.

Binary Decision Diagram (BDD) [42] is an efficient structure that were used to model network properties. ConfigChecker [32] is general verification tool based on

symbolic model checking. It uses a BDD to represent a set of state transitions (also flowchecker [31] by the same first author). If n header bits are used for filtering, each BDD of ConfigChecker uses $2n$ state variables which is less efficient than BDDs used in our design and [118] (In our design and AP Verifier, each BDD represents a set of packets and requires the use of n bit variables only). Anteater [76] uses boolean formulas to represent policies for packets traveling over edges in a network graph. McGeer [80] models network verification as Boolean satisfiability problems. They both use a SAT solver to check network properties. All of these general-purposes tools are slow and operate on time scales of seconds to hours [65].

All of the above methods focus on analyzing network-wide invariants (e.g., reachability, loop-freedom) but were not designed to identify the reachability of a specific packet. For example, they can determine whether it is possible to reach box B from box A but cannot tell whether a given packet can reach B. AP Verifier [118] can check whether all packets entering a port in the network pass through a waypoint (e.g., a firewall) but cannot tell whether a specific packet traverses a given waypoint.

One possible solution to packet behavior identification problem is checking the packet against the set of atomic predicates calculated by AP Verifier linearly [118] which is impractically slow. Another solution is to obtain all related data plane rules of the packet by searching the trie created in Veriflow and then compute the forwarding path based on the rules. However storing all rules requires non-trivial memory cost (tens of GBs for the Stanford network) which could cause disk I/Os during query processing. As a result, using the Veriflow trie for packet behavior identification was shown to be very slow by Inoue *et al.* [59] who proposed a tool that can quickly classify a packet to an EC. Its main drawback is that their MDD structure cannot correctly represent the current network state because its does not support real-time updates, especially for SDNs where data plane updates are frequent [70]. Prefix DAG [99] employs

a data structure similar to MDDs, but it focused on a simple classification problem with a single header field.

Recently, Network Optimized Datalog is proposed as a general specification language to model high-level abstraction of network beliefs and dynamism [75]. A new approach to derive data plane from network configurations is in [49].

Chapter 3

SICS: Secure and Dynamic Middlebox Outsourcing

3.1 Overview

In modern networks, most middleboxes choose the appropriate processing actions based on headers of incoming packets. When a middlebox processes a packet, it finds a rule that matches the packet header and follows the action of the rule. Hence, rule information specifies the packet processing policies of the middleboxes. Both packet headers and rules contain private information belonging to the enterprise network. To facilitate middlebox outsourcing without compromising privacy, we design and implement SICS, a Secure In-Cloud Service function chaining framework.

3.1.1 The SICS Outsourcing Architecture

As shown in Fig. 3.1, SICS contains three parties: an enterprise (middlebox user), middlebox providers, and a third party cloud that holds in-cloud middlebox processing. The middlebox providers set up middleboxes per request. The enterprise configures

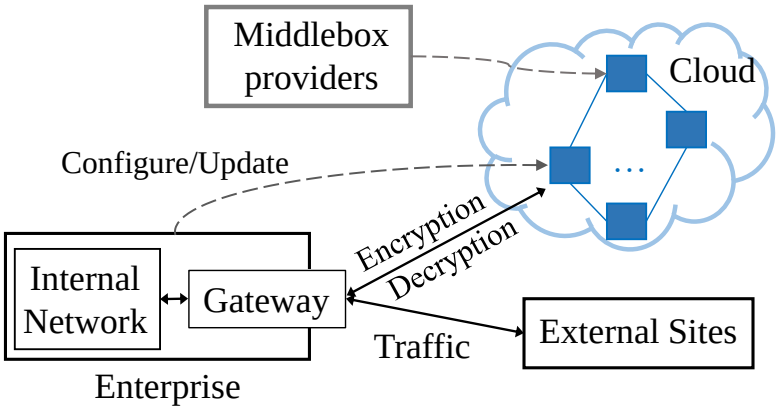


Figure 3.1: The architecture of SICS

and updates rules in these middleboxes. The enterprise has a gateway that connects the internal and the external network. All incoming packets to the enterprise will be forwarded to the gateway. The gateway encrypts the packet headers and payload and sends the packets to the cloud for middlebox processing. The encryption can use symmetric-key algorithms, such as the Advanced Encryption Standard (AES), which can be performed in near line speed for 10Gbps links [71]. The encryption key is only known by the enterprise. The in-cloud middleboxes process packets following the service function chains and then the cloud transmits the packets back to the enterprise. The gateway decrypts the packets and sends them to the internal network.

The key challenge in this architecture is how the in-cloud middleboxes correctly match packets to rules given that the packet headers are encrypted. To enable correct rule-matching, SICS assigns each packet a label. The label represents all behavior of the packet in the cloud, including to which middleboxes the packet should be forwarded and in which order, as well as which rules the packet should match at a middlebox.

The operations on outgoing packets from the enterprise to an external site are similar: before being transmitted over the Internet, outgoing packets are encrypted at a gateway, redirected to the cloud, and sent back to the gateway.

Note the SICS gateway does not encrypt the checksum or TTL and instead adds a

new checksum based on ciphertexts. Middleboxes can recompute checksums as usual.

An optimization that saves on bandwidth and latency can be adopted when communications are between two networks belonging to a same enterprise or two enterprises that have established a secure channel. After in-cloud processing, the traffic can directly go to the destination site without sending back since the same encryption key is shared by the two networks.

3.1.2 Security Model

In our security model, we assume the cloud and middlebox providers to be "honest but curious" [56]. They are honest to perform their services correctly. However, they might be curious to learn the user-configured processing policies at middleboxes or peek at the traffic received. This security model is practical and reflects the following real situations. First, the cloud or middlebox providers will not interrupt the normal cloud services because such an interruption will be detected [48] [121]. However, it is possible that the customer data might be gathered and sold by disgruntled employees [6] [22]. Additionally, hackers may try to steal the customer traffic and policy data [7]. [1]. SICS aims to protect the enterprise network privacy from all these attacks. We do not consider "active" attackers which manipulate costumers' traffic maliciously.

SICS provides two security properties of middlebox outsourcing: (1) For an encrypted packet, the cloud and middlebox providers should not be able to infer its packet headers based on its in-cloud behavior. (2) The cloud and middlebox providers should not be able to learn the plaintext of header spaces specified by the enterprise's processing rules.

In SICS, label assignment of packet headers does NOT need to be collision-resistant. Distinct packets can be assigned with the same label if they have identical behavior in the cloud. Distinct flows can still be differentiated based on their encrypted header

fields if needed.

3.1.3 Middlebox with Label Matching

Label matching (known as label switching in layer 3 routing) is a technique of network relaying that is much faster than traditional IP-header switching. Each packet is assigned a label and the switching takes place after examination of the label assigned to each packet. SICS applies label matching to middlebox outsourcing which provides two promising advantages: it can simultaneously achieve privacy protection and efficient packet processing.

Privacy protection of packet headers and rules. We name the service function chain and middlebox rule matching behavior of a packet as its cloud-wide behavior. A set of packets that have the same cloud-wide behavior form an policy equivalence class. In SICS, we assign the same label to all packets belonging to the same policy equivalence class, even if their packet headers are different. Given an encrypted packet with a label, SICS prevents an attacker from obtaining its original packet header. For example, h specifies a set of packet headers, and packets whose headers fall in h share the same cloud-wide behaviors. At the gateway, a packet is assigned a label (A label is represented as a binary string, e.g, "10110110", whose value has no relationship to the packet header) if its header belongs to h. The length of a label is determined by the total number of policy equivalence class. A label only includes two types of information: 1) which middlebox the packet should visit in the cloud, and 2) which action a middlebox should apply to this packet. The rule tables at the in-cloud middleboxes consist of label-matching entries as opposed to header-matching entries. In this way, neither the cloud nor middlebox providers can learn the original middlebox processing policies with respect to the packet headers.

Note that label-matching does not protect packet behavior, such as forwarding and

middlebox actions. These are known to the cloud no matter what type of protection is used.

Efficient table lookup. Label matching can achieve better performance compared to the traditional header based matching (e.g., IPv4 header), especially in software middlexboxes running on general-purpose servers: (1) A label corresponds to a policy equivalence class and may cover multiple header ranges, the number of entries in a label matching table could be much smaller than that in a header matching table. In our experiment, a rule set with approximately 100K header matching rules of a function network is converted to less than 250 labels. (2) With a properly designed hash table, label matching can achieve $O(1)$ lookup time, without the use of specialized hardware such as TCAM. 3) Label matching adds little per-packet bandwidth overhead. In our experiments, a 16-bit long label is sufficient to represent cloud-wide behavior in a network with nearly one million rules. The label can be placed in the options field in IPv4 protocol header.

While the use of label matching is not new in a general networking, our specific contributions lie in the design of header space mapping in the context of secure middlebox outsourcing.

3.1.4 Design Framework

Fig. 3.2 shows the system model of SICS. Those modules run on a controller in the enterprise network. At runtime, the enterprise network administrator decides middlebox processing rules and the service function chaining requirements based on the business objective of the enterprise. The rule preprocessing module takes these rules and specifications as input and converts them into label-based rules. A SICS gateway is constructed which assigns labels to packets based on the header space mapping relationship. To simplify in-cloud deployment, the controller then creates an abstract function

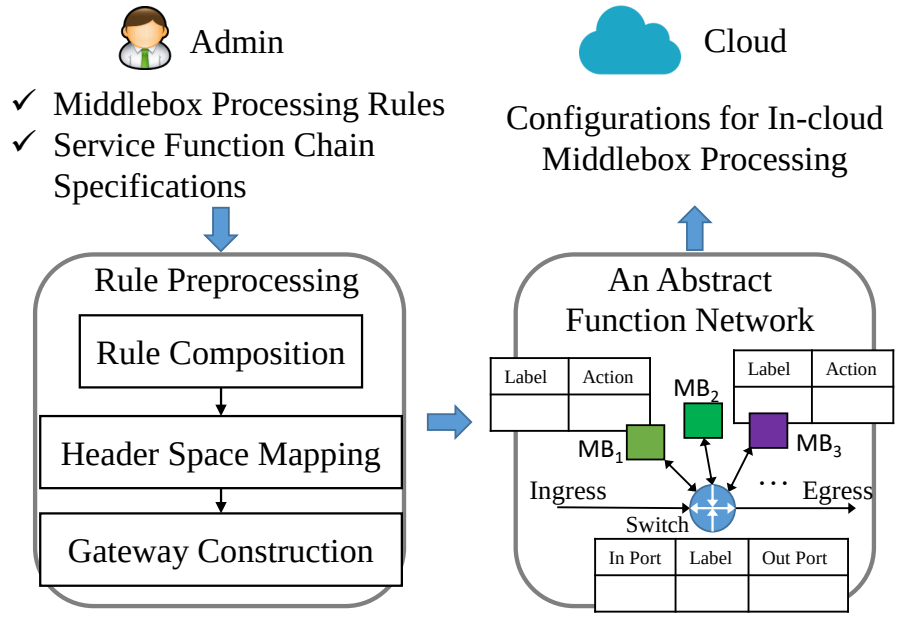


Figure 3.2: The system model of SICS

network which includes configurations for all middleboxes and an abstract switch that is connected to all middleboxes. For each middlebox, there is a rule table identifying the action applied to each packet based on the label. The abstract switch is equipped with a forwarding table. Besides label and output port entry, the forwarding table has an extra entry classifying packets based on their input ports. The input ports are used to identify the segment in the service function chain that the packet is currently in. The abstract switch determines the next hop of a packet based on its label and input port.

The abstract function network can be easily mapped to the configurations of a practical deployment in the cloud that ensures packets are processed by required middleboxes in a specified sequence. Configurations are sent to the cloud from the enterprise using a VPN tunnel. When there exist processing policy or rule changes, this procedure is called repeatedly to update both the enterprise gateway and the middleboxes running in the cloud.

3.2 Enterprise Modules of SICS

To enable secure middlebox outsourcing, SICS dynamically maps the header spaces specified by the middlebox processing policies to labels at the enterprise gateway. To keep the complexity low and maintain scalability, the gateway performs only inexpensive per-packet operations, which are parallelizable. In this section, we present the design of three key modules at the SICS enterprise side.

3.2.1 Rule Composition

The rule composition module takes the service function chain requirements and the middlebox processing rules as its input and implement its functionality in two steps.

It first combines different service function chain requirements and determines the overall service function chains for each set of packets. A service function chain requires that a class of packets must be processed by a number of middleboxes in a designated sequence. For example, all HTTP packets should go through IDS $\rightarrow$ Proxy. Packets from an internal site should be processed by NAT $\rightarrow$ Firewall. A service function chain is formulated with respect to a set of packets, specified by their packet headers, represented as a predicate P. P specifies the set of packets X for which $P(x)$ is $true$ for a packet $x \in X$. A packet may relate to multiple service function chain requirements and needs to be processed by all the middleboxes included in those chains. Consider m service function chain requirements: (P_i, c_i, r_i), for $i = 1, ..., m$. For the i-th requirement, let P_i be the predicate specifying the set of packets, c_i be the sequence of middleboxes, and r_i be the priority which is provided by administrators to determine the order of middlebox processing when two chains are combined. Requirements are listed in descending order of priorities. To ensure that packets are processed by all required middleboxes, SICS uses Algorithm 1 to calculate a set of middlebox chaining equivalence class, each of which specifies a set of packets with an identical service

Algorithm 1: Compute equivalence classes for middlebox chaining requirements.

Input : Predicates of service function chain requirements (P_i for $i = 1, ..., m$).

Output: A list of predicates $\mathcal{F} = \{f_1, f_2, ...f_n\}$.

```
1  T1 = ∅, T2 = ∅, T1.add(P1), T1.add(¬P1)
2  for i = 2 to m do
3  |   for each f ∈ T1 do
4  |   |   if f ∧ Pi ≠ false then
5  |   |   |   T2.add(f ∧ Pi)
6  |   |   end
7  |   |   if f ∧ ¬Pi ≠false then
8  |   |   |   T2.add(f ∧ ¬Pi)
9  |   |   end
10 |   end
11 |   T1 = T2, T2 = ∅
12 end
13 F = T1
14 Return F
```

chain.

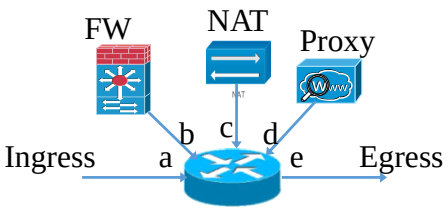
The output of Algorithm 1 is a list of predicate $\mathcal{F} = \{f_1, f_2, ...f_n\}$. The conjunction of any two predicates in $\mathcal{F}$ is *false* (referring to an empty set). Therefore packet sets specified by any two predicates have no intersection. Each predicate f_i corresponds to a service function chain, which can be obtained by concatenating c_i of P_i, if the conjunction of f_i and P_i is not false. The order is determined by their corresponding priorities.

Based on the composed service chain requirements, the rule composition module generates the forwarding table at the abstract switch to steer traffic along the required middleboxes in a sequence. Based on the input port field, we can partition the forwarding table into sub-tables. In each sub-table, we calculate one predicate for each output port by combining corresponding packet header prefixes or ranges. In our implementation, by representing packet sets as predicates, the merge operation can be performed

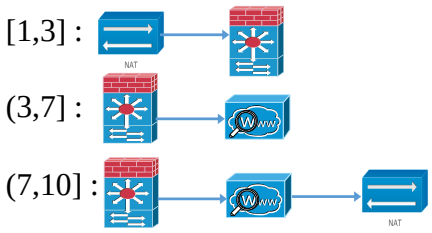
efficiently using graph-based algorithms with Binary Decision Diagrams (BDDs) [42]. With predicate compositions, there exists at most one predicate per output port in each sub-table. We use the example shown in Fig. 3.3 to illustrate this process. Fig. 3.3(a) is an abstract function network with three middleboxes. All middleboxes are connected by an abstract switch with five ports. Port b,c and d are used to link the middleboxes and port a and e are ingress and egress ports. Fig. 3.3(b) shows three sample service function chains. The set of packets in each chain is specified by an integer range. [1] Fig. 3.3(c) is the original forwarding table at the virtual switch that steers traffic across the middleboxes according to the service chains in Fig. 3.3(b). From Fig. 3.3(c), we see that many items in each sub-table share the same output port. This allows us to reduce the size of each table by merging ranges which have the same output port. The resulting forwarding table is shown in Fig. 3.3(d). We reduce the total items in the forwarding table from 14 to 9.

The second step of the rule composition module is combining user-configured middleboxes processing rules which are created locally either by the network administrator or middlebox providers. We define the middlebox rules with the 3-tuple: (P_i, b_i, r_i), where P_i denotes the predicate from the i-th rule, b_i is the action performed on packets matching this rule and r_i is the priority. We sort all rules at a middlebox in descending order with respect to priorities. When a packet is checked against the rules at a middlebox, it is matched by the first rule whose predicate evaluates to true. We use Algorithm 2 to convert the rules of a middlebox to a list of predicates $\mathcal{F} = \{f_1, f_2, ...f_n\}$, each of which specifies the packets sharing the same behavior at the middlebox, where n is the total number of distinct behavior. For example, a firewall may have a predicate specifying packets allowed by the ACLs and another predicate specifying the ones denied.

[1] In our implementation, all packet sets are converted to predicates and represented by binary decision diagrams (BDDs) [42]. Here we use integer ranges for simplicity.



(a)



(b)

Input a

Header space	Output port
[1,3]	c
(3,7]	b
(7,10]	b
other	e

Input b

Header space	Output Port
[1,3]	e
(3,7]	d
(7,10]	d
other	e

Input c

Header space	Output Port
[1,3]	b
-	-
(7,10]	e
other	e

Input d

Header space	Output port
-	-
(3,7]	e
(7,10]	c
other	e

(c)

Header space	Output port
[1,3]	c
(3,10]	b
other	e

Header space	Output Port
(3,10]	d
other	e
-	-

Header space	Output Port
[1,3]	b
other	e
-	-

Header space	Output port
(7,10]	c
other	e
-	-

(d)

Figure 3.3: (a) An abstract function network. (b) Service function chain requirements. (c) Original forwarding table. (d) Merged forwarding table.

Algorithm 2: Compute a predicate for each action.

Input : Sorted processing rules at a middlebox (P_i for $i = 1, ..., m$)
Output: A list of predicates $\mathcal{F} = \{f_1, f_2, ...f_n\}$

```
for j = 1 to n do
    f_j ← false
end
valid ← false
for i = 1 to m do
    if P_i shares the same action as f_j then
        f_j ← f_j ∨ (P_i ∧ ¬valid)
        valid ← valid ∨ P_i
    end
end
Return F
```

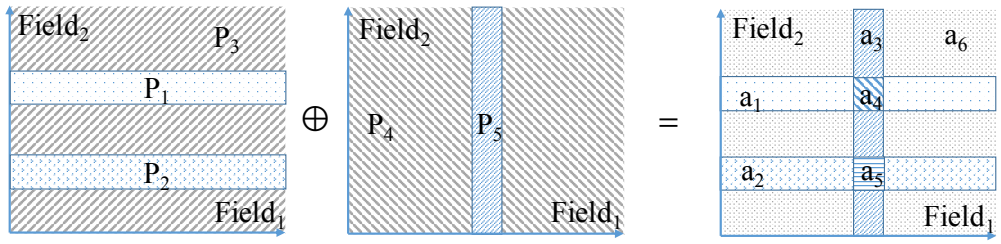


Figure 3.4: Header space divided by predicates

3.2.2 Header Space Mapping

After rule composition is performed, we obtain a list of predicates for each middlebox and the abstract switch. Predicates from a box can be seen as a partition which divides the packet header space into several disjoint sub-spaces, each with the same action. If we place predicates from all of the boxes together, the partition of the header space will become combinatorically finer due to the intersection of predicates from different boxes.

Fig. 3.4 shows an example illustrating the process of placing predicates from two boxes into a single header space. Each predicate is associated with two header fields[2].

[2]In practice, a predicate may be defined over multiple fields, e.g., 5-tuple in TCP/IP packets. Here, we use two dimension headers as an example.

Five predicates $P_1 \sim P_5$ from the two boxes are placed together in one packet header space. Then, the header space is partitioned into 15 blocks. Each block represents a set of headers belonging to the same set of predicates. The packet headers within one block will match the same set of predicates and exhibit identical behavior at all boxes. Therefore, they have the same cloud-wide behavior and hence belong to the same policy equivalence class. Note that a policy equivalence class is not necessarily a single block. Blocks that are specified by the same set of predicates belong to the same equivalence class. As shown in Fig. 3.4, the original predicate P_1 is divided into three segments. The right and left segments are only covered by P_1 and form an equivalence class a_1. The segment in the middle is covered by both P_1 and P_5 and forms an equivalence class a_4. In total, the partition of 15 blocks forms 6 equivalence classes represented by $a_1 \sim a_6$.

To obtain the policy equivalence classes, SICS reuses Algorithm 1 given a list of predicates. At this time, the input is the set of predicates from all middleboxes and the abstract switch. The set of policy equivalence classes has two key properties: (1) Packets within the same class have identical cloud-wide behavior. That is, these packets will traverse the same sequence of middleboxes and have same behaviors at each middlebox in the network. (2) Each input predicate is equal to the disjunction of a subset of policy equivalence classes, shown in Fig. 3.4 where $P_1 = a_1 \vee a_4$ and $P_5 = a_3 \vee a_4 \vee a_5$.

SICS maps packet headers within an policy equivalence class to one label. In the rule tables of the in-cloud boxes, predicate P is represented as *a set of labels*, which are determined by the subset of policy equivalence classes whose disjunction is P.

3.2.3 Example

We show an example abstract function network configured with labels in Fig. 3.5. The abstract switch is divided into four separate switch instances with each connecting

to a single middlebox. We have two flows h_1 and h_2. Flow h_1 is required to go through a firewall, a NAT and a load balancer, while flow h_2 should go through a proxy. For simplicity, we assume the sets of predicates for all middleboxes and switches in Fig. 3.5 have a similar partition of the packet header space as in Fig. 3.4. For example, switch S_1 has two predicates that specify the same partition as P_4 and P_5. P_5 specifies the set of packets that are forwarded to the firewall and other packets specified by P_4 are forwarded to S_3. The NAT has three predicates which specify the same partition as $P_1 \sim P_3$. Packets matching P_1 are translated to packets specified by P_2. P_3 represents a default drop predicate. The set of policy equivalence classes are still $a_1 \sim a_6$ as in Fig. 3.4. h_1 and h_2 belong to the packet sets specified by a_4 and a_1, respectively. Relevant entries for the two flows are shown in the label-matching tables of middleboxes. The two forwarding tables are for switch S_1 and S_4. From the figure, we can see packets in h_1 (red arrows) will be forwarded to and allowed by the firewall. After that, the label is changed to a_5 and then a_2 by the NAT and the load balancer sequentially based on label replacement actions. Details on label replacement are presented in §3.3.2. Finally the packets are forwarded to the egress by S_4 with the label a_2. Similarly, packets in h_2 (greed dotted arrows) are processed by the proxy before they are sent back to the gateway. Note the input port field at a switch is necessary when incoming and outgoing packets share the same label.

3.2.4 Packet Classification

To assign labels to packets, the gateway determines to which policy equivalence class a given packet belongs. Policy equivalence classes can be represented as the conjunction of input predicates. An intuitive approach is to test the packet against these predicates linearly. However, this approach is obliviously too slow.

SICS uses all predicates obtained from the rule composition module to build a

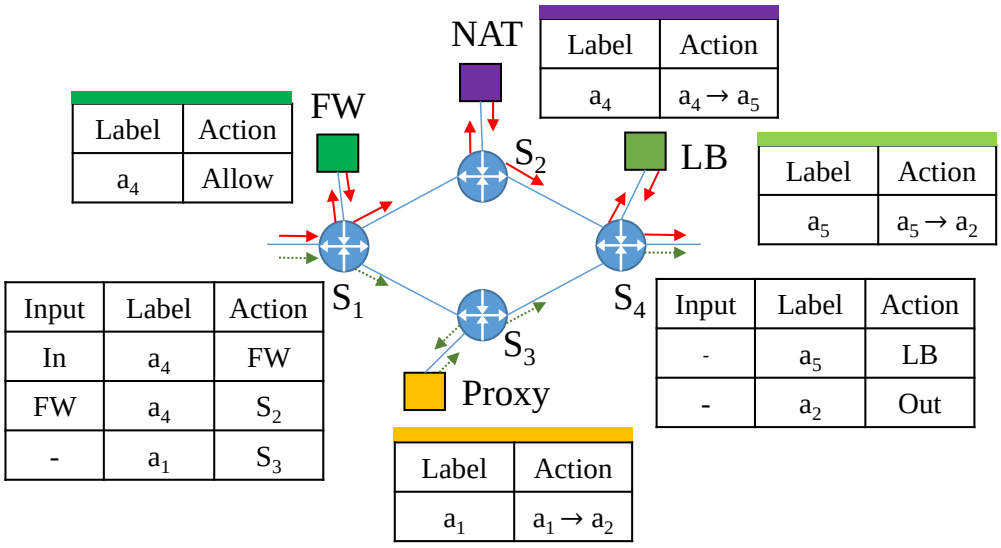


Figure 3.5: An example Abstract Function Network

packet classifier, using the algorithms in [112]. The proposed classifier includes a binary tree whose root has a predicate p_1. At level i, the 2^i internal nodes each has a predicate p_i. Starting from the root, at each internal node, the input packet header is evaluated by the predicate of the node. If the result is true, the packet continues to be evaluated in the left sub-tree. Otherwise, it goes to the right sub-tree. A leaf node represents an policy equivalence class and the set of packets that can reach this leaf belong to the policy equivalence class. In practice, for a tree constructed by k predicates, its height is considerably lower than k and the number of leaves is significantly smaller than 2^k. The reason behind this observation is that conjunctions of a large number of predicates are are *false* and specify empty sets of packets, no new node will be created. More importantly, using the methods in [112], the classifier supports incremental updates when there exist policy changes. For example, new predicates can be added at the bottom of the tree with little overhead.

The gateway classifies packets into one of the policy equivalence classes, with each has a unique cloud-wide behavior. This corresponds to the provable coarsest refinement of packet header space and thus can be used to provide best computation time and space

performance of the gateway.

3.3 In-Cloud Modules of SICS

SICS supports the core functionality of a wide range of middleboxes. For those middleboxes that examine packet headers (e.g., firewall, NAT, L3/L4 load balancer), SICS can be directly applied using the label-matching method. For middleboxes that also check payload (e.g., web proxy, IDSes), SICS can be combined with recent works of secure DPI [104, 122].

Note that for very simple middleboxes, such as a stateless firewall blocking certain IPs, the gateway can fulfill its task when computing the label, packets that only traverse these middleboxes are processed locally and do not need to be redirected to the middleboxes running in the cloud. However, we observe many middleboxes involve expensive operations and for this reason enterprises tend to outsource them.

3.3.1 Stateful Middlebox

Unlike switches or routers, common middleboxes conduct stateful functionalities (e.g., bidirectional firewall and address translation [89, 95] , stateful load balancing [10,45]) and use advanced statistical techniques to detect and prevent potential security threats (e.g., flood protection [20,50]). Packet behavior at middleboxes may depend on the history of packets the middleboxes have observed. Such functions can be resource-consuming since they need to maintain a separate state for every single connection. For example, a stateful firewall will permit an inbound packet filtered by the ACLs if it belongs to an established connection. Such functions reply on per-connection states, in-cloud middleboxes should be able to recognize packets of the same connection based on encrypted packet headers.

In SICS, all header fields are encrypted as a whole to provide high security guarantee and thus cannot be used to identify packets of the same connection. To support per-connection states, SICS adds a 32-bit connection identifier to each packet based on a pseudorandom function [55] , seeded in a given seed s:

$$I_c = prf_s((IP_{src} \| port_{src}) * (IP_{dst} \| port_{dst}))$$

Using the equation above, the inbound and outbound packets of the same connection will have the same identifier. By conducting experiments using a real dataset [12], we observe that the probability that two packets from different connections having the same identifier is negligible. Note adding an identifier to recognize packets of the same connection is a general approach that can be applied to other middlebox outsourcing work, such as Embark [71] and Splitbox [33].

Algorithm 3: Compute equivalence classes after adding header transformers.

Input : A list of predicates $\mathcal{P}$ and a set of packet transformers $\mathcal{T}$
Output: A list of predicates $\mathcal{F} = \{f_1, f_2 ... f_n\}$

$\mathcal{F} \leftarrow \mathcal{EC}(\mathcal{P}), \mathcal{P} \leftarrow \mathcal{F}$
for $T \in \mathcal{T}$ *and* $f_i \in \mathcal{F}$ *that can be transformed by* T **do**
  $\mathcal{P} \leftarrow \mathcal{P} \bigcup T(f_i)$
end
$\mathcal{F} \leftarrow \mathcal{EC}(\mathcal{P}), \mathcal{P} \leftarrow \mathcal{F}$
for *each deterministic* $T \in \mathcal{T}$ *and* $f_i \in \mathcal{F}$ **do**
  Compute the set $\mathcal{B} = \{b_1, b_2, ...b_l\} \subseteq \mathcal{F}$ whose disjunction is $T(f_i)$
  $\mathcal{R} \leftarrow \{T^{-1}(b_j) |$ for each $b_j \in \mathcal{B}\}$
  $\mathcal{P} \leftarrow \mathcal{P} \bigcup \mathcal{R}$
end
$\mathcal{F} \leftarrow \mathcal{EC}(\mathcal{P})$
Return $\mathcal{F}$

3.3.2 Header Transformer

In SICS, a single label is sufficient to guide all rule matching behavior of a packet if it does not traverse middleboxes that modify packet headers. As shown in Fig. 3.5, header transformers such as NAT, load balancer may modify packet headers. When a packet goes through a header transformer, the behavior of the packet at downstream boxes is determined by its new header. With label-matching, the subsequent packet behavior must be determined by the new label corresponding to the new header. Hence, middleboxes must be able to assign new labels to packets they have just modified without ever learning their headers.

To address the above problem, we design a label-to-label replacement scheme. A packet transformer maps an input packet set to an output packet set. For a packet transformer T and a predicate P specifying its input packet set, $T(P)$ denotes the transformed predicate specifying the output packet set. More specifically, given a predicate P, $T(P)$ can be calculated by replacing constraints on corresponding header bits. For example, a transformer for a four-bit prefix $11\star\star$ modifies the second bit from 1 to 0. This operation can be modeled by applying existential quantification and conjunction of the new constraints to the second bit. The transformed predicate represents prefix $10\star\star$. Similarly, T^{-1} can be calculated using the inverse process. SICS supports both deterministic (e.g., one to one mapping from a prefix to another) or non-deterministic (e.g., randomly choose a new address from a given prefix) packet transformers.

Header transformers may produce new policy equivalence classes. Given a list of predicates $\mathcal{P}$, we extend Algorithm 1 to calculate the new set of policy equivalence classes, denoted as $\mathcal{EC}(\mathcal{P})$, when header transformers exist. As shown in Algorithm 3, a new set of policy equivalence classes is calculated after a set of transformed predicates are added (line 5). For a transformer T, the transformed predicate $T(f_i)$ for a policy equivalence class f_i is equal to the disjunction of a subset of equivalence classes

$\mathcal{B} = \{b_1, b_2, ...b_l\}$. If T is non-deterministic, a packet in the packet set specified by f_i is randomly transformed into a packet that belongs to either one of equivalence classes within $\mathcal{B}$. However, if T performs a one-to-one mapping, a transformed packet must belong to a deterministic policy equivalence class. To decide into which equivalence class a packet should be transformed, lines 6-11 of Algorithm 3 calculate the inverse predicate for each $b_i \in \mathcal{B}$ and update the set of equivalence classes. Then, each deterministic transformer has a one-to-one mapping for all policy equivalence classes. With the refined set of policy equivalence classes, SICS can easily build a label replacement table for each header transformer. Upon receiving a packet with a label that can be processed by the transformer, a non-deterministic header transformer randomly modifies the label to one of the multiple labels, whereas a deterministic header transformer always conducts a unique label replacement action. Example label replacement tables are shown in Fig. 3.5 for a NAT and a load balancer.

In addition to replacing labels, the middlebox also assigns an index corresponding to the modified header, e.g., an index for an IP in a prefix stored at the gateway. When the gateway receives a packet with such an index, it restores the modified header fields.

To keep the connection identity, a header transformer maintains a mapping from the newly assigned header/index to the original connection identifier. For reverse packets, the gateway does not encrypt assigned header fields (e.g., random port number ranges assigned by a NAT). Upon receiving packets with the same assigned header fields, the transformer restores the connection identifier. So the same processing policy is applied in subsequent middleboxes.

3.3.3 Case Studies

Next, we use a proxy and a Palo Alto firewall [20] as examples to discuss how SICS combines the two techniques above to support more complex real-world middleboxes.

Proxy. An HTTP proxy accepts a TCP connection from a client, extracts the URI, and looks it up in its cache. This results in one of two cases: hit or miss. (a) Hit: The proxy extracts the encrypted header of the packet and creates a new reply packet with the header and the requested contents which it then sends to the client. The proxy also adds a label to each reply packet which directs the packet for subsequent processing. When the gateway receives the HTTP reply packet, it decrypts the packet header and restores the source and destination addresses of the packet. (b) Miss. The proxy creates a new HTTP connection and forwards the same encrypted request to the Web server. The proxy also adds its own encrypted address and a label for further processing. When the packet bounces back to the gateway, the gateway decrypts the packet header and replaces the source address with the proxy's address. In the reverse direction, reply packets from the Web server are encrypted and received by the proxy. The proxy caches the replied content and sends the content back, as in case (a). During this process, packets are forwarded and processed by the proxy in the cloud without exposing the headers.

Palo Alto firewall. Palo Alto firewall is a commercial network gateway which performs firewall, NAT, and/or IDS functions organized in a chain. Here, we consider a firewall-NAT chain that examines packets headers. The NAT function can be divided into two categories: source NAT and destination NAT. A source NAT translates the headers of connections initiated within internal networks, while a destination NAT applies to connections started from outside networks.

For a packet initiated within the inside network, the firewall first applies its label-based ACLs and stores the connection identifier if the packet is allowed. Then, the NAT adds an index for a reserved external IP, a random port number and assigns a new label to the packet based on the label replacement table. Note header transformers may break the connection identity between outbound and inbound packets. To make

the connection reversible, the NAT maintains a mapping from the newly assigned port number to the packet's original encrypted headers as well as the connection identifier. Before packets are sent out to external networks, the gateway decrypts and restores the header fields assigned by the source NAT. For a reverse packet, if the destination port belongs to the range of random port numbers assigned by the source NAT, the gateway encrypts the packet and places the port number in the options field of the packet. Using the port number, the NAT restores reverse packets with the corresponding original encrypted headers and the connection identifier. So the same processing policy is applied to reverse packets at the firewall. Packets initiated from outside networks have similar processing schemes, except that a destination NAT maintains a deterministic one-to-one mapping from a public address to a private address.

3.4 Update operations

Overload is a common cause of middlebox failures [54]. Traffic should be steered across different middlbox instances dynamically. Service function chain requirements and middlebox processing rules are also changing constantly to meet the new costumers' needs or reduce security threats. All changes in traffic processing result in rule updates at the enterprise and on the cloud sides. To keep the correctness and performance of in-cloud processing, it is necessary for a middlebox outsourcing framework to support incremental rule updates with low latencies. A rule insertion or deletion can be converted to predicate changes [118]. If there are predicate changes after the rule updates, SICS performs the following methods to update both the enterprise side and the in-cloud boxes.

Update at the enterprise side. SICS starts by updating the packet classifier at the gateway. When a new predicate is added, SICS adds the new predicate to the bottom of the packet classifier. If the update produces new equivalence classes, the packet

classifier starts to classify packets to the new set of equivalence classes. When existing predicates are deleted, SICS updates the set of equivalence classes by merging the equivalence classes if they identify the same cloud-wide behavior. Updates to the classifier can be executed very fast. In our experiments, the average cost of adding/deleting a predicate is less than 0.5 ms.

To figure out the update schemes of in-cloud boxes, the enterprise controller maintains a representation list for each predicate. This list includes all equivalence classes whose disjunction is equal to the predicate. In the example shown in Fig. 3.4, the representation list of P_5 is $\{a_3, a_4, a_5\}$ and for P_2 it is $\{a_2, a_5\}$. Representation lists of predicates are maintained dynamically, so when the list of a predicate is modified, the controller sends update instructions to the in-cloud box which produces the predicate.

Update in the Cloud. In SICS, a rule update in the cloud consists of the updating of the rule tables (hash tables) at each middlebox and the abstract switch. The forwarding table of the abstract switch is partitioned into several sub-tables which are updated independently. When a new equivalence class is added into the representation list of a predicate, its label-action pair is inserted into the rule table of the in-cloud box that produced the predicate. Here, the key is the label which corresponds to the policy equivalence class and the value is the action of the predicate. In contrast, a label-action pair is removed from the rule table when the corresponding equivalence class is deleted from the representation list of the predicate.

The connection states maintained in the stateful middleboxes will not be disrupted during an update since states are identified by encrypted packet headers or connection IDs.

Maintaining Processing Consistency. Rule updates need to be treated carefully. Any inconsistency in state between the gateway and the boxes in the cloud may lead to incorrect middlebox processing. To maintain per-packet consistency, the controller first

calculates the incremental rule update schemes for the enterprise gateway and boxes involved in the cloud. During this time, the gateway and in-cloud middleboxes continue to encrypt and process traffic according to the old rules. Once the update schemes are determined, the gateway buffers incoming packets until all in-cloud packets finish processing in the cloud (The buffering time is bounded by the packet processing time, which is typically hundreds of milliseconds [102]). Then, the gateway and in-cloud boxes install updates and start processing new packets. To maintain flow consistency, ongoing flows should continue traversing the original sequence of middleboxes while they are updating. SICS employs the migration avoidance mechanism in [86]. New flows are steered to new middlebox instances while existing flows are still processed by old ones.

3.5 Security Analysis

SICS converts IP prefixes and other header spaces from middlebox processing rules to a list of predicates. Each predicate is represented as a set of labels that are used as matching fields to enable in-cloud functionalities. Labels do not leak size, order or borders of header spaces specified in the rules. The cloud is unable to learn to which field of the packet header a match corresponds. Labels at in-cloud middleboxes are updated independently and the information about header spaces represented by these labels cannot be inferred from updates. A gateway encrypts packet headers and assigns a label to each packet in order to identify its in-cloud processing. In this case, given an encrypted packet with a label, its original packet header cannot be reversed from the label. For any two packets that are assigned the same label, the cloud is limited to learning that the two packets have the same cloud-wide behavior, but prevented from determining any other information about their orders or values.

Information leakage. From an information-theoretic point of view, information

leakage of a communication system is at least $log_2 N$ bits, where N is the number of observable equivalence classes [77]. In the context of SICS, each equivalence class identifies a cloud-wide behavior, which is represented by one label. The label instructs the in-cloud boxes to process the packet as configured. With a less number of cloud-wide behaviors, the cloud may not be able to correctly perform its functionalities. In this sense, SICS achieves minimal information leakage. On the other hand, Embark employs a field-by-field encoding to convey the information about how packets should be processed in the cloud. The set of cloud-wide equivalence classes are the Cartesian product of per-field equivalence classes. Consequently, Embark exposes a larger number of observable equivalence classes and hence more information leakage.

Next, we demonstrate that the security of SICS is stronger than the PrefixMatch in Embark [71] under two attacks.

Chosen Plaintext Attack. A chosen plaintext attack allows an attacker to determine which plaintext message is encrypted into an input ciphertext message. We assume that an attacker (e.g., the cloud itself or a hacker) selectively sends sample packets to the gateway and observes their cloud-wide behavior, attempting to figure out the plaintext of the rules at a middlebox. PrefixMatch adopts a per-field encryption scheme where prefixes or ranges for each header field are encrypted separately. For an encrypted prefix or range, the attacker knows to which field of the packet header the prefix or range corresponds. The plaintext of the encrypted prefix or range can then be obtained by traversing the entire search space of that field.

An example of such attack is the following: for the destination port field in the IPv4 header, PrefixMatch encrypts a port number interval $[s, e]$ to a random interval $[S, E]$. All port numbers falling in $[s, e]$ are encrypted to values in $[S, E]$. Knowing the interval $[S, E]$, it takes an attacker at most 2^{16} queries (e.g., sample packets with a destination port traversing from 0 to 2^{16}) to find all port numbers in $[s, e]$, where 16 is the length

of the port field. Now the attacker has successfully deciphered the encrypted interval $[S, E]$ in the cloud. In addition, when a future packet matches the interval $[S, E]$, the attacker learns that the original destination port of the packet falls in $[s, e]$. Similarly, the attacker could learn mapping relationships for other fields. Since a chosen packet header can test each header field simultaneously, the number of required queries to decipher all header fields is determined by the length of the longest header field. For a 5-tuple, the longest header field is 32 bits. So it takes at most 2^{32} queries to decipher a 5-tuple based ruleset which is encrypted using PrefixMatch.

As described in §3.2.2, SICS encrypts packet header fields as a whole. This means all packet header fields are involved in the header space mapping process, i.e., the label of a packet is determined by all of the bits in its header. When considering the same attack just described, we clearly see the benefit of SICS which require 2^{104} queries to decipher, a significant improvement over PrefixMatch's 2^{32}. PrefixMatch cannot be modified to encrypt all fields as a whole since the encryption in PrefixMatch is based on comparing per-field values of packets and the endpoints of rules.

Frequency Analysis Attack. Frequency analysis is a classic inference attack that has been historically used to recover plaintexts from substitution-based ciphertexts, and is known to be useful for breaking deterministic encryption. In frequency analysis, an adversary acquires knowledge of the frequency distribution of plaintext messages (e.g., via unintended data release or data breaches), counts the frequency of ciphertext messages and maps each ciphertext to the plaintext in the same frequency rank. To conduct frequency analysis, we assume the cloud is able to obtain the plaintext enterprise traffic from a previous time period and tries to infer the current encrypted traffic using the previous frequency distribution. To prevent frequency analysis, SICS adds randomness to the encryption of the original packet headers and the connection identifiers by changing the seed for symmetric key generation and the pseudorandom function after a certain

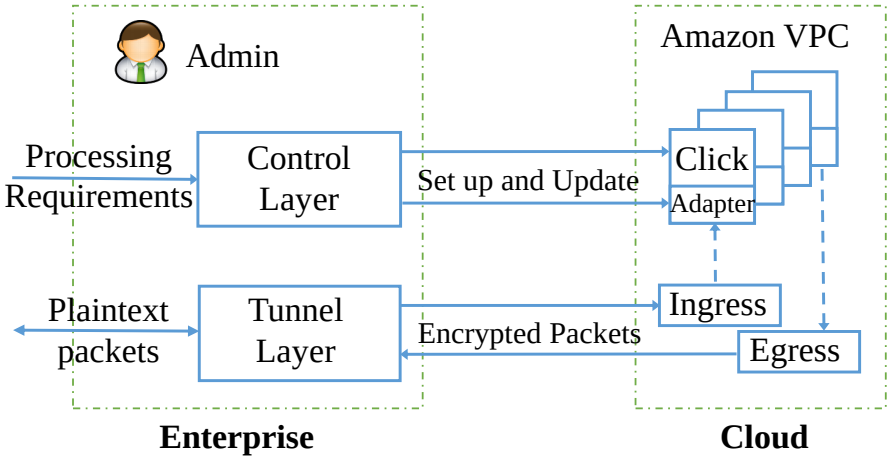


Figure 3.6: SICS software architecture

time period. In SICS, it is not useful to add randomness to the labels of packets. For example, if a new label is assigned to a packet when the behavior of the packet does not change, the cloud can easily determine the new label is equivalent to the old label because they specify the same cloud-wide behavior. However, frequency analysis only achieves low inference accuracy in SICS. One reason is that because a label in SICS covers a range of packet headers, the cloud cannot infer the frequency of each single packet header using the frequency of the label. Another reason is that the frequency analysis is sensitive to label updates that occur during middlebox load balancing and the changes in processing policy over time. An update to a label can change the frequency rank of multiple labels, including the label itself as well as other labels with similar frequencies. In contrast, PrefixMatch uses a one-to-one deterministic header mapping which is less secure in terms of frequency analysis.

3.6 Implementation

We have built a SICS prototype in our laboratory using middleboxes running in the Amazon Virtual Private Cloud (VPC) [2] and a gateway running on a general purpose

desktop computer with quadore@3.2G and 6GB memory. The gateway redirects traffic from another machine using the same model.

Fig. 3.6 shows the software architecture of SICS. The enterprise side consists of two layers: a control layer and a tunnel layer. The control layer takes the service function chain requirements and processing rules of the middleboxes as its input to calculate an abstract function network. When there are changes, the control layer updates the packet classifier in the tunnel layer and calculates the necessary updates in the cloud. Then, it sends batched update instructions to the middlebox instances running in the cloud. The tunnel layer, acting as a gateway, performs packet manipulation, header encryption and VPN tunnels connecting remote instances in the cloud.

On the cloud side, the abstract function network can be easily converted into a practical deployment within the Amazon VPC. SICS supports all header-related middleboxes. We implemented middleboxes using Click [68] and rule tables using the Cuckoo hash table [46, 85]. To enable in-cloud middlebox chaining, SICS adds an adapter layer which holds a sub-forwarding table from the abstract switch at each middlebox instance. Based on their labels, the adapter decapsulates incoming packets for current processing and encapsulates outgoing packets with the address of the next middlebox.

A possible limitation of SICS is that SICS employs label matching which requires modifications to the existing header matching based middlebox implementations. However, we believe this issue will be fully relieved with the emergence of new and promising modularized network function frameworks such as OpenBox [41].

3.7 Evaluation

We now investigate the performance of SICS at both the enterprise side and in-cloud middleboxes.

3.7.1 Enterprise-side performance

Gateway

We first evaluate the performance of the SICS gateway. For most experiments, we use a synthetic workload generated by the Pktgen traffic generator powered by Intel's DPDK [21]. We create an abstract function network using Stanford dataset [12] with three types of middleboxes: firewalls, source NATs and destination NATs. A destination NAT is used to implement a L4 load balancer. The Stanford dataset has 16 routers (2 backbone routers connected to 14 zone routers) with 757170 IPv4 forwarding rules and 1584 ACL rules. Firewalls can be placed on any router. For each firewall, we randomly select ACLs from the ruleset and shuffle the order to achieve different security policies. NATs are added to the dataset connecting zone routers to private subnets. For each NAT added, we use a different public IP address for the newly created port of the zone routers and a different private prefix for the subnet. A subset of forwarding rules are used to steer traffic along middlebox chains. We vary the number of middleboxes from 0 to 16 with the total number of rules increasing from 100K to 800K to show how the performance of SICS is affected by the network size. We compare the SICS gateway with PrefixMatch in Embark [71] since PrefixMatch is the only existing *cryptographic* approach that supports service function chaining. We report the median of 10 iterations for each experiment.

Construction time. Table 3.1 shows the construction time of the gateway with respect to the network size. For SICS, rule composition accounts for the most of the overhead while computing equivalence classes and constructing the packet classifier can be finished in tens of milliseconds. In Embark, the time cost is the time to construct the data structure for PrefixMatch. The PrefixMatch structure in Embark works only on one header field, so PrefixMatch needs to be run for every header field, one after another. In Table 3.1, we see that the time cost of PrefixMatch in Embark is at least

No. of Rules (K)	Rule Composition (s)	Computing ECs (ms)	Packet Classifier (ms)	Embark (s)
100	0.3	14.9	53.4	7.2
200	1.1	15.2	83.2	12.6
400	2.9	22.4	129.0	18.8
600	7.1	25.2	148.2	50.3
800	9.4	30.5	249.8	76.43

Table 3.1: Construction time of the gateway.

5 times larger than SICS for all six network sizes. The reason is that the total number of sub-intervals for each header field in PrefixMatch is much larger than the number of policy equivalence classes in SICS. For example, the test network with 100K rules produces approximately 200 equivalence classes; whereas the number of sub-intervals calculated using PrefixMatch is over 9000. This highlights the efficiency of the SICS approach compared with the process used by PrefixMatch when it finds the intervals pertaining to the same set of prefixes, especially when the size of the network is large. As shown in Table 3.1, the construction of the gateway in SICS only uses 368.3ms for the network with 100K rules and it is still less than 10s when the size of the network increases to 800K.

Incremental rule update cost. In this set of experiments, we first construct the packet classifier using a subset of predicates and then keep adding new or deleting existing predicates. In Fig. 3.7, we measure the time cost to update each predicate. We find that the medium time cost for updating a predicate does not have a distinct difference when the network size increases. The medium time cost for updating a predicate is less than 0.5 ms for all networks.

PrefixMatch in Embark may need to be reconstructed when a rule changes and the reconstruction process costs nearly 100s. PrefixMatch can still process packets using old configurations during the reconstruction; however, the long update delay may incur packet losses and harm the accuracy of middlebox processing. The situation worsens

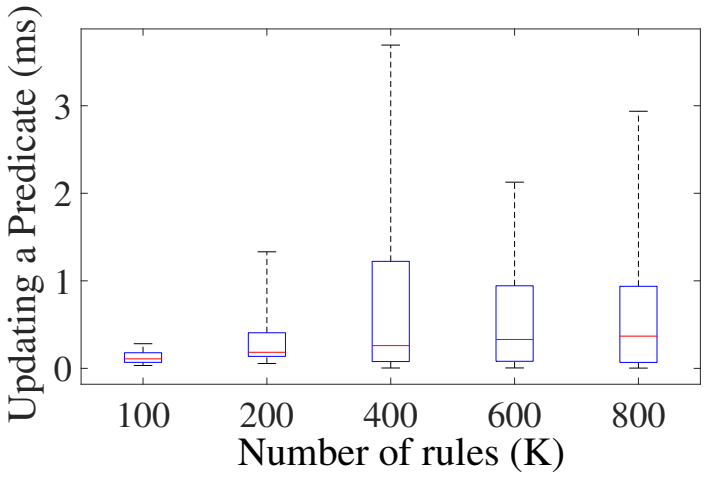


Figure 3.7: Box plot of update cost.

when updates happen frequently.

Throughput. We measure the throughput of the gateway in SICS by the number of queries per second (qps). Packets used in the experiments are generated uniformly with respect to equivalence classes and results for various network sizes are shown in Fig. 3.8. From the figure, we find that the gateway in SICS can achieve 3.92 Mpps for the network with 100K rules. For the largest network with 800K rules, the throughput is 2.2 Mpps. For all networks, the throughput of the gateway in SICS is higher than Embark by approximately 20%.

Memory usage. The gateway of SICS only stores predicates, calculated by the rule composition module, instead of rules. Predicates are represented as BDDs in our implementation. For each predicate, the controller maintains a representation list recording a subset of equivalence classes and their corresponding labels whose disjunction is equal to the predicate. Each equivalence class is represented as a set of pointers to predicates which contain the equivalence class. With Embark, the memory cost of the data structure for PrefixMatch is also calculated. For all network sizes, the gateway of SICS uses less memory than Embark. The memory cost is 0.267MB for SICS and 0.274MB for

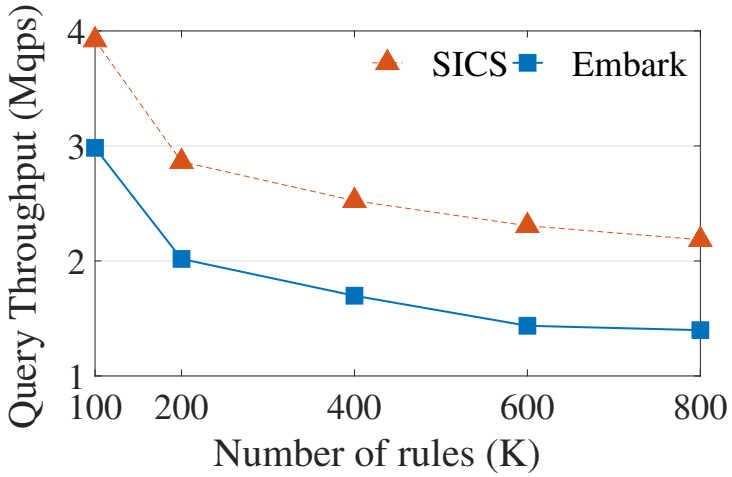


Figure 3.8: Throughput as the number of rules increases.

Embark when the network size is 100K. For the largest network with 800K rules, SICS and Embark uses 0.349MB and 1.345MB respectively. Neither the gateway in SICS nor Embark consumes appreciable memory since they only store the classifier and not the rules.

Scalability of the gateway. As shown in previous results, the performance of Embark degrades sharply as the total number of rules increases. Compared with Embark, the performance of SICS mainly depends on the number of equivalence classes calculated from these rules, which is a much smaller value than the number of rules. Given processing rules and service chaining requirements, the number of equivalence classes is determined by the number of various possible actions at the middleboxes and the service function chains, not by the total number of rules. For example, a firewall with 10K ACL rules produces only two equivalence classes, with each one corresponding to the action deny and allow, respectively.

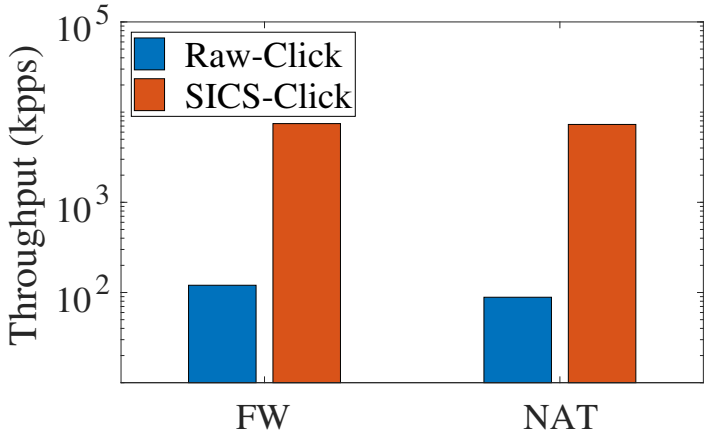


Figure 3.9: Lookup throughput of Middleboxes.

Bandwidth Overhead

We evaluate the extra bandwidth overhead between the enterprise and the cloud. Embark introduces a 20-byte overhead per IPv4 packet because it converts them to IPv6. SICS only inserts a 16-bit label into the options field of IPv4 packets which encodes up to 65536 equivalence classes (cloud-wide behavior). For middleboxes that modify packet headers, SICS uses another 16 bits as the identifier to represent rewritten header fields. For stateful middelboxes, SICS adds a 32-bit connection ID. Hence, the total per-packet bandwidth overhead introduced by SICS is 64 bits or 8 bytes. This is placed in the options field of IPv4 protocol header.

Processing Delay

SICS employs a similar middlebox outsourcing architecture as Embark which involves encryption and redirection overhead. Compared with local processing, deploying SICS in the Amazon VPC incurs hundreds of milliseconds processing delay; whereas an ISP based deployment with a larger footprint with respect to the Amazon VPC can reduce the delay to tens of milliseconds [71].

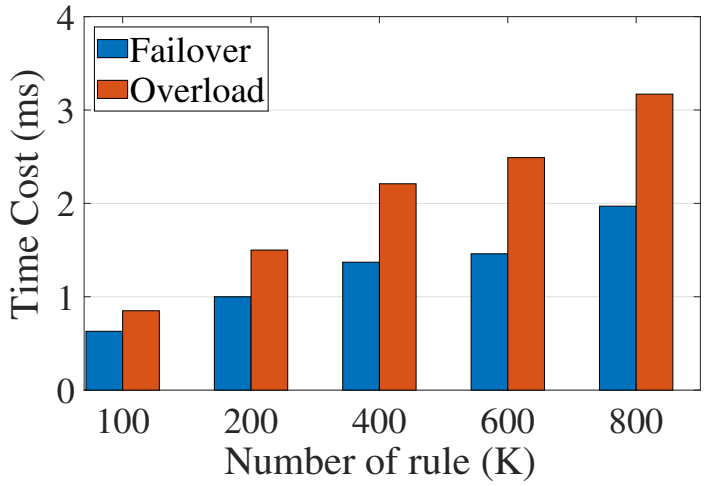


Figure 3.10: Response time in the case of a middlebox failure and traffic overload.

3.7.2 In-cloud Middleboxes

In this section, we evaluate the performance of label-matching based in-cloud middleboxes. We develop middleboxes using Click [68] and lookup tables using (2,4)-Cuckoo hash tables [46], with each uses 64 KB memory.

Throughput of in-cloud middleboxes. For comparison, we also implement a Click firewall and a Click NAT that examine packet header based rule tables. Each middlebox has 1000 IPv4 5-tuple rules. Fig. 3.9 shows the throughput in thousand of packets per second (kpps, log scale) for the two middleboxes. We see that the throughput of label-matching based firewall and NAT in SICS is about 8000 kpps, which shows an improvement of two orders of magnitude over their header based pattern matching counterparts.

Reacting to middlebox failures and overload. We consider two dynamic scenarios: (1) a middlebox fails and (2) traffic overload at a middlebox. We measure the reaction time of SICS for each scenario and the results are shown in Fig. 3.10. When a middlebox fails, we need to migrate the state of the failed middlebox to a new instance and configure the network to reroute packets with certain labels to the new instance.

To prevent traffic overload at a middlebox, in addition to middlebox state migration, we need to add new predicates to split a portion of traffic on the current middlebox to another middlebox. This requires additional updating of the packet classifier at the gateway and representation lists at the controller. From Fig. 3.10, we see that the overall time to react to middlebox failure and traffic overload is low (several milliseconds) and in fact the overhead is negligible.

3.8 Related Work

APLOMB [102] and Jingling [53] are the pioneer of middlebox outsourcing. APLOMB demonstrates that the latency inflation due to outsourcing is negligible. As a parallel work to APLOMB, Jingling focuses on the interfaces and inter-operations between the cloud and customers. Neither of them takes privacy issues into consideration. Blindbox [104] enables equality based operations on encrypted payload of packets for a specific class of middleboxes, DPI; However, it cannot examine packet headers and/or perform range queries. Melis et al. [82] model the behavior of common middleboxes and proposed a privacy preserving middlebox outsourcing scheme based on fully homomorphic encryption [40], which has very poor performance. Embark [71] presents the method PrefixMatch to hide the packet header and rule information from the cloud. PrefixMatch uses the set of processing rules to divide each header field into multiple intervals and then it assigns a random IPv6 prefix to each interval. At a local gateway, every header field of a packet is mapped to a pseudorandom value of an IPv6 field separately and the entire IP packet header is mapped to a new IPv6 header. PrefixMatch does not support incremental rule updates and updating one rule requires all rules to be reconstructed, which may take as long as 100s. Before that, packets are still routed and processed as the old configuration which may incur unexpected packet loss and inaccurate processing. From a security perspective, such a field-by-field encryption scheme

is vulnerable to certain types of attack, such as chosen plaintext attack. More details will be analyzed in §3.5. Splitbox [33] distributes a rule to several virtual machines (VMs), which reside on multiple clouds assuming an adversary cannot corrupt all VMs simultaneously. Computation results from all VMs are collected by a local middlebox and the final actions of the packets are calculated at the local middlebox. It is difficult for Splitbox to support service function chaining. Meanwhile, Splitbox increases bandwidth overhead several-fold as it needs to send multiple copies of a packet to different VMs for the same network function.

SafeBricks [90] and Shieldbox [109] are state of the art *enclave-based* middlebox outsourcing solutions. Besides the potential security threats from *curious* middlebox providers and side channel attacks, they impact performance by around 15% across different in-cloud middleboxes due to the use of SGX enclaves [90] and do not support incremental update. Changing of service chains and provisions (e.g., number of deployed middlebox instances) requires to rebuild the whole enclave which takes a few minutes.

Chapter 4

Epinoia: Intent Checker for Stateful Networks

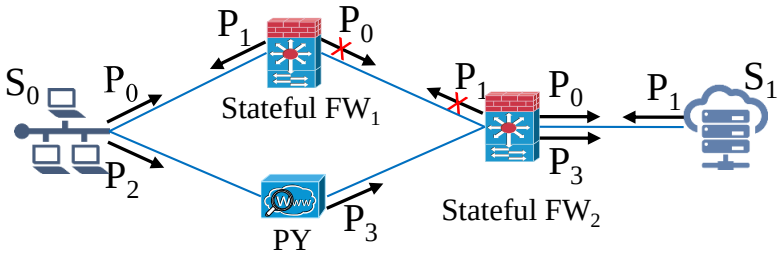
4.1 Epinoia Design and Architecture

We first describe several motivating examples to illustrate some challenges for checking network intents in stateful networks. Then, we provide insights behind the design choices and an overview of our system, Epinoia.

Consider the network pictured in Figure 4.1 with an end host subnet S_0 and a server subnet S_1. FW_1 and FW_2 are two stateful firewalls and PY is a forward proxy that works as an intermediate agent between clients and servers. The operators intend to block S_0 from S_1. That is, hosts in S_0 should not be able to send any packet to servers in S_1. The bottom of Figure 4.1 shows configuration snippets that implement this intent. Line 1 is a security rule at FW_1 that denies all packets from S_0 to S_1. However, as FW_1 conducts stateful processing, those packets may still be allowed if they belong to established connections initiated from S_1. To prevent such connections, a similar deny rule for packets from S_1 to S_0 (line 3) is added at FW_2. Even with this simple example, checking intent using existing tools could give inaccurate results and be time-consuming.

Static vs. temporal modeling. Recent work on network control plane configuration (e.g., BGP configuration) synthesis [51] and verification [35] have shown routing messages between routers can be effectively modeled using static boolean variables. Following this idea, as in Figure 4.1, the property that a packet P_0 from S_0 is able to reach S_1 through FW_1 and FW_2 can be represented using a Boolean variable r_0. As P_0 is denied by the security rule at FW_1, for r_0 to be $True$, it implies that a earlier reverse packet P_1 was relayed by FW_1 from S_1 to S_0, denoted as r_1, where $r_0 \Rightarrow r_1$. For P_1 to reach FW_1, it must first be allowed by FW_2. Likewise, due to the deny rule at FW_2, it requires a reverse packet P_0 to go through FW_2 from S_0 to S_1, or denoted as $r_1 \Rightarrow r_0$. Given the stateful network and the configurations, analyses solely based on such static modeling techniques report a violation of the block intent when both r_0 and r_1 are assigned to $True$. However, this turns out to be a false alarm. FW_1 will allow P_0 only if it saw P_1 before, which requires that P_1 went through FW_2 earlier. Thus, it cannot rely on the state created by P_0. This example shows the necessity of utilizing temporal modeling instead of static modeling for stateful networks as packets may have different behavior at stateful NFs when they arrive in different sequences.

Partial vs. complete path set. To scale with modern solvers, several optimization techniques have been studied in solver-based approaches [35, 88]. The core idea is to reduce the size of constraints given to the solver by restricting packet headers and their forwarding paths based on destination addresses. That is, the checking is conducted over a slice of the network (e.g., a single forwarding path). Though such simplifications could reduce the time cost, they may also lose completeness and lead to unsound checking results, especially when there are NFs that modify packet headers. One such example is shown in Figure 4.1. Line 6 of the configuration snippets indicates that PY in the bottom will explicitly intercept request packets from S_0 to S_1 and forward them with a new source S_2 (line 7). Those packets are also allowed at FW_2 (line 5).



//---------- Configurations ----------

Security policy on FW_1
1 service ANY address S_0 S_1 deny
2 service ANY address S_1 S_0 allow

Security policy on FW_2
3 service ANY address S_1 S_0 deny
4 service ANY address S_0 S_1 allow
5 service ANY address S_2 S_1 allow

Proxy policy on PY
6 web-proxy explicit enable address S_0 S_1
7 outgoing-ip S_2

Figure 4.1: Example NF configuration snippets.

Instead of sending packets directly to S_1, a host in S_0 could first send packets to PY, which then forwards the packets to S_1. This indicates a potential violation of the block intent between S_0 and S_1. In addition, networks are built with fault tolerance. Critical services are multi-homed, and communication endpoints have redundant paths. The dynamic nature of the underlying routing plane may assign different paths at different time even for the same set of packets. The NF processing taken depends on the path a packet actually traverses. *Configurations of NFs must ensure that no potential path violates network intents.*

Host vs. group level querying. In existing intent-based systems, all intents are specified with respect to end point groups (e.g., engineering department, a group of servers) [4, 30, 93]. Recall the previous intent: S_0 should not be able to reach S_1. Consider S_0 as a guest network with 100 hosts and S_1 to be a data center with 1000 servers. To check the block intent, the naive approach of exploding the query into

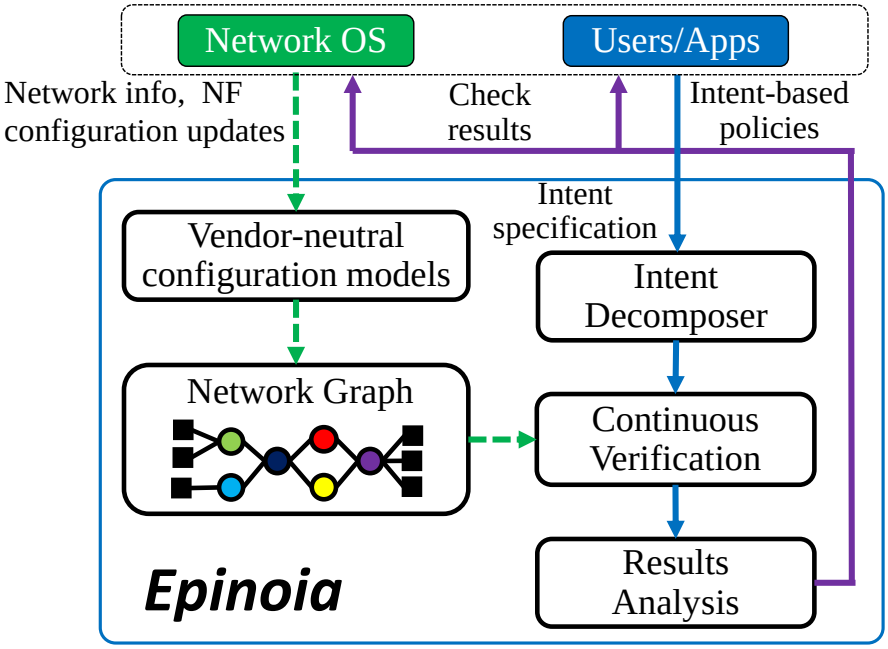


Figure 4.2: Epinoia workflow

100 thousand separate queries is too slow. A typical effective solution would convert the original query and check whether its negation can be satisfied. However, due to the stateful processing of NFs, this technique cannot be applied for stateful networks. More details are discussed in Section 4.3.1. We observe that NF processing policy commonly partitions end hosts into policy equivalence groups, i.e., into set of end hosts, to which the same policy applies. In Epinoia, endpoints relating to the same set of intents are represented as groups and queries for the same group are aggregated to achieve better efficiency.

Epinoia Overview. Figure 4.2 illustrates an overview of the Epinoia workflow with its key components. Epinoia allows users/applications to specify network intents based on extended policy graph models (Section 4.2.1).

NFs from different vendors may support different configurations and features. We break down the functionalities of advanced NFs into function units and propose vendor-neutral configuration models for each function unit (Section 4.2.2). Such function units

can be combined and extended to support real-world NFs. To correlate configurations of NFs and packet behavior in stateful networks, we formulate key causal precedence relationships [97] among NF packet I/Os and states (Section 4.2.2). All constraints are attached to a network graph, containing all potential paths needed to be checked for each intent to ensure that NF configurations match intents under arbitrary routing dynamics.

Along each path, an end to end intent is decomposed into sub checking tasks (Section 4.3). Each smaller task can be efficiently checked using a SMT solver. The continuous verification module maintains a causality graph with all checked results (Section 4.4). The goal is to enable the intent checker to check for network-wide intent violations incrementally whenever there are changes to network and/or intent. Finally, checking results are analyzed and all reported violations are returned to the network OS or intent creators.

4.2 Intent and Network Models

4.2.1 Network Intent Specification

Network intents specify the desired outcome of the network. In this dissertation, we look at two very basic intents: reachability and isolation, which can be used as building blocks to implement other advanced intents.

Epinoia extends the intent specifications in PGA [93]. Our choice is motivated by the intuitive graph representation of network intents, support of NF chaining. Figure 4.3 shows four example network intents in an enterprise network. Nodes are pre-defined end point groups and directed edges indicate the communication intents between endpoints. Boxes along edges specify the required NF traversal for each communication. In addition to the required ones, packets are allowed to go through other NFs by de-

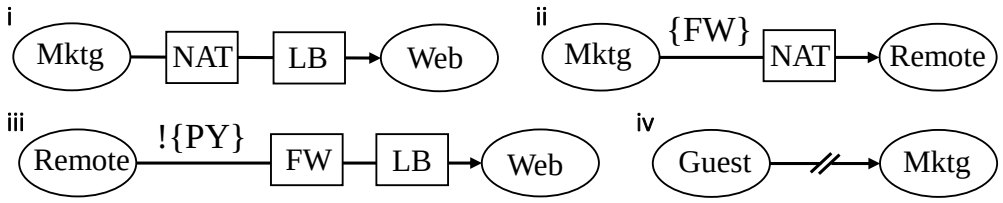


Figure 4.3: Example network intents

fault. Constraints on possible optional NFs are annotated on each edge segment in the form of $\{NF_1...NF_n\}$. Similarly, avoidance of NFs are specified using the form $!\{NF_1...NF_n\}$. For an isolation intent, a double slash is added on the edge to indicate that the communication must be blocked. The four intents in Figure 4.3 are: i) Marketing department should be able to access web services and the traffic must go through a NAT and a load balancer. ii) The department should also be able to access remote sites by going through a NAT and possibly one or more firewalls before the NAT. iii) Packets from remote sites to web services must be inspected by a firewall and a load balancer. No proxy is allowed before they are inspected by any firewall. iv) Packets from guest networks to the marketing department must be blocked.

4.2.2 Network Models

NF Configuration Models

Recent work on NF modeling has shown that NFs of the same type from different vendors have similar operating logic [88, 106, 114]. For example, the firewall function of `iptables` [94], pfSense [100] as well as Palo Alto Firewall [87] all start with detecting whether a packet belongs or relates to an established connection. Then the packet is matched against a list of ACLs. If one is found and it allows the packet, then the packet is forwarded; otherwise it is dropped. Contrary to the similarity in the operating logic, we observe that NFs differ greatly in the format or features they support in their configurations, which are the major input that operators provide and want

to check before they are installed into NFs. To mitigate the complexity brought by vendor specificity, open source communities such as OpenConfig [84] as well as some emerging IBN platforms in industry (e.g., Apstra AOS [4] and Google Zero Touch Network [69]) have been working on designing vendor-neutral configuration models. However, most of those models are for routers or routing related protocols and none include NFs. Another observation is that advanced NFs usually consist of a chain of basic functions. For example, a Palo Alto Firewall can be configured to implement a firewall-NAT-Load balancer chain. Inspired by the observations above, in Epinoia, we propose vendor-agnostic configuration models for common function units (e.g., address objects, security rules, NAT rules) which are written as extensions of the OpenConfig YANG models. Models for each function unit can be combined to form the configuration model of more advanced NFs. Moreover, with off-the-shelf tools, configurations written using the model can be easily converted into serialization formats (e.g., JSON) for transmission or other third-party services (e.g., network intent verification). Listing 4.1 shows an example configuration instance of a real security rule in JSON. The model is extensible to support additional features based on the actual functionalities of NFs.

```
...     "security rules": {
         "security rule": {
           "23": {
             "id": "23",
             "config": {
               "src address": "guest",
               "dst address": "marketing",
               "service": "ANY",
...            "action": "DENY",
```

Listing 4.1: Snippets of a security rule in JSON

Network Graph

To obtain the complete path set that should be checked for intents, Epinoia models a network as an undirected graph. Nodes in the graph are either endpoints or NFs while edges represent possible packet exchanges between those nodes. Such a graph can be extracted by traversing the network topology: if the current node is a switch or has been visited, continue to examine the next node; otherwise, create a new node in the network graph representing the corresponding NF or endpoints. Note that Epinoia does not aim to check the correctness of stateless switching fabrics as there already exist plenty of solutions [62, 63, 65, 118]. Meanwhile, by removing the switching fabric, the network graphs result in much smaller sizes (degrade the size by at least 50% [103]) but are still able to capture all potential paths.

Figure 4.4 shows a network graph of a typical enterprise network. Internal endpoints m_1 and g_1 belong to the marketing and guest networks, connected to remote sites with two firewalls. A web service is hosted in a demilitarized zone, guarded by a destination NAT and a load balancer. Epinoia leverages an off-line path generation step to obtain all simple paths. For most scenarios, the set of paths is fairly static and can be precomputed (e.g., regular hardware maintenance), we expect this step to be performed infrequently.

Encoding NF packet processing

The functionality of a NF can be factored into two generic parts: i) a classifier that searches for a matching over packet header fields or payload, and ii) a transfer function that transforms incoming and outgoing packets. Upon receiving a packet, based on configurations, a NF processes the packet with the actions corresponding to the rules or states that the packet matches. Naturally, the input packet on which an output depends must be received before the output is produced. In other words, there exists a causal

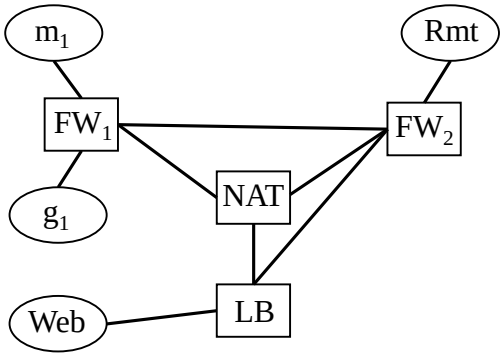


Figure 4.4: A network graph

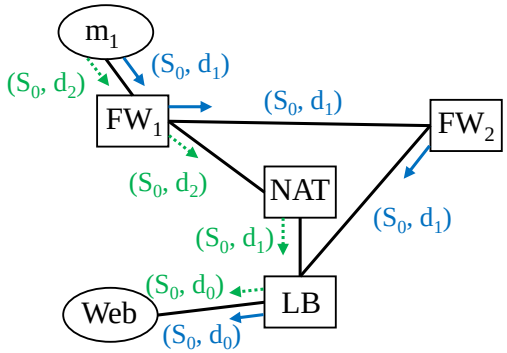


Figure 4.5: Path segmentation from marketing to Web

precedence relationship [97] between the input and output. We can generically express this relationship as $send_{p_2} \Rightarrow recv_{p_1}$, where $[A] \Rightarrow [B]$ denotes event A depends on B. p_1 and p_2 correspond to the same packet before and after NF processing. Both p_1 and p_2 are subject to certain constraints determined by NF configurations, e.g., for a firewall, p_1 and p_2 are exactly the same since firewalls do not modify packets; p_1 must match an established connection or be allowed by security rules. States at NFs correspond to packet histories. For example, if a content c is cached at a proxy, the proxy must have received a request packet for c and a response packet from the server that holds c before it can be cached at the proxy. Written generically: $state_s \Rightarrow recv_P$, where P represents a sequence of packets required to establish state s. Such causality also exist between one NF's output and another NF's input. For example, a packet must be sent out before it is received. Written generically: $recv_p \Rightarrow send_p$.

A rich set of causalities exists in NFs, e.g., a timeout must be reached before a state expires; a configuration must be loaded before it can be applied to packets. However, most of these causalities are orthogonal to our intents. We therefore only consider packet processing causalities that affect how packets are forwarded or modified.

To encode the causal precedence relationship to a format that can be accepted by a SMT solver, it is intuitive to model packet behavior at NFs using two Boolean val-

ued uninterpreted functions with universal/existential quantifiers. For example, we define `send(n,i,p,t)` as a sending event of packet `p` by NF `n` through interface `i` at time `t`. Similarly, receiving a packet receive is denoted as `recv(n,i,p,t)`. We aggregate all interfaces of a NF into either the internal (`i==0`) or external (`i==1`) interface as some NFs may apply different processing policies for inbound and outbound packets. The send and receive functions return $True$ when the input arguments correspond to a valid event in the network; or they must return $False$.

We show how to capture causal precedence relationships using example SMT encodings for some common stateful NFs.

Stateful firewall. A stateful firewall (Listing 4.2) utilizes ACLs to determine whether to allow or deny a packet from a new connection. ACLs can be modeled using a predicate $acl\_func(a_1, a_2)$, where a_1 and a_2 correspond to the source and destination address of a packet. Packets that belong to established connections are allowed by a stateful firewall even if they are denied by ACLs. An established state indicates that the firewall has received and allowed a reverse packet before.

$$
\begin{aligned}
&\text{Forall } [i_0, p, t_0] \ send(fw, i_0, p, t_0) \text{ Implies} \\
&\text{Exists } [i_1, t_1] \ recv(fw, i_1, p, t_1) \wedge t_1 < t_0 \wedge i_0 \neq i_1 \\
&\\
&\text{Forall } [i_0, p_0, t_0] \\
&send(fw, i_0, p_0, t_0) \wedge \neg\, acl\_func(p_0.src, p_0.dst) \text{ Implies} \\
&\text{Exists } [i_1, p_1, t_1] \ recv(fw, i_1, p_1, t_1) \wedge t_1 < t_0 \wedge i_0 \neq i_1 \wedge \\
&acl\_func(p_1.src, p_1.dst) \wedge p_1 == p_0.reverse
\end{aligned}
$$

Listing 4.2: Encoding of a stateful firewall

Load balancer. A load balancer (Listing 4.3) holds a shared address ($share\_addr(a)$) for a back-end server pool ($server\_addr(a)$). Requests sent to the load balancer are randomly distributed to one of the servers and replies from servers look for a matched request which is sent back by the load balancer.

$$
\begin{aligned}
&\texttt{Forall}\ [p_0, t_0]\ \ send(lb, 1, p_0, t_0)\ \ \texttt{Implies} \\
&\texttt{Exists}\ [p_1, t_1]\ \ recv(lb, 0, p_1, t_1) \wedge\ t_1 < t_0\ \wedge\ share\_addr(p_1.dst) \\
&\wedge\ p_1.src == p_0.src \\
\\
&\texttt{Forall}\ [p_0, t_0]\ \ send(lb, 0, p_0, t_0)\ \ \texttt{Implies} \\
&\texttt{Exists}\ [p_1, p_2, t_1, t_2]\ \ recv(lb, 1, p_1, t_1) \wedge\ recv(lb, 0, p_2, t_2) \wedge \\
&t_2 < t_1 < t_0\ \wedge\ p_2 == p_0.reverse \wedge \\
&share\_addr(p_2.dst)\ \wedge\ server\_addr(p_1.src) \wedge \\
&p_0.dst == p_1.dst == p_2.src
\end{aligned}
$$

Listing 4.3: Encoding of a load balancer

NAT. NAT can either work as a source or a destination NAT. For outbound packets, a source NAT (Listing 4.4) translates the private source IP to a public IP of the NAT, modeled using a predicate $pub\_addr(a)$. If an inbound packet matches an established state, the source NAT translates its destination IP back to the private IP. A destination NAT maintains a one to one destination address mapping for connections initiated from outside networks and has a similar encoding as a load balancer.

$$
\begin{aligned}
&\texttt{Forall}\ [p_0, t_0]\ \ send(nat, 1, p_0, t_0)\ \ \texttt{Implies} \\
&\texttt{Exists}\ [p_1, t_1]\ \ recv(nat, 0, p_1, t_1) \wedge\ t_1 < t_0\ \wedge\ pub\_addr(p_0.src) \\
&\wedge\ p_1.dst == p_0.dst \\
\\
&\texttt{Forall}\ [p_0, t_0]\ \ send(nat, 0, p_0, t_0)\ \ \texttt{Implies} \\
&\texttt{Exists}\ [p_1, p_2, t_1, t_2]\ \ recv(nat, 1, p_1, t_1) \wedge\ recv(nat, 0, p_2, t_2) \wedge \\
&t_2 < t_1 < t_0\ \wedge\ p_2 == p_0.reverse \wedge \\
&pub\_addr(p_1.dst)\ \wedge\ p_0.src == p_1.src
\end{aligned}
$$

Listing 4.4: Encoding of a source NAT

Reverse proxy. A reverse proxy (Listing 4.5) is configured with ACLs specifying which clients have access to content originating at certain servers. Upon receiving a

request that is allowed by ACLs, it initiate a new request to the corresponding server if the contents have not been cached. When receiving responses from the server, it forwards the response to the client who originally requested the content.

$$\text{Forall } [p_0, t_0] \ send(py, 1, p_0, t_0) \ \text{Implies}$$
$$\text{Exists } [p_1, t_1] \ recv(py, 0, p_1, t_1) \wedge t_1 < t_0 \wedge p_0.src == py \wedge$$
$$acl\_func(p_1.src, p_1.dst) \wedge p_0.payload == p_1.payload$$

$$\text{Forall } [p_0, t_0] \ send(py, 0, p_0, t_0) \ \text{Implies}$$
$$\text{Exists } [p_1, p_2 t_1, t_2] \ recv(py, 1, p_1, t_1) \wedge recv(py, 0, p_2, t_2) \wedge$$
$$t_2 < t_1 < t_0 \wedge acl\_func(p_2.src, p_2.dst) \wedge acl\_func(p_0.dst, p_0.src)$$
$$p_1.dst == py \wedge p_0.src == p_1.src == p_2.dst \wedge$$
$$p_0.payload == p_1.payload == p_2.payload$$

Listing 4.5: Encoding of a reverse proxy

4.3 Intent Decomposer

Given a network intent, we can use a SMT solver to check whether the intent is satisfied or not. However, even with the smallest network (18 nodes) we use in our evaluation, the solver cannot return an answer in a reasonable time. To improve the scalability, one key observation is that though a network intent specifies a high level end to end objective, it is possible to decompose it into several sub-tasks, where each task can be checked separately. In this section, we present how the intent decomposer of Epinoia decomposes network intents in two dimensions.

4.3.1 Atomic Address Object

The concept of address objects (mostly referred as zones or aliases) are widely used in network management ecosystems. Assume we are about to configure a set of security

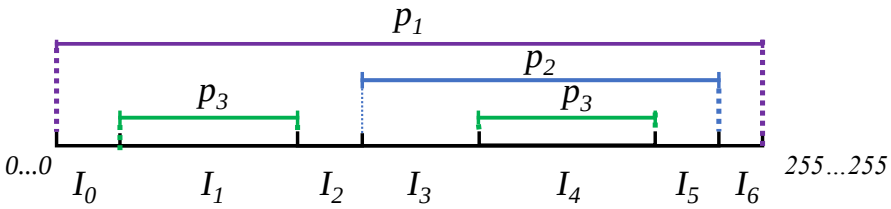


Figure 4.6: Calculating the set of atomic address object for three address objects p_1, p_2 and p_3

rules guarding the servers in a data center to allow traffic from hosts in the marketing department while blocking mobile devices connected to the guest network. Instead of spelling out each address explicitly when a rule is added, we can define address objects as placeholders (e.g., data center, marketing department, guest network); each rule can be applied directly to such address objects. We define the set of atomic address objects which specifies the largest common refinement over the address space given the set of address objects. As is shown in Figure 4.6, three address objects p_1, p_2 and p_3 are represented as ranges with all the endpoints laid out on an axis in increasing order. p_3 has two ranges as it corresponds to two non-continuous subnets.

We consider all the non-overlapping intervals $I_0 \sim I_6$ formed by each consecutive pair of endpoints. The set of atomic address objects can be easily calculated by combining intervals that belong to the same set of address objects. For example, I_1 and I_4 are two separate atomic address objects. $I_0 \cup I_2 \cup I_6$ and $I_3 \cup I_5$ are the other two atomic address objects. In addition, an address object can be represented as a union of a subset of atomic address objects. For example, $p_2 = I_3 \cup I_4 \cup I_5$. We call packets sent from one atomic address object to another atomic address object as a *traffic class*. With the same network state, packets within the same traffic class are treated equally at all NFs in the entire network as they match the same set of processing rules. An endpoint group in an intent can be represented as a union of atomic address objects whose intersection with the endpoint group is not empty. To check an intent between two endpoint groups, instead of querying each pair of end hosts, we can instead simply

check the more compact traffic classes between the two endpoint groups. For example, an intent from endpoint group e_0 to e_1 can be checked using two traffic classes (s_0, d_0) and (s_1, d_0) if $e_0 \cap s_{0,1} \neq \oslash$, $e_0 \subset s_0 \cup s_1$, $e_1 \cap d_0 \neq \oslash$ and $e_1 \subset d_0$. The benefit is two-fold:

Header matching elimination. Most NFs decide processing actions for incoming packets by matching packet headers against processing rules. The natural way to represent a packet and a processing rule for this check is to use bit vectors and check for equality using a bit mask. However, bit vectors are expensive and solvers typically convert them to SAT. In Epinoia, the matching fields of processing rules are represented as a set of integers. The integers are identifiers for atomic address objects. Header matching at NFs are converted to integer membership checking which is more efficient for solvers. For processing rules that modify packet headers (e.g., NAT rule), the modified addresses are also represented as one or more atomic address objects. Depending on a deterministic (one to one mapping) or nondeterministic (one to multiple mapping) modification, an atomic address object is mapped to a certain or random atomic address object.

Adapting to temporal modeling. A solver usually returns a single solution when the set of constraints are satisfiable. Sometimes, we need all solutions for a query, i.e., all hosts in the marketing department should be able to reach the web service. In static modeling, this problem can be solved by testing the satisfiability of the negation of the query, However, with the temporal modeling required by stateful NFs, the negation of the query can be satisfied either with a packet that would be blocked in the network, or a packet sequence that could not have existed because it violates the casual precedence constraints. We need to differentiate between these, and find only true packet loss. To do this, we can only check an intent directly, which could boil down to a large of number of sub-queries corresponding to each pair of end hosts specified in the intent.

With atomic address objects, the number of necessary queries as well as the total time cost is significantly reduced as the checking results can be applied to all end hosts that belong to the same atomic address object.

4.3.2 Path Segmentation

Epinoia pre-calculates all paths for each intent and an intent is satisfied if there is no violation along all potential paths. Along a path, checking an end to end intent can be divided into several sub-tasks, each of which includes a single NF. The intuition is based on two observations: i) Many NFs have concrete constraints on headers of incoming or outgoing packets. For example, a source NAT translates private addresses to its public addresses; A load balancer uniformly distributes packets heading to its shared address to a set of servers. Such concrete constraints are specified in NF configurations and can be propagated along the path, which helps remove redundant information that the SMT solver might otherwise have to discover by itself. ii) State constraints refer to the local packet processing history at a NF. To check if a state could be valid, only constraints within the NF need to be included.

To illustrate the idea of path segmentation, we review the intent (i) in Figure 4.3 within the network graph shown in Figure 4.4. Two potential paths from m_1 to Web are shown in Figure 4.5. Address pairs annotated on each path segment specify the concrete constraints on source and destination addresses of packets that can reach this segment. s_0 denotes the atomic address object corresponds to m_1 while d_0 represents Web. For packets going through FW_1, FW_2 and LB, the source address of packets are always s_0 since no NF along the path modifies the source address. For the last hop, the destination address must be d_0. As a load balancer requires an incoming packet to use its shared address as the destination address, denoted as d_1, the first three segments all have d_1 as destination address. For packets going through FW_1, NAT and LB, the

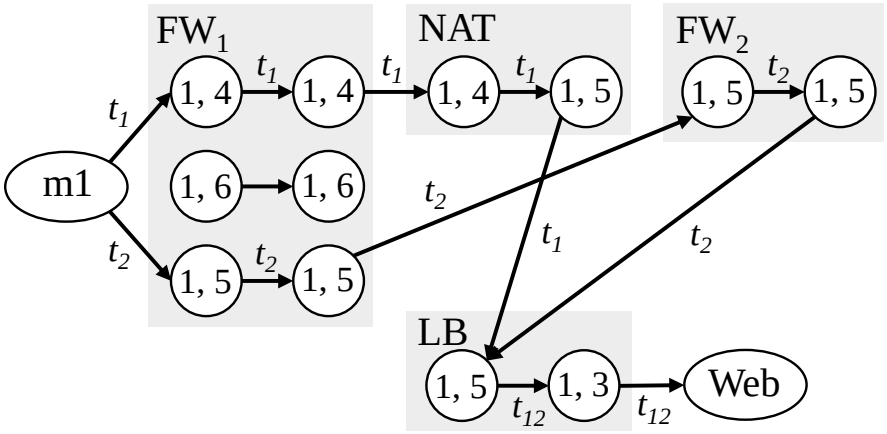


Figure 4.7: The causality graph for the reachability between m1 and Web.

source address is always s_0 while the destination address is modified from d_2 to d_1 and d_1 to d_0 at NAT and LB respectively.

To check the reachability intent between m_1 and Web, Epinoia starts with checking whether those concrete and state constraints within a segment can be satisfied using a solver. A path can be valid only if all segments are satisfiable; otherwise the path is not valid.

4.4 Continuous verification

After checking each segment, Epinoia still needs to combine the results returned by the solver to make sure they are consistent with each other. Meanwhile, upon a network change, Epinoia should be able to identify the affected parts that may need to be rechecked. To achieve these goals, Epinoia maintains a customized causality graph that stores all checked results. Intent checking can be conducted incrementally by traversing the causality graph.

4.4.1 Causality Graph

A node in a causality graph represents either a packet sending or receiving event. Each node is tagged with a pair of atomic address objects specifying the set of source and destination addresses of the packets. An arrow in the graph indicates a causal precedence relationship among two events. The event on the front end depends on and must happen after the event on the rear end. For a single NF, it is straightforward to construct a causality graph of packet sending or receiving events required by the satisfiability assignment from the solver. When there is more than one NF, receiving a packet must be traced back along the selected path to a packet sending node. If the corresponding packet sending node already exists, an arrow is added between the sending and the receiving node. If not, the packet sending is checked within the upward NF and other nodes or edges are added as needed. This procedure continues until the packet receiving node is traced back to an endpoint.

Figure 4.7 shows an example causality graph for the two potential paths in Figure 4.5. Atomic address objects are represented as integers. 1 and 3 correspond to m_1 and Web respectively; 5 is the shared address configured at the load balancer; the NAT maintains two deterministic atomic address object mapping: from 4 to 5 and 6 to 7. Consider the $FW_1 - NAT - LB$ path, possible packets received and forwarded by FW_1 are $(1, 4)$ and $(1, 6)$ since NAT only accepts packets heading to 4 and 6. We assume both packets are allowed by FW_1. Later, only packet $(1, 4)$ goes through NAT as the transformed packet must be $(1, 5)$ to be processed by LB. At LB, packet $(1, 5)$ is changed to $(1, 3)$ and finally sent to Web. Similarly, we add nodes and edges for path $FW_1 - FW_2 - LB$. We add tag t_i along each edge to identify path i. Based on the causal relationship, it's obvious that a path i is valid if the subgraph tagged by t_i has no loop, which indicates that there exists a valid time sequence for all packet sending and receiving events to achieve the end to end intent. In this example, both paths 1 and 2 are

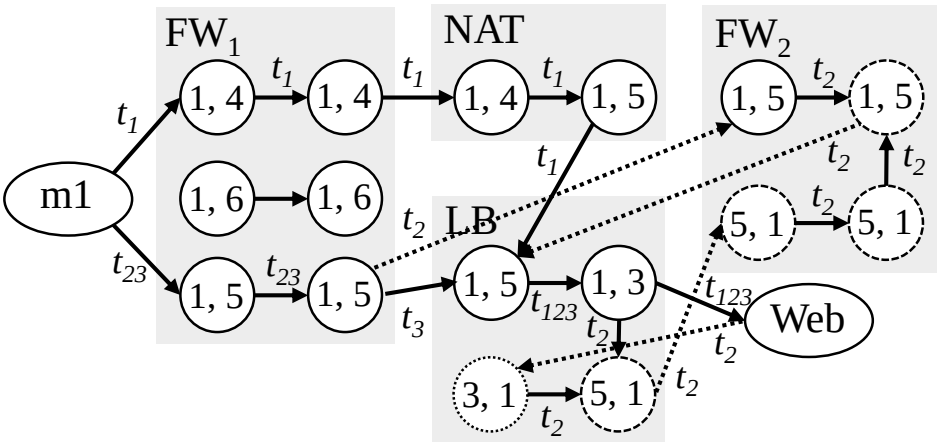


Figure 4.8: The causality graph under a rule insertion and a link up.

valid. To reuse the checked results, both satisfied and unsatisfied checking (not shown for simplicity) results are stored in the graph. In Epinoia, only one causality graph is maintained as the checked results can be shared among paths and intents. When the graph is storing more results, the size of a sub-graph tagged by a path identifier is independent of the complexity of the causality graph.

As events occur to the network, Epinoia identifies affected intents and incrementally updates the causality graph. We handle the following six events.

Adding an address object. When a new address object is added, an existing atomic address object may be divided into two new ones. Nodes and edges related to the atomic address object should be duplicated to reflect the changes. However, an intent needs rechecking only if a new rule using the new address object is inserted.

Deleting an existing address object. Similarly, when an address object is deleted, two existing atomic address objects may specify the same atomic address object. Duplicated nodes and edges in the causality graph are removed. No intent needs to be rechecked.

Inserting a rule. To identify the set of intents that may be affected by the new rule, each node in causality graph maintains an attribute specifying the set of intents and

corresponding paths relying on the node. For example, the packet receiving node $(1, 5)$ in FW_2 is created by intent (i) in Figure 4.3 along path $FW_1 - FW_2 - LB$. When a new rule is inserted at a NF, Epinoia first identifies existing packet receiving nodes that match the new rule and the set of intents in the attributes of those nodes must be rechecked. Meanwhile, the behavior of some other packet receiving nodes may also be affected by the new inserted rule indirectly, even though they do not match the rule. Though the remaining satisfied checking results must not be affected (if they are, their packet receiving nodes should have matched the rule), all other intents with unsatisfied checking results within the NF should be rechecked.

We show how the causality graph is updated when a deny rule for packet $(1, 5)$ is added at FW_2 in Figure 4.8. Now packet sending $(1, 5)$ requires a previous sending of $(5, 1)$, which then is traced back to a sending $(5, 1)$ at LB. At LB, the packet sending $(5, 1)$ relies on a previous sending of $(1, 3)$, which is traced back to a receiving and sending of $(1, 5)$ at LB and FW_2 respectively. After adding all necessary nodes and edges, the subgraph tagged by t_2 introduces a loop, so path 2 becomes invalid. Edges only tagged by t_2 are removed from the causality graph (dotted lines).

Deleting a rule. When a rule is deleted, intents relying on the packet receiving matching the deleted rule need to be rechecked as they will be handled by lower priority rules, and may result in different checking results.

Link up. A link up may lead to two cases where the graph needs to be updated. For each intent, Epinoia first extracts new paths from the pre-calculated path set that traverses the new link and checks if the paths are valid. Meanwhile, Epinoia checks whether packet receiving previously cannot be traced back to endpoints at the two NFs connected by the new link become valid. If so, the set of paths relying on those packet receiving events may become valid. As shown in Figure 4.8, if a link is up between FW_1 and LB, a new path 3 is added by going through FW_1 and LB.

Link down. When a link goes down, all the edges using that link are deleted, which in turn removes all the paths going through those edges.

Complexity analysis. The rule insertion has the highest complexity of $O(nd(V + E))$, where, n is the number of atomic address objects and d is the diameter of the network. V and E denote the number of nodes and edges of a subgraph tagged by a path. When a rule is inserted, Epinoia first checks the set of existing packet receiving events that are affected by the new rule (There exist $O(n)$ such packet receiving events). For each affected events, Epinoia collects checked results along its path. Since the maximum path length is the diameter d, this is $O(nd)$. If all path segments are satisfiable, Epinoia extracts the sub-graph tagged by the path and uses a graph traversal algorithm (e.g., depth first search) to detect if there is a loop. Thus, the overall runtime complexity is $O(nd(V + E))$.

4.4.2 Running Intent Checking Queries

Given an intent, Epinoia divides the intent into sub checking tasks using the intent decomposer. With the checking results maintained by the causality graph, Epinoia calls a SMT solver only when a sub-task has not been checked before. For a reachability intent, valid paths are collected for each traffic class. Each valid path corresponds to a sequence of NFs in the network. Epinoia finds all valid paths that satisfy the NF chaining requirement in an intent. The remaining valid paths correspond to the ones that are reachable but violate the NF traversal requirements. For a block intent, any valid path indicates a potential intent violation.

Once an intent is added, it is evaluated against all future snapshots of the network graph if necessary, unless the intent is removed from the network. For all reported violations, Epinoia reports corresponding network elements or paths the violating traffic is taking. Each piece of configuration is tagged with its intent. Given a reported violation,

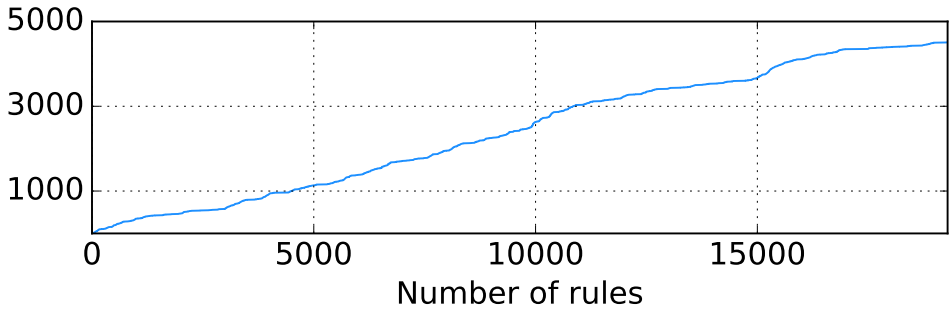


Figure 4.9: Number of atomic address object as number of rules increases.

the tag helps trace back to the intent that generates the configuration.

4.5 Evaluation

We have developed a prototype of Epinoia in approximately 6K lines of Python. To evaluate Epinoia, we first examine how it deals with a real-world enterprise ACL dataset and then investigate the effectiveness of the intent decomposer. Finally we evaluate the runtime performance of Epinoia. All our experiments were done on a machine with 4 cores, 2.93 GHz Intel Xeon Processor and 6 GB RAM. We report times taken when the checking is performed using a single core. We use a SMT solver Z3 [43] for our evaluations. SMT solvers rely on randomized search algorithms, and their performance can vary widely across runs. The results reported are generated from 100 runs of each experiment.

4.5.1 Real-world evaluation

We obtain an ACL dataset from a policy management system of a large enterprise network. These policies are specified using 801 pre-defined address objects located at 137 compartments (groups of subsets). Each ACL rule permits or denies the communication between two address objects, each address object corresponds to one or more

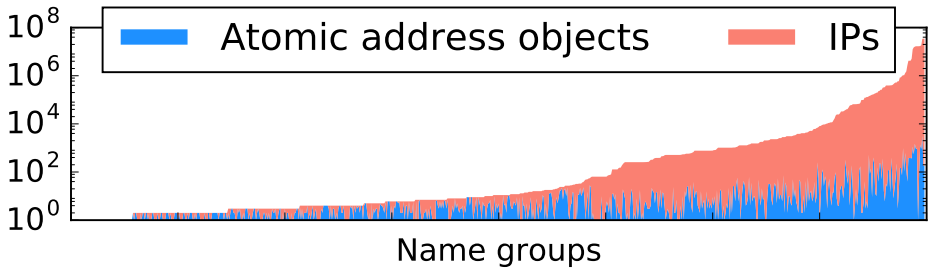


Figure 4.10: Number of atomic address objects and IP addresses for name groups.

IP subnets (address objects may overlap with each other). Given a set of ACLs, we calculate the number of atomic address objects based on the address objects used by those ACLs. As shown in Figure 4.9, the number of atomic address objects increases with a slope less than $1/3$ with increased rule set size. This indicates the similarity between rules with respect to their target address space. In total, there are over 19K ACL rules and 4508 atomic address objects. While some atomic address objects contain large address blocks, about half (2510) of them specify only a single IP. The size of the address objects also varies widely, ranging from a single IP to over 600 non-contiguous subnets (representing around 100 million IP addresses). In contrast, the variation in the number of atomic address objects within an address object is much smaller. As shown in Figure 4.10, address objects are sorted by the number of IPs within the object. Over 90% of address objects have less than 6 atomic address objects. With fewer atomic address objects, it's more likely for Epinoia to achieve better performance when checking group level intents.

Next we use Epinoia to detect potential security breaches that may occur using the ACL dataset. We assume all compartments are connected with a full mesh topology and the ACL policies conduct stateful processing. We measure the time cost to check the reachability for each traffic class between two compartments. The average cost is 0.78 seconds with a maximum of 3.32 seconds. In total, we found 351 potential breaches due to inconsistent deny rules. For example, a packet matches a deny rule

either at the local or the remote compartment, which indicates a block intent from the administrator. However, the block intent may be violated if its reverse traffic is able to pass the compartment.

4.5.2 Scalability

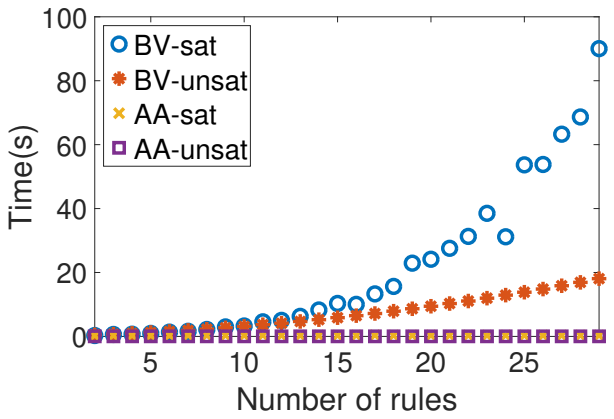


Figure 4.11: Time taken to check a reachability query as # of rules increases.

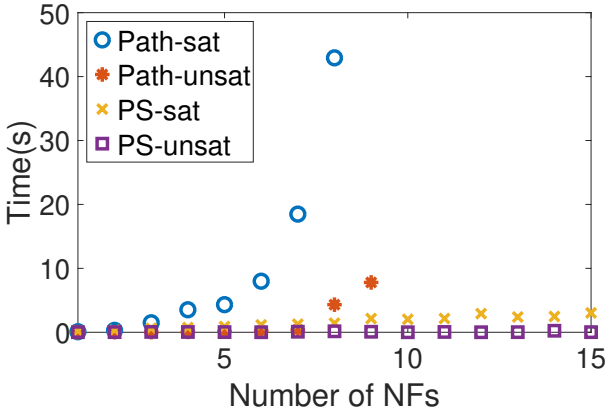


Figure 4.12: Time taken to check a reachability query as # of NFs increases.

To evaluate the scalability of Epinoia, we quantify the effectiveness of the intent decomposer by measuring the time cost of an end to end reachability query. We connect

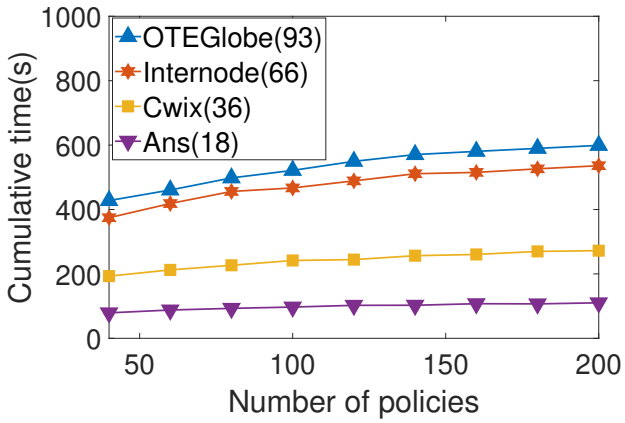


Figure 4.13: Time taken to check all intents

two end hosts with a single firewall. Then we keep inserting ACL rules into the firewall and measure the time cost to check the reachability between the two hosts.

First, we represent addresses as bit vectors (BV) in the SMT encoding and use it as a baseline to show the effectiveness when atomic address objects (AA) are used. As shown in Figure 4.11, the time cost of the query increases exponentially for bit-vector based encoding while all queries cost less than one second when atomic address objects are used. This speeds up the intent checking by 100x when there are 30 rules. The reason is that bit vectors are expensive for SMT solvers and each rule inserted introduces at least 32 extra variables. However, by aggregating addresses to atomic address objects, symbolic variables representing IP prefixes are replaced with integers. A satisfied query requires more time as it needs to calculate valid assignments for all variables in the constraint set, while an unsatisfied query returns immediately when a conflict is found.

To evaluate the benefit of path segmentation, we add additional firewalls between the two hosts to create a firewall chain. We measure the time cost to check the reachability between the two hosts when all the constraints along the path are solved as a whole. This corresponds to a key optimization in VMN [88], where the checking is re-

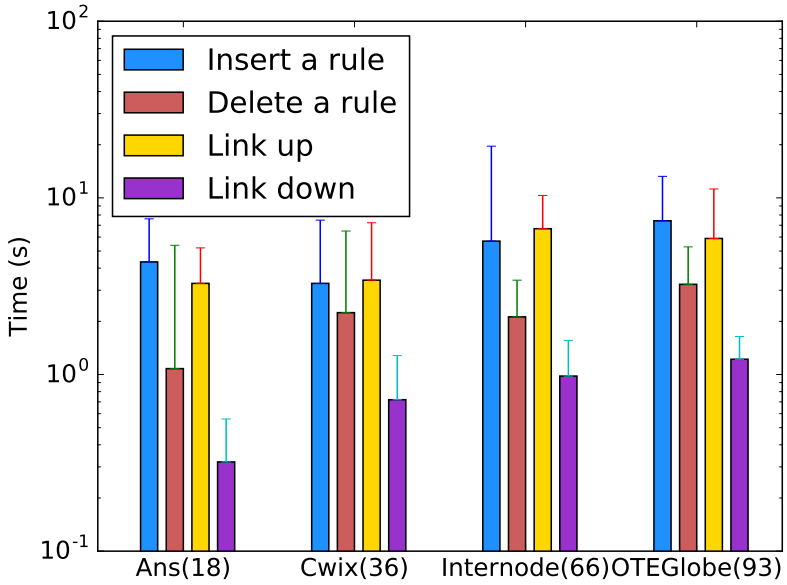


Figure 4.14: Time taken to recheck affected intents per network change.

stricted to the forwarding path between end hosts. When the path segmentation (PS) is applied, we check each firewall one by one and sum up the time cost. As shown in Figure 4.12, when the path is checked as a whole, the time cost increases significantly with increased number of firewalls. The SMT solver Z3 we used in our experiments cannot return before timeout when the number of firewalls is larger than 9 for satisfied queries and 10 for unsatisfied query. With path segmentation, the time cost increases linearly and the maximum cost for satisfied query is 7.73 seconds. For unsatisfied queries, the cost does not necessarily go up with increased number of NFs as the checking process terminates whenever one of the segments cannot be satisfied. The maximum time cost is 0.26 seconds, which highlights the effectiveness of the intent decomposer in Epinoia for large networks.

4.5.3 Runtime performance

In this set of experiments, we evaluate the runtime performance of Epinoia using four topologies from Topology Zoo [67] with number of nodes ranging from 18 to 93. In our experiments, we create 200 network intents, each of which contains 0 to 10 NFs of different types and we randomly attach end hosts belonging to pre-defined address objects to different nodes in the topology. We also randomly assign a NF instance to each node in the topology. Epinoia executes a pre-computation procedure to enumerate the paths for all the intents, which could be costly for large topologies. However, we emphasize that this procedure only needs to be done once and this can be performed off-line.

In the first experiment, we check each intent one after another, and all checked results are stored in the causality graph. Figure 4.13 shows the cumulative time cost to check all intents for the four networks. All time costs grow slightly as the number of polices increases. The reason is that many intents share the same set of sub checking tasks for different traffic classes. The checked results can be reused among intents when there are no network changes.

With all the checked results, we next evaluate how Epinoia reacts to network dynamics. We randomly choose to insert/delete a rule or add/remove a link and measure the time cost for Epinoia to identify and recheck the set of affected intents for each scenario. As each network change may affect a different amount of intents, we report both the average and maximum time cost to recheck the affected intents in each network. As shown in Figure 4.14, the average cost of rechecking after a change is less than 10 seconds, with the maximum for inserting a rule in Internode being close to 20 seconds. Without the incremental checking, a full check is required for all intents whenever there is any change. The average speedup of Epinoia incremental checking is 34x, 79x, 94x and 101x for each network respectively.

4.6 Related Work

To model stateful NFs, existing approaches either work on extracting models by analyzing NF source code [106, 113, 114] or hand crafted models [88] based on expert knowledge. We take a different approach, in which we have designed vendor-agnostic NF configuration models and construct NF forwarding models using key causality relationships. There is a rich body of work for verifying forwarding behaviors in stateless networks [62, 63, 65, 118]. While these work can efficiently check a number of policies such as reachability and loop freedom, it is nontrivial to extend these work to support stateful data planes. There are several proposals on verifying network control planes [35, 52], where the processing is stateful; however, all of those work rely on a converged routing state and cannot be used for stateful NFs. To check stateful networks, Symnet [106] runs symbolic execution over an abstracted NF implementation and SFC-Checker [110] extends the network graph in HSA [63] by adding nodes for each NF state. Both of these approaches are path-based and cannot check state consistency between different NFs. VMN [88] also uses a SMT solver and identifies an end to end slice for each checking. However, VMN only supports block intents and cannot scale to large networks with dynamic updates.

Chapter 5

AutoInfer: Automated Network Intent Inference

5.1 Motivation

Consider an operator planning to write an intent to improve the resilience of a Web service hosted in the enterprise data center. In order to decide the bandwidth requirement, the operator wants to learn what intents about the Web service have already been deployed and how resilient is the current deployment. An intuitive approach is to run start of the art network verification techniques over network configurations and collecting all potential accesses related to the Web service. While the runtime network state will be missing from this analysis, it may derive inaccurate network intents. We present three examples which motivate our idea in AutoInfer to augment configuration analysis with runtime monitoring.

Fig. 5.1 shows an example of endpoints migration, which could happen due to company reorganization or a department/team moving into a new office. While new configurations are installed to support new intents related to the migration, the old ones

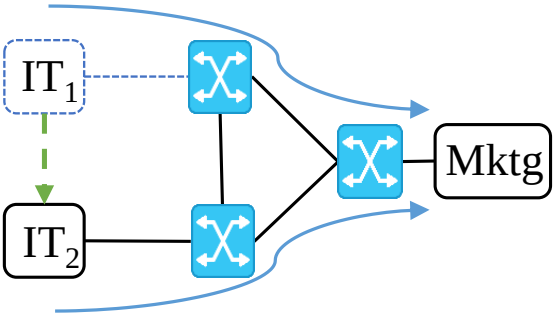


Figure 5.1: Example of endpoints migration. Endpoint group IT_1 migrates to IT_2. The original intent between IT_1 and $Mktg$ no longer exists.

may still exist in the configuration file [16]. As shown in Fig. 5.1, IT department moves from IT_1 to IT_2. The original communication intent from IT_1 to Marketing department $Mktg$ may still be inferred, but without any active traffic.

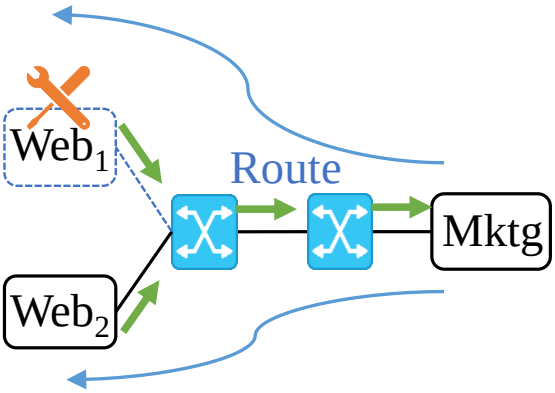


Figure 5.2: Example of rule aggregation. The aggregated route for both Web_1 and Web_2 is still valid when $Web1$ goes down for emergency maintenance. Web_1 cannot be accessed.

To achieve high availability, services may migrate more often compared to endpoints. As shown in Fig. 5.2, there are two Web services $Web1$ and Web_2. Both of them advertise their routes from the left to the right side. The two routes are aggregated into a single route on their way to the Marketing department $Mktg$ since they share the same prefix. When Web_1 experiences an emergency maintenance, existing connections and states on Web_1 are migrated to Web_2. As Web_2 is still alive, the original aggre-

gated route will stay valid [47]. However, the inferred intent from $Mktg$ to $Web1$ does no longer exist since Web_1 cannot be accessed at the moment.

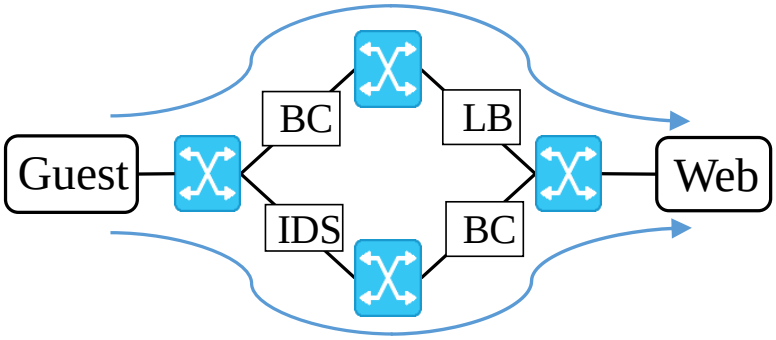


Figure 5.3: Example of equal-cost paths. Though the upper (traverses a byte counter and a load balancer) and lower (traverses an Intrusion Detection System(IDS) and a byte counter) paths between Guest network and the Web service have the same cost, only one path is active at a time.

Finally, another misleading intent may be inferred from configurations when there are multiple equal-cost paths between the source and destination endpoint groups. As shown in Fig. 5.3, between the Guest network and the Web service, there are two paths with the same hop count but different function boxes. While there seems to be a conflict between the two intents inferred along the two paths, this is an example of policy routing [25], only one path/intent is active at a particular time .

5.2 Overview

Next, we provide an intuitive description of AutoInfer (see Fig. 5.4) using a running example.

Input. AutoInfer relies on three input from operators or network OSes: labels, dataplane rules and topologies. No other input is required during the intent inference process. Labels refer to predefined names representing endpoint groups, which are commonly used in existing network management tools and applications [5] [20]. La-

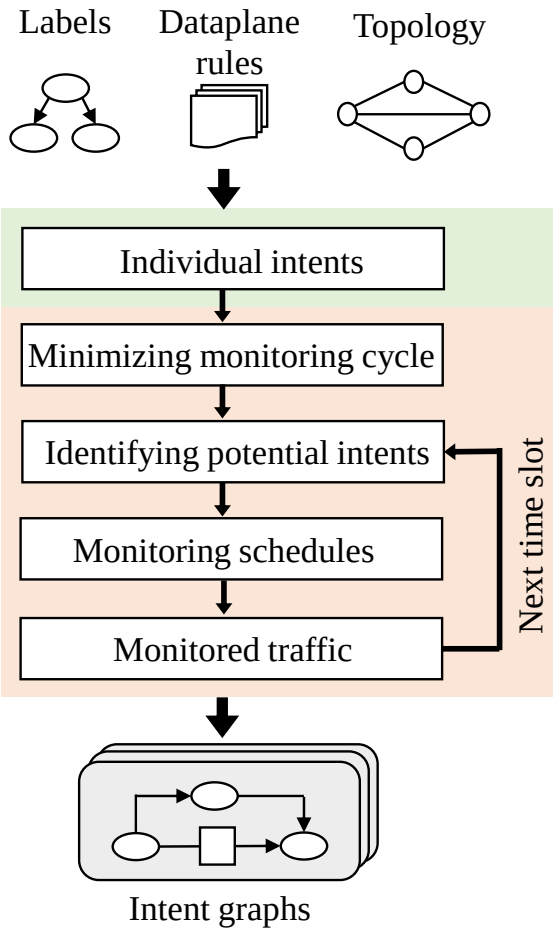
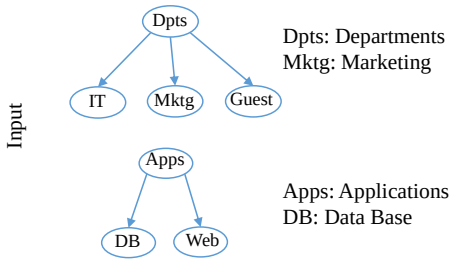


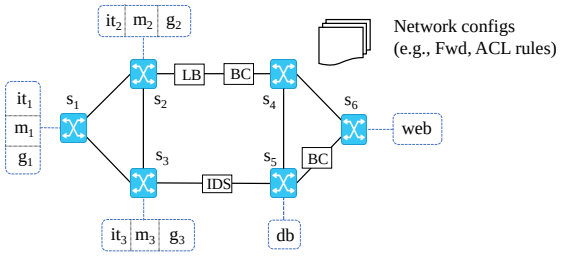
Figure 5.4: Workflow of AutoInfer

bels are naturally organized in tree structures. More specifically, as shown in Fig. 5.5a, the example network has seven predefined labels including three departments (IT, Marketing, Guest) and two applications (Data Base, Web). Each label corresponds to a set of IP addresses or prefixes. AutoInfer also accepts network topology and dataplane configurations (e.g., forwarding/ACL rules) in order to compute potential intents. Fig. 5.5b shows the topology of the example network consisting of six switches (s_1 to s_6) and several function boxes.

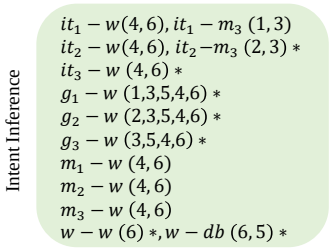
From configurations to individual intents. Given network configurations, AutoInfer obtains potential intents by calculating network reachability between each pair of edge ports along all simple paths. Intersected with the label definition, we can in-



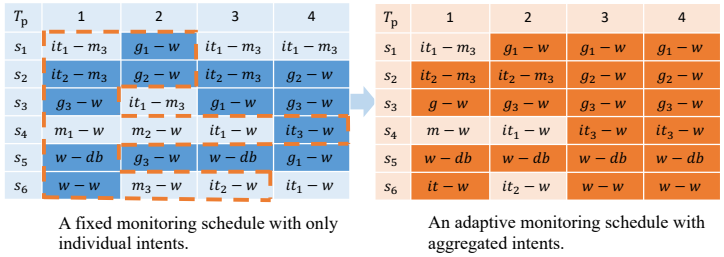
(a) Input label namespace hierarchy



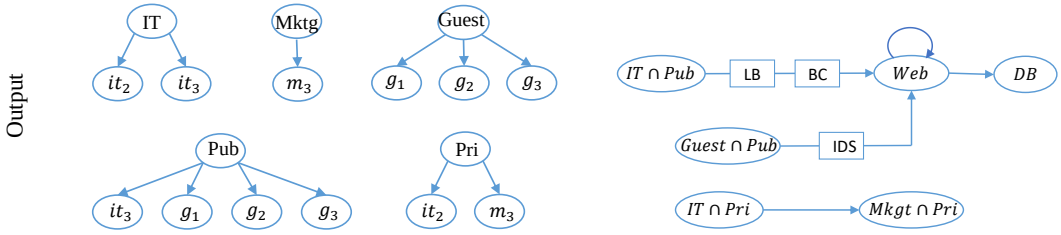
(b) Network configurations and topology



(c) From configurations to potential individual intents and available monitoring spots. ($\star$ indicates the intent with active traffic.)



(d) Process refining monitoring schedules. Intents with monitored traffic are marked in darker colors.



(e) Refined label hierarchy

(f) Inferred network intents

Figure 5.5: A running example of AutoInfer

fer locations of endpoints and their corresponding intents (see Fig. 5.5b). Note that an endpoint group may partially belong to an intent. To represent sub-endpoint groups, AutoInfer creates labels with subscripts. As shown in Fig. 5.5b, some endpoints in the IT department may sit behind switch s_1, denoted as it_1, and are able to talk to Web. This intent is represented as $it_1 - w$. We call such potential intents *individual intents*. Following this method, AutoInfer spells out all potential individual intents.

Fig. 5.5c shows the all individual intents calculated using the network configurations. An individual intent can be monitored on the switches along its forwarding path between the source and destination endpoints (Numbers in parentheses represent potential monitoring spots for corresponding intents). In order to show how AutoInfer works, we assume a subset of individual intents has active traffic, marked with a star at the end of an intent. Note that the function boxes which modify packet headers introduce additional constraints on which spot an intent can be monitored. For example, traffic for the intent $it_1 - w(4, 6)$ goes through the path $s_1 \rightarrow s_2 \rightarrow s_4 \rightarrow s_6$. However, since the load balancer may modify destination addresses, only the downward switches (s_4 and s_6) are valid monitoring spots for the intent.

Adaptive monitoring refinement. With the individual intents extracted from the configurations, AutoInfer conducts monitoring to further infer network intents from the runtime state. Since different intents could contend for more resources than available in the network, it may not be possible to monitor all intent all the time. Monitoring schedules in AutoInfer are made of one or more timeslots. While it is intuitive to spend more resources measuring intents with active traffic, a current latent intent may become active in seconds. To assure fairness, monitoring schedules are initialized periodically. The length of the cycle is chosen to be the minimum number of timeslots to monitor each individual intent for at least once. In the running example, we assume a switch can monitor at most one intent at a time. Limited by the switch resource budget, it takes

at least four timeslots to schedule each individual intent at least once. As shown by the timeslots surrounded by dotted lines in the fixed monitoring schedule of Fig. 5.5d, switches s_4 and s_6 become bottlenecks for intents between IT, Marketing and Web service. Other timeslots are filled accordingly and we call this a minimum fixed schedule. After the schedule for a cycle is decided, all intents are continuously monitored by repeating the minimum schedule.

Limitations of such fixed schedules are: 1) Some intents may experience long delay before they are initially monitored (e.g., intent $it_3 - w$ isn't monitored until the fourth timeslot), which further delays any reaction operators may take by observing such intents. 2) Some intents without active traffic are monitored repeatedly (e.g., intent $it_1 - m_3$ is monitored four times while an active intent $it_3 - w$ is monitored only once). Therefore, those slots are wasted and won't contribute to the intent graphs.

An improvement is to aggregate individual intents to high level intents based on the input label trees (e.g., Intents $m_1 - w$, $m_2 - w$ and $m_3 - w$ are aggregated to $m - w$). High level intents are scheduled with higher priority whereas a sub-intent is scheduled only if its high level intent has active traffic. Otherwise, the sub-intents will not be monitored within the same cycle. However, an overly aggregated scheme may lead to resource underutilization. In an extreme case, all individual intents are aggregated into a single high level intent and thus only one intent is available to schedule, leaving other switch rearouses wasted. To address the problem, the monitoring schedule in AutoInfer is refined adaptively with objectives to maximize intent coverage as well as resource utilization (see the adaptive monitoring schedule in Fig. 5.5d).

Compared with the fixed monitoring schedule, the adaptive monitoring schedule brings two benefits: 1) In addition to individual intents, it can also generates intent graphs for aggregated intents which provide operators with a high level view of network states in a timely manner. It's especially useful when many intents exist. 2) On the other

hand, it improves the efficiency of the monitoring by giving up the inactive intents at an early stage and only focuses on active intents. As shown in Fig. 5.5d, in the fixed monitoring schedule, 10 out of 24 slots do not capture any traffic, the ratio is decreased by x2.5 to 4/24 in the adaptive monitoring schedule.

Output. Labels are often defined over network structures or company organizations. New policy labels can be inferred based on obtained intent graphs to suggest operators configure and reorganize their networks. As shown in Fig. 5.5e, it_3 and $g_{1\sim3}$ are included in a public (pub) group since they all have accesses to the Web service while it_2 and m_3 are forming a private (pri) group.

Using labels, intent graphs are displayed to operators and are continuously updated based on the captured traffic. Fig. 5.5f shows intents graphs AutoInfer created for the running example.

5.3 From Configurations to Individual Intents

Given input configurations, the first operation performed by AutoInfer is to identify individual intents and their potential monitoring spots. We now detail how this happens.

Network Models. We follow existing work on network verification [63] [118] to model a packet header as a flat sequence of ones and zeros. Formally, a header is a point in the $\{0,1\}^L$ space, where L is an upper bound on the header length. A wildcard expression is a sequence of L bits where each bit can be either 0, 1 or x. The whole header space is defined as a union of wildcard expressions.

We model the network as a set of boxes with external ports. A box is modeled using a transfer function T:

$$T(h,p): \quad (h,p) \rightarrow \{(h_1,p_1),(h_2,p_2),...\} \tag{5.1}$$

In general, the transfer function may depend on the input port to model input-port-specific behavior and the output may be a set of header-port pairs to allow multicasting. More precisely, transfer functions model their protocol dependent functions. For instance, an IP lookup can be modeled by a masking AND while header updates can be represented by a masking AND followed by a rewrite OR.

Computing Individual Intents. For each edge port, we consider the space of all headers leaving the source, then track this space as it is transformed by each successive box along a simple path. This process terminates when either another edge port is reached or no header space remains. If an edge port is reached, we trace the remaining header space backwards (using the inverse function at each step) to find the set of headers the source can send to reach the destination. For a simple path with reachable source and destination addresses, we intersect them with the pre-defined labels to get an individual intent. Switches along the path are potential monitoring spots for the intent. If an intent contains a function box which modifies destination addresses, monitoring spots are restricted to its downward switches where traffic can be captured with correct destinations. Similarly, for a function box which modifies source addresses, the monitoring spots can only be its upward switches. An alert is triggered when no available monitoring spot is found.

5.4 Adaptive monitoring refinement

After analyzing configurations, we end up with a group of individual intents and their potential monitoring spot. The next step performed by AutoInfer is to schedule those monitoring tasks, creating network intent graphs. While it is intuitive to seek high monitoring efficiency (e.g., to spend more resources on intents with active traffic), AutoInfer also values the fairness among all intents (e.g., to monitor each intent as often as possible). To achieve the goal, AutoInfer firstly identifies a minimum cycle in which

Type	Symbol	Meaning
Constant	T	the set of all timeslots
	W_i	weight assigned to intent i based on importance
	CAP_s	capacity of swtich s
	α	weight assigned to the second objective of maximizing switch utilization
	ρ	weight of penalty for changing the spot for a continuous monitoring task
Variable	I_{it}	indicator variable set as 1 if intent i is scheduled at timeslot t
	P_{ist}	indicator variable set as 1 if intent i is monitored at switch s at timeslot t
	U_t	indicator variable set as 1 if any monitoring task is assigned at timeslot t
	δ_s	slack variable for switch s
	θ_{is}	penalty to represent changing of monitoring spot assignment for intent i

Table 5.1: Symbols and notions.

each individual intent can be scheduled for at least once. Monitoring schedules are initialized at the beginning of each cycle. Within a cycle, AutoInfer adaptively refines monitoring schedules to compute a maximum filling for each timeslot, hence increasing inference accuracy.

Symbols and functions. Table 5.1 and Table 5.2 list the symbols and functions used in our formulation, along with their meanings.

5.4.1 Calculating a minimum cycle

To identify the length of a minimum cycle, we need to schedule each individual intent for exactly one timeslot, with the goal to minimize the total number of timeslots. This problem can be reduced from a bin packing problem, and therefore it is NP-hard. To improve time efficiency, AutoInfer first computes an upper bound on the size of the minimal cycle, using a First-Fit heuristic which schedules all intents in increasing

Function	Meaning
$indivInts$	returns all individual intents
$switches$	returns all switches
$spots(i)$	returns available monitoring spots for intent i
$ints$	returns all potential intents for the current timeslot
$descInts(i)$	returns all descendants of intent i in the current intent pool
$actInts$	returns all monitored intents having active traffic in the current timeslot
$actSpot(i)$	returns the spot selected to monitor intent i in the current timeslot

Table 5.2: Functions and notions.

order with respect to the number of monitoring spots. The computed upper bound is then exploited to compute a minimum cycle, using a Integer Linear Program (ILP) formulation.

We use indicator variables, also known as binary variables, which can take the value of 0 or 1. For example, U_t is an indicator variable for timeslot $t \in T$. It will take the value 1 if any intent is scheduled to be monitored in timeslot t. The goal is to minimize the length of a cycle.

$$\textbf{Objective} : min \sum_{t \in T} U_t \tag{5.2}$$

The objective should be achieved with the following constraints:

Every individual intent must be scheduled. $\forall i \in \ indivInts$:

$$\sum_{s \in spots(i)} \sum_{t \in T} P_{ist} = 1 \tag{5.3}$$

In any timeslot, monitoring tasks assigned to a switch must be equal or less than its capacity. $\forall\, t \in \ T, \forall\, s \in \ switches$:

$$\sum_{i \in indivInts} P_{ist} \ \leq \ U_t \ \times \ CAP_s \tag{5.4}$$

A timeslot is used if any intent has been scheduled to be monitored in the timeslot. $\forall\, t \in\ T$:

$$\sum_{i \in indivInts} \sum_{s \in\ switches} P_{ist}\ \geq\ U_t \tag{5.5}$$

Intent should be scheduled to timeslots in sequence. $\forall\, t, t' \in\ T, t \leq\ t'$:

$$U_t\ \geq\ U_{t'} \tag{5.6}$$

5.4.2 Calculating a maximum filling

Within a cycle, a maximum filling is calculated for each timeslot. The primary goal is to maximize the intent coverage and its secondary goals are to maximize resource utilization and minimize spot changes during the monitoring process. Before we show the optimization problem and the heuristic algorithm to achieve our goals, we first explain how to identify potential intents and how they can be adaptively refined across different timeslots.

Identify potential intents. As shown in the running example in Fig. 5.5, aggregated intents are helpful to quickly get rid of inactive intents, leaving more resources for active intents. However, not every aggregated intent is useful and needless aggregation may increase the number of potential intents, slowing down the scheduling process. AutoInfer first creates an intent relationship graph and then prunes redundant intents. The remaining intents in the graph are the set of potential intents to be monitored.

Fig. 5.6 shows an example intent relationship graph built using a subset of individual intents (on the right side) from our running example. We can see that an intent is upgraded by either source or destination label each time. An arrow points from a father intent to all its children intents. The pruning process starts from the left, removing all intents which cover the same set of individual intents as their children. In the example, only three aggregated intents are left, shown in green boxes.

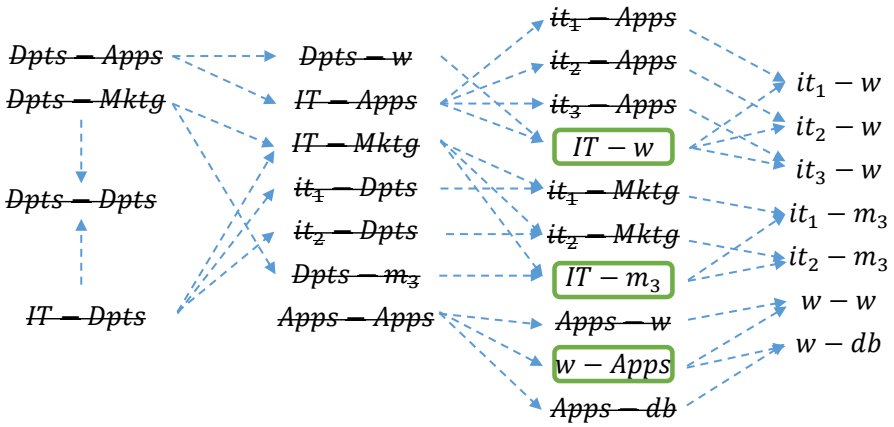


Figure 5.6: Example of an intent relationship graph

By computing the intersection of monitoring spots of individual intents, we can obtain the monitoring spots for the corresponding aggregated intent. Note that the potential intents only need to be calculated once and can be reused for all cycles.

Refining intent list. Across different timeslots within a cycle, the potential intent list is refined adaptively based on what intents have been scheduled and their monitoring results in previous timeslots. More specifically, AutoInfer follows three rules to update the intent list:

1. Once an intent is scheduled, remove all its ancestors from the intent list.

2. If an intent is measured to be inactive, all its descendants including itself are removed.

3. If an aggregated intent is active, just remove the intent itself. Its descendants will stay in the intent list.

Ancestor and descendant intents can be obtained easily by traversing the intent relationship graph. The intent list is reset to the original set when a new cycle starts.

Maximizing intent coverage. Monitoring scheduling depends on switch capacity. For a timeslot, it may not possible to schedule all intents at once. The primary goal of

the scheduling is to maximize the intent coverage, defined as the weighted sum of all potential intents. AutoInfer uses weight to represent the importance of an intent. The weight of an aggregated intents is calculated as the sum of weights of all its individual descendants.

$$\textbf{Objective} : max \sum_{i \in ints} W_i \, I_i \tag{5.7}$$

There are three constraints that need to be considered in formulating the optimization problem. If intent i is scheduled, i.e., $I_i = 1$, exactly one monitoring spot should be reserved.

$\forall i \in \ ints$:

$$\sum_{s \in spots(i)} P_{is} = I_i \tag{5.8}$$

The total number of intents scheduled to be monitored on a switch should be less than the switch capacity.

$\forall s \in \ switches$:

$$\sum_{i \in \ ints} P_{is} \leq \ CAP_s \tag{5.9}$$

To avoid double booking, an intent and its descendants should not be scheduled for the same timeslot.

$\forall i \in \ ints, I_i = 1$:

$$\bigcup_{j \in \ descInt(i)} I_j = 0 \tag{5.10}$$

An ILP that considers all potential monitoring spots $spots(i)$ of an intent may become inefficient since the number of spots grows with the size of the network. AutoInfer uses a heuristic algorithm [58] which uses a random subset of valid monitoring spot as the candidate spots.

Maxmizing resource utilization. While scheduling more high level aggregated

intents improves intent coverage, there will be far less remaining intents that can be scheduled, leaving the resource wasted. Slack variables are introduced for switches to represent unused capacity.

$\forall s \in \ switches$:

$$\sum_{i \in \ ints} P_{is} = CAP_s - \delta_s \tag{5.11}$$

We incorporate slack variables as penalty in the objective function and use variable α to control the weight of the penalty associated with the total wasted capacity. By increasing α, we can ensure maximum utilization is guaranteed.

$$\textbf{Objective} : max \sum_{i \in ints} W_i \, I_i - \alpha \times \sum_{s \in switches} \delta_s \tag{5.12}$$

Minimizing monitoring spot changes. For an intent which is scheduled for multiple continuous timeslots, changing its monitoring spot requires activating as well as deactivating monitoring rules at switches and may lead to additional changes at other switches. While obtaining an optimized schedule for all timeslots is hard, we use a greedy approach, adding a secondary goal of minimizing monitoring spot changes from the previous optimization solution. Indicator variables associated with switches can be used as a signal to represent monitoring spot changes. The value of P_{is} for switch s and intent i can change from 1 in the initial solution to 0 in the new solution in two scenarios: 1) intent i is monitored at some other switch, and, 2) no switch is configured to monitor intent i. Both require modifying monitoring rules. We minimize such changes using a variables θ_{is}. As shown in Eqn. 5.13, for an active intent $i \in actInts$, changing the value of P_{is} from 1 to 0 in the new solution will set θ_{is} to 1. These variables represent monitoring spot changes and are used to create the penalty function in Eqn. 5.14.

$\forall i \in \ actInts, s = actSpot(i)$:

$$P_{is} = I_i - \theta_{is} \tag{5.13}$$

Similar to Eqn. 5.12, AutoInfer uses another variable ρ to control the penalty associated with spot changes. The main objective is still maximizing intent coverage, which can be achieved by assigning ρ a low value.

$$\textbf{Objective} : max \sum_{i \in \ ints} W_i \ I_i - \rho \times \sum_{i \in \ actInts} \theta_{is} \tag{5.14}$$

5.5 Evaluation

We start by evaluating the algorithm pipeline of AutoInfer using synthetic benchmarks over realistic topologies, to answer the questions: 1) Can AutoInfer improve the accuracy of intent inference? 2) Does it scale to large networks and is suitable for online use? Then, we validate the useability of AutoInfer using an emulated testbed on Mininet [72].

5.5.1 Methodology

Benchmarks. We select fifteen topologies from Internet Topology Zoo [67] and group them into three categories, namely Small, Medium and Large based on their size. The Small networks have approximately thirty nodes while Medium and Large networks have one hundred and two hundred nodes respectively. We create label trees ranging from two to four levels. Each leaf label is further divided into five sub-labels. To obtain endpoints, we randomly attach all leaf labels to different nodes in each network. Potential individual intents and their corresponding monitoring spots are calculated as shortest-paths based on the number of hops between endpoints. For each

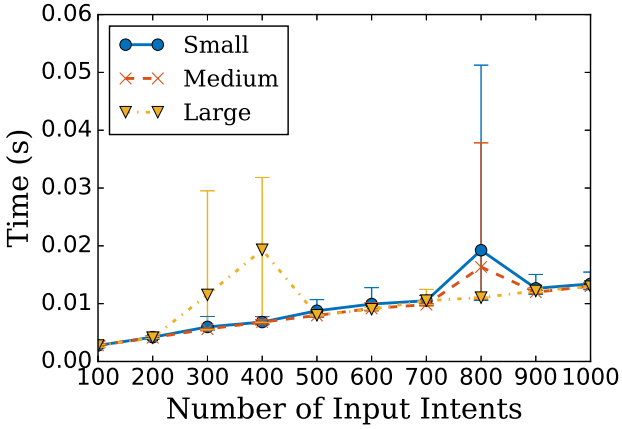


Figure 5.7: Time cost of intent aggregation

experiment, we randomly sample a number of individual intents as the input for the intent aggregation process and gradually increase the input size to evaluate the scalability of AutoInfer. Unless specified otherwise, the number of candidate monitoring spots for each intent is limited to five. The capacity of a switch is set to one. That is, only one monitoring task can be assigned to a switch at a time. The weight of each individual intent is set to one and the weight of an aggregated intent is obtained by adding weights of all its individual descendant intents.

We implemented an AutoInfer prototype using Python and Gurobi [9], a state of the art ILP solver. We performed more than one hundred experiments for all topologies and report the average for each algorithm. All our experiments were done on a machine with eight cores, 3.40 GHz Intel Core i7-6700 Processor and 32 GB RAM.

5.5.2 Intent Aggregation

Given a set of individual intents, AutoInfer first performs the intent aggregation process to obtain a list of potential intents for the monitoring phase. Fig. 5.7 shows the execution time of intent aggregation for Small, Medium and Large networks. We can

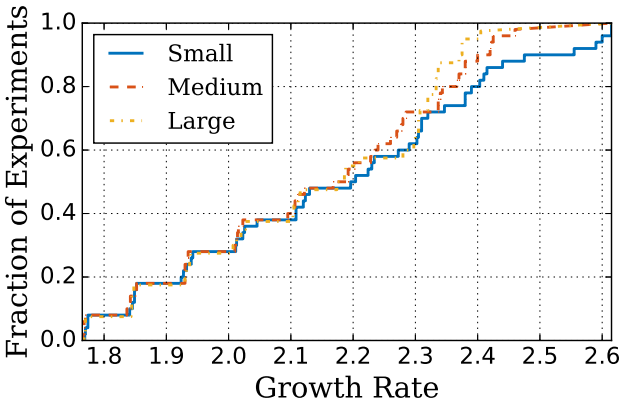


Figure 5.8: CDF of growth rate of number of intents

see that the time cost grows slightly with the size of input individual intents increasing from one hundred to one thousand, which confirms that algorithms used in the intent aggregation are linear in the size of input intents. The maximum cost is still less than 0.06 seconds. For most of experiments, networks of different size share similar time costs. This is because the only difference in this phase between different networks is calculating potential monitoring spots to determine if an intent should be pruned. Such calculations are very efficient and do not incur notable delays for large networks.

While adding aggregated intents improves the monitoring efficiency, it also increases the size of potential intent list, which may slow down the monitoring scheduling process. Label trees with larger depth tend to introduce more aggregated intents. To study how much the size of intent list increases, we calculate the growth rate as the number of intents after aggregation divided by the number of input individual intents for each experiment and show the Cumulative Distribution Function (CDF) in Fig. 5.8. From the figure, we can see that growth rates of all networks fall between 1.8 and 2.6. The intent size of 80% of networks increases less than 2.4x.

Note that the intent aggregation only need to be performed once and the result intent

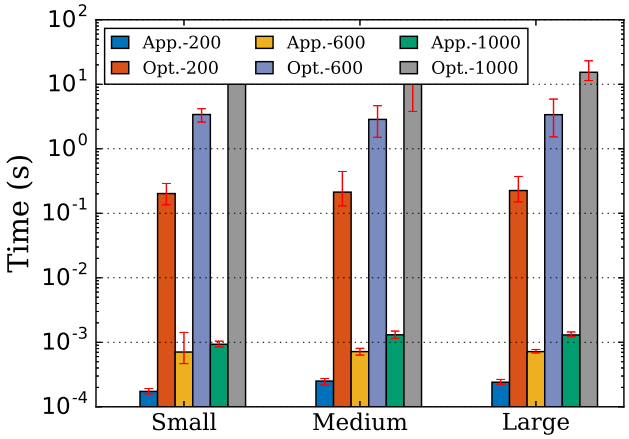


Figure 5.9: Time cost to identify the minimum cycle

list remains the same for further cycles if the network does not change. The speed of intent aggregation affects AutoInfer's ability to recompute the intent list online. A smaller execution time enables AutoInfer to quickly react to changes (e.g., link failures and routing changes), hence improves the accuracy of inferred intents.

5.5.3 Scheduling Performance

Calculating the minimum cycle. In AutoInfer, monitoring schedules are organized in minimum cycles. We first evaluate the time cost to identify the length of the minimum cycle using the approximated heuristic (App.) and the optimized (Opt.) algorithm discussed in Sec. 5.4.1. Fig. 5.9 shows the time cost for different networks when the number of input individual intents is equal to two hundred, six hundred and one thousand. We find that the approximated method is extremely fast and the largest cost among all experiments is less than 10 ms. In contrast, the cost of the optimized method grows exponentially with respect to the number of input intents. For all networks, the costs can be up to tens of seconds when there are one thousand individual intents.

We remind that identifying the minimum cycle is also a one-time calculation and

Topology	Optimality Gap (%)				
	200 intents	400 intents	600 intents	800 intents	1000 intents
Small	4.3	4.9	3.6	3.9	3.9
Medium	13.3	0	7.9	1.6	4.4
Large	20.8	15.4	7.7	10.6	8.4

Table 5.3: Optimality gap between the approximated heuristic and optimized algorithm.

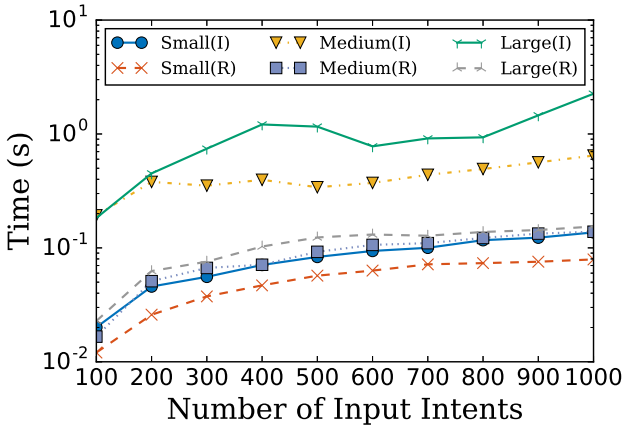


Figure 5.10: Time cost to compute a monitoring schedule

can be done offline. Though the optimized method costs more time, it achieves a smaller cycle length in most of experiments. Table 5.3 shows the optimality gap ($\frac{len_{App}-len_{Opt}}{len_{Opt}}$) between minimum cycles obtained by the approximated and the optimized method. In most cases, the optimality gap is smaller than 10%. The largest gap is 20.8% for Large networks with two hundred input intents.

Calculating a maximum filling. Within a cycle, AutoInfer refines the monitoring schedule across each timeslot adaptively. LP solvers like Gurobi use "warm start", which allow them to start from the existing solution. For minor changes in constraints or objective functions, "warm start" can be significantly faster [9]. AutoInfer takes advantage of "warm start" and avoids an exhausted calculation by adaptively updating the

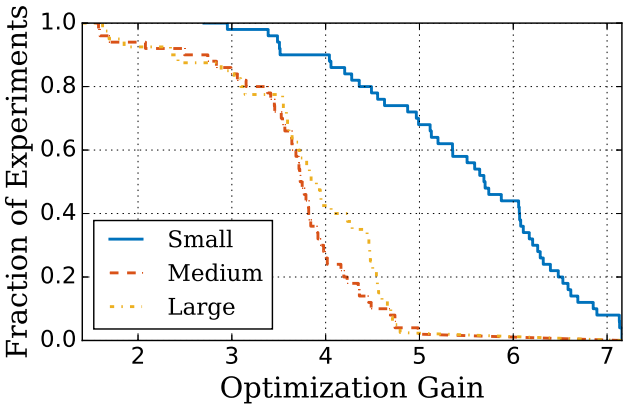


Figure 5.11: CCDF of increased slots allocation

potential intent list for each time slot. Fig. 5.10 shows the execution time to compute the initial and the rest of monitoring schedules (denoted as (I) and (R)). As shown in the figure, the average cost to compute a monitoring schedule with "warm start" increases slightly with the number of input intents. Compared with the initial calculation, it reduces the time cost by at least an order of magnitude for Medium and Large networks. In AutoInfer, after an initial monitoring schedule is obtained, it is reused at the beginning of each cycle and refined adaptively within a cycle. Therefore, the maximum cost to compute a monitoring schedule is less than 0.15 seconds, which highlights that AutoInfer is suitable for online use.

Optimization gain. We evaluate the optimization gain of AutoInfer over a fixed monitoring schedule, in which a cycle filled with a fixed monitoring plan of individual intents is repeated continuously. The optimization gain is calculated as the average number of time slots assigned to each active intent using AutoInfer divided by the corresponding number using the fixed schedules. Fig.5.11 shows the Complementary Cumulative Distribution Function (CCDF) of the optimization gain. We can see that 60% of experiments for Medium and Large networks have increased the slots allocation

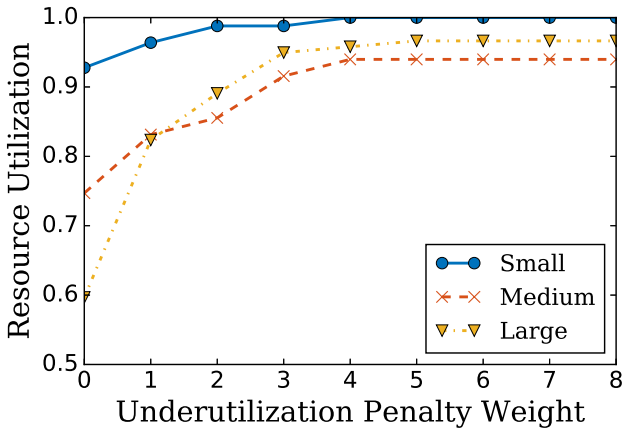


Figure 5.12: Resource utilization with different penalty weight

by at least 3.7x. For Small networks, that value goes up to 5.4x.

Resource utilization. Here we show that having a high penalty weight (α) associated with the unused switch capacity allows us to schedule more intents, and hence higher resource utilization rate. Resource utilization in this experiment is calculated as the ratio of the number of intents scheduled to the total capacity of switches. We vary the penalty weight (α) from zero to eight. As shown in Fig. 5.12, the resource utilization rate is directly proportional to α. Note that increasing the utilization requires more low level intents to be scheduled which may decrease the intent coverage as the primary goal. Setting α to 3.0 allows AutoInfer provides a decent intent coverage while still enabling it to keep wasted resources as low as 10%.

Reducing monitoring spot changes. Here we evaluate the performance of our greedy algorithm to reduce monitoring spot changes. We increase ρ in Eqn. 5.14 from 0 to 1. For intents that are scheduled for two consecutive time slots, we measure the percentage of the intents with the same monitoring spots for all networks. As shown in Fig. 5.13, with ρ equals to 0, about 40% of intents are monitored at the same spots for Small networks. For Medium and Large networks,only less than 30% of intents are

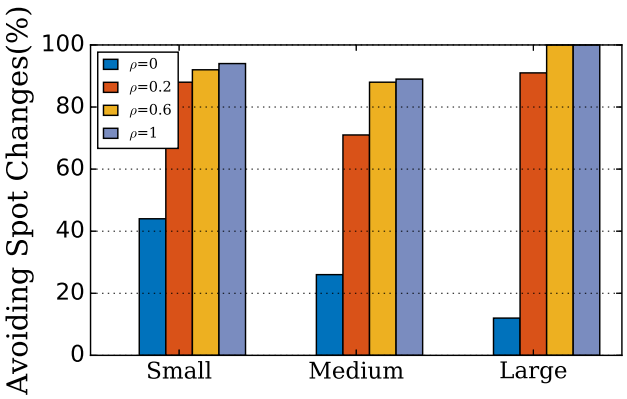


Figure 5.13: Performance of heuristic algorithm to reduce spot changes

assigned with the same spots. When ρ increases, the corresponding percentage quickly goes up to at least 70% for all networks. In AutoInfer, we set ρ to 0.2 since all larger values of ρ achieve similar performance and a lower ρ should be selected to maximize the primary goal.

5.5.4 Testbed Evaluation

To evaluate the practicality of AutoInfer, we integrate our prototype into Mininet running Open vSwitch v2.5.8 [17].

Intent monitoring. Most commercial routers and software switches (e.g., Open vSwitch, P4 switches [19]) support counting packets/bytes that matches a rule (e.g., an IPv4 five-tuple, a VLAN tag) using a counter. To monitor intents, AutoInfer installs rules on switches to passively collect those counters. Due to resource constraints, most routers or switches only support a limited amount of counters [126] [108], so as limited counted criteria, which prevents AutoInfer from capturing practical intents consisting of a large number of flows. To overcome this limitation, AutoInfer triggers intent monitoring using dynamic ACLs. More specifically, counters on switches are configured to

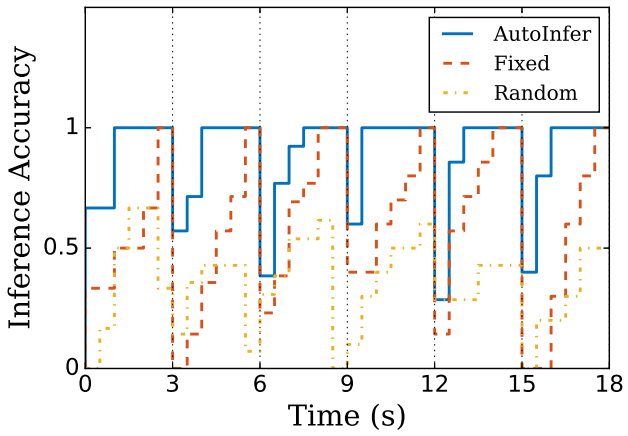


Figure 5.14: Inference accuracy

match a specific tag. AutoInfer then dynamically update ACLs to add that tag to only the packets belonging to the intent to be monitored. Monitoring rules are activated by executing a pre-loaded script on each switch. The script specifies a list of flows for an intent and an active duration.

Choice of timeslot duration. AutoInfer schedules small monitoring tasks lasting for one timeslot. The length of a timeslot should be selected based on the used monitoring technology. To test how short a timeslot can be in our testbed, we create two end hosts connected with a single switch, with one host sending packets to the other at full speed. We write a script to periodically query the packet/byte count corresponding to the rule which forwards packets between two hosts. An observation is that the counter in Open vSwitch is not updated in real-time, which determines the minimum timeslot we can have in AutoInfer. We gradually increase the time interval between two queries util the counter reflects the real-time traffic volume correctly. In the following experiment, we conservatively set the timeslot length to 500 ms.

Intent inference accuracy. To show the practicality of AutoInfer, we measure the intent inference accuracy in Mininet using the network from our running example (see

Fig.5.5). Besides the individual intents shown in the example, we add more potential intents from IT to Data Base, IT to Marketing and Guest to Marketing, making the total number of 32 individual intents. By applying the calculation in Section 5.4.1, the length of the minimum cycle is set to 6 timeslots. For each cycle, endpoints in the network generate traffic for each individual intent with a probability of of 40%. The inference accuracy is defined as the percentage of active intents the monitoring algorithm has captured for the current cycle. To show the improvement of AutoInfer, we also evaluate the performance of the fixed scheduling and random sampling. The fixed scheduling calculates a fixed monitoring plan for all individual intents ensuring that each intent is monitored at least once. The plan is repeated for all the cycles. The random sampling randomly decides which intent to monitor at a spot for each timeslot. The length of a timelsot is set to 500 ms. Fig. 5.14 shows the inference accuracy of the three methods. From the figure, we see that both AutoInfer and the fixed scheduling are able to capture all active intents before the end of each cycle while the random sampling cannot provide guarantee on accuracy. For most of cycles, AutoInfer captures all active intents within two timeslots which is 2x faster than the fixed scheduling. This result is consistent with the optimization gain of AutoInfer since more timeslots will be only allocated to active intents.

5.6 Related work

Recent work on network policy management [30, 66, 92, 105] have enabled network operators to create different types of network intents more expressively and conveniently. Their work mostly focus on the techniques that operators can use to specify their intents and how the intents are converted into each details of practical deployment. However, in many cases, knowing what intent to create is hard and our work complements existing policy specifications to assist operators to understand the run-

time network state. Considerable work have been done on network monitoring and they can be broadly divided into two categories. Some example work [74, 101, 119] designed various sketch-based structures to support different monitoring applications. Some other work [83, 107] discussed monitoring frameworks that cooperate switches and end hosts. Our monitoring algorithm is similar to [108]. While it focuses on monitoring pre-defined flows, our work tries to extract such popular flows from all potential flows and keeps adapting to the runtime network state. A recent work [39] proposed a tool to assist network operators in reasoning network forwarding behaviors. It used a heuristic algorithm to summarize the traffic forwarding records. Our work aims to assist operators to create or refine network intents for a running network and does not rely on any pre-knowledge of the runtime network state. Companies [4, 27] working on IBN have also included features on automated intent inference. Most of those functions are based on best practice of network management. While such functions can provide generic advices, our work is compatible with them and can be combined to provide a comprehensive network intent inference functionality.

Chapter 6

Conclusion

Managing a large packet network is a complex task. Network management benefits from automated tools to fulfill each functionality in its control loop. This dissertation presents efficient methods to improve the reliability and useability of network automation.

We propose AP Classifier for network-wide packet behavior identification that can be utilized by many important network management applications. We design algorithms to construct the AP Tree for a network, which can be used to quickly classify a packet to an atomic predicate. Each atomic predicate represents the network-wide forwarding behaviors of a set of packets. Experimental results using the datasets of two real networks show that the proposed AP Tree construction algorithm can optimize the average depth of leaf nodes. AP Classifier can process millions of packet queries per second. The speed is faster than existing tools by at least an order of magnitude. Furthermore, it uses only a few MBs memory. It can be updated in real time and is robust under dynamic data plane changes.

As an application of AP Classifier, we present SICS, a middlebox outsourcing framework that protects the private information of packet headers and middlebox rules. Compared with existing methods, SICS has several unique advantages including a

stronger security guarantee, high-throughput processing, and support for quick updates. SICS assigns each packet a label identifying its matching behavior in a service chain and all middlebox processing in the cloud is based on labels. We use a prototype implementation and evaluation on VPC and local computers to demonstrate the feasibility, high performance, and efficiency of SICS.

We present our intent checking solution, Epinoia. Epinoia efficiently supports stateful networks with a variety of network functions. Epinoia includes vendor-agnostic network function modeling combined with capturing causality precedence relationships for incremental intent checking. A comprehensive evaluation shows that Epinoia can check network intents in under 10 seconds per network update and reduce checking time by a factor of up to 100x compared with a full checking for all intents.

Creating or updating network intents is an essential component in Intent Based Networking. We present AutoInfer, our automated network intent inference tool for running networks. AutoInfer first analyzes network configurations to extract all potential intents and then utilizes an adaptive refinement scheme to conduct runtime monitoring. The output of AutoInfer are network intent graphs for the runtime states. Experimental results based on realistic benchmarks show that AutoInfer achieves higher inference accuracy compared with existing solutions and is suitable for online use.

Bibliography

[1] 2015 data Breach Investigations Report. http://www.verizonenterprise.com/DBIR/2015/.

[2] Amazon Virtual Private Cloud. https://aws.amazon.com/vpc/?nc1=h_ls.

[3] Amazon Web Service (AWS). https://aws.amazon.com/.

[4] Apstra Operating System. https://www.apstra.com/products/.

[5] Aruba ClearPass Policy Manager. https://www.arubanetworks.com/products/security/network-access-control/.

[6] AT&T fined $ 25 million after call center employees stole customers. http://arstechnica.com/techpolicy/2015/04/att-fined-25-million-after-call-centeremployees-stole\-customers-data/.

[7] Chronology of data breaches. http://www.privacyrights.org/data-breach.

[8] Google Cloud Platform. https://cloud.google.com/.

[9] Gurobi. http://www.gurobi.com/.

[10] Haproxy-the reliable, high-performance tcp/http load balancer. http://haproxy.

[11] Hassel-C. http://bitbucket.org/peymank/hassel-public/.

[12] Header Space Library and Netplumber. http://bitbucket.org/peymank/hassel-public/.

[13] Intent based networking. https://www.cisco.com/c/en/us/solutions/intent-based-networking.html.

[14] The internet2 observatory data collections. http://www.internet2.edu/observatory/archive/data-collections.html.

[15] Microsoft Azure Cloud Computing Platform & Services. https://azure.microsoft.com/en-us/.

[16] Network complexity, change, and human factors are failing the business. https://www.veriflow.net/wp-content/uploads/2016/11/VRFL0926-Veriflow-Survey-PPT-FINAL.pdf.

[17] Open vSwitch. https://www.openvswitch.org/.

[18] Opendaylight network intent composition (nic) graph implementation. https://tinyurl.com/gld2qzn.

[19] P4. https://p4.org/.

[20] Palo Alto Networks. https://www.paloaltonetworks.com/products/secure-the-network/next-generation-firewall.

[21] Pktgen. http://pktgen.readthedocs.io/en/latest/.

[22] Radioshack sells customer data after settling with states. http://www.bloomberg.com/news/articles/2015-05-20/radioshackreceives-approval-to-sell-nameto-standard-general.

[23] Service Function Chaining. https://datatracker.ietf.org/wg/sfc/documents/.

[24] Software-Defined Networking. https://www.cisco.com/c/en/us/solutions/software-defined-networking/overview.html.

[25] understand policy routing. https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/10116-36.html.

[26] University of oregon route views project. http://www.routeviews.org.

[27] Veriflow network verification tools. https://www.veriflow.net/.

[28] WAN Optimization as-a-Service. http://www.routeviews.org.

[29] Zscaler Cloud Firewall. https://www.zscaler.com/products/next-generation-firewall.

[30] Anubhavnidhi Abhashkumar, Joon-Myung Kang, Sujata Banerjee, Aditya Akella, Ying Zhang, and Wenfei Wu. Supporting diverse dynamic intent-based policies using janus. In *Proc. of ACM CoNEXT*, 2017.

[31] Ehab Al-Shaer and Saeed Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In *Proc. of ACM SafeConfig*, 2010.

[32] Ehab Al-Shaer, Will Marrero, Adel El-Atawy, and Khalid Elbadawi. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *Proc. of IEEE ICNP*, 2009.

[33] Hassan Jameel Asghar, Luca Melis, Cyril Soldani, Emiliano De Cristofaro, Mohamed Ali Kaafar, and Laurent Mathy. Splitbox: Toward efficient private network function virtualization. In *Proc. of ACM HotMiddlebox*, 2016.

[34] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Antony Rowstron. Towards predictable datacenter networks. In *Proc. of ACM SIGCOMM*, 2011.

[35] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proc. of ACM SIGCOMM*, 2017.

[36] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proc. of ACM SIGCOMM*, 2016.

[37] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *Proc. of ACM PLDI*, 2017.

[38] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. of ACM IMC*, 2010.

[39] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Net2text: Query-guided summarization of network forwarding behaviors. In *Proc. of USENIX NSDI*, 2018.

[40] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF formulas on ciphertexts. In *Theory of cryptography*, pages 325–341. 2005.

[41] Anat Bremler-Barr, Yotam Harchol, and David Hay. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *Proc. of ACM SIGCOMM*, 2016.

[42] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[43] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[44] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[45] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *Proc. of USENIX NSDI*, 2016.

[46] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proc. of ACM CoNEXT*. ACM, 2014.

[47] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *Proc. of USENIX OSDI*, 2016.

[48] Seyed Kaveh Fayazbakhsh, Michael K Reiter, and Vyas Sekar. Verifiable network function outsourcing: requirements, challenges, and roadmap. In *Proc. of ACM HotMiddlebox*, 2013.

[49] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *Proc. of USENIX NSDI*, 2015.

[50] Jazib Frahim and Omar Santos. *Cisco ASA: All-in-One Firewall, IPS, Anti-X, and VPN Adaptive Security Appliance*. Pearson Education, 2009.

[51] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. Automatically repairing network control planes using an abstract representation. In *Proc. of ACM SOSP*, 2017.

[52] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proc. of ACM SIGCOMM*, 2016.

[53] G. Gibb, H. Zeng, and N. McKeown. Outsourcing network functionality. In *Proc. of ACM HotSDN*, 2012.

[54] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 350–361. ACM, 2011.

[55] Oded Goldreich. Foundation of cryptography: volumes I Basic tools, 2001.

[56] Michael T Goodrich and Roberto Tamassia. *Introduction to computer security*. Pearson, 2011.

[57] R. Guerzoni et al. Network functions virtualisation: an introduction, benefits, enablers, challenges and call for action, introductory white paper. In *SDN and OpenFlow World Congress*, 2012.

[58] Victor Heorhiadi, Michael K Reiter, and Vyas Sekar. Simplifying software-defined network optimization using {SOL}. In *Proc. of USENIX NSDI*, 2016.

[59] T. Inoue, T. Mano, K. Mizutani, S. Minato, and O. Akashi. Rethinking packet classification for global network view of software-defined networking. In *Proc. of IEEE ICNP*, 2014.

[60] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *Proc. of ACM SIGCOMM*, 2013.

[61] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proc. of ACM IMC*, 2009.

[62] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Proc. of USENIX NSDI*, 2013.

[63] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proc. of USENIX NSDI*, 2012.

[64] A. R. Khakpour and A. X. Liu. Quantifying and querying network reachability. In *Proc. of IEEE ICDCS*, 2010.

[65] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proc. of USENIX NSDI*, 2013.

[66] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *Proc. of USENIX NSDI*, 2015.

[67] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 2011.

[68] Eddie Kohler. *The Click Modular Router*. PhD thesis, Massachusetts Institute of Technology, 2000.

[69] Bikash Koley. The zero touch network. `https://research.google.com/pubs/pub45687.html`, 2016.

[70] M. Kuzniar, P. Peresini, and D. Kostic. What you need to know about SDN flow tables. In *Proc. of PAM*, 2015.

[71] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *Proc. of USENIX NSDI*, 2016.

[72] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proc. of ACM HotNets*, 2010.

[73] Xin Li and Chen Qian. Traffic and failure aware vm placement for multi-tenant cloud computing. In *Proc. of IEEE IWQoS*, 2015.

[74] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proc. of ACM SIGCOMM*, 2016.

[75] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *Proc. of USENIX NSDI*, 2015.

[76] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *Proc. of ACM SIGCOMM*, 2011.

[77] Pasquale Malacaria and Jonathan Heusser. Information theory and security: Quantitative information flow. In *Formal Methods for Quantitative Aspects of Programming Languages*, pages 87–134. Springer, 2010.

[78] Sharad Malik and Lintao Zhang. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.

[79] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proc. of USENIX NSDI*, 2014.

[80] Rick McGeer. Verification of switching network properties using satisfiability. In *Proc. of IEEE ICC*, 2012.

[81] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *Proc. of ACM HASP*, 2013.

[82] Luca Melis, Hassan Jameel Asghar, Emiliano De Cristofano, and Mohamed Ali Kaafar. Private processing of outsourced network functions: Feasibility and constructions. In *Proc. of ACM SDN-NFV Security*, 2016.

[83] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proc. of ACM SIGCOMM*, 2016.

[84] OpenConfig. Vendor-neutral, model-driven network management designed by users. `http://openconfig.net`. Online; accessed 22 January 2018.

[85] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Proc. of ESA*, 2001.

[86] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: a framework for NFV applications. In *Proc. of ACM SOSP*, 2015.

[87] Palo Alto Networks. Palo Alto Networks next-generation firewalls. `https://www.paloaltonetworks.com/`. Online; accessed 22 January 2018.

[88] Aurojit Panda, Ori Lahav, Katerina J Argyraki, Mooly Sagiv, and Scott Shenker. Verifying reachability in networks with mutable datapaths. In *Proc. of USENIX NSDI*, 2017.

[89] Helder Pereira, André Ribeiro, and Paulo Carvalho. L7 classification and policing in the pfsense platform. *Atas da CRC*, 2009.

[90] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *Proc. of USENIX NSDI*, 2018.

[91] Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle: a field study of middlebox failures in datacenters. In *Proc. of ACM IMC*, 2013.

[92] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *Proc. of ACM SIGCOMM*, 2015.

[93] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM Computer Communication Review*, 2015.

[94] Gregor N Purdy. Linux iptables-pocket reference: firewalls. *NAT and accounting*, 2004.

[95] Gregor N Purdy. *Linux iptables Pocket Reference: Firewalls, NAT & Accounting*. 2004.

[96] Z. A. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using SDN. In *Proc. of ACM SIGCOMM*, 2013.

[97] Michel Raynal and Mukesh Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, 1996.

[98] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proc. of ACM SIGCOMM*, 2012.

[99] Gábor Rétvári, János Tapolcai, Attila Kőrösi, András Majdán, and Zalán Heszberger. Compressing ip forwarding tables: towards entropy bounds and beyond. In *Proc. of ACM SIGCOMM*, 2013, extended version in *IEEE/ACM Transactions on Networking*.

[100] André Ribeiro and Helder Pereira. L7 classification and policing in the pfsense platform. In *21st International Teletraffic Congress (ITC 21), Paris, France*, 2009.

[101] Vyas Sekar, Michael K Reiter, and Hui Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *Proc. of ACM IMC*, 2010.

[102] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. In *Proc. of ACM SIGCOMM*, 2012.

[103] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. In *Proc. of ACM SIGCOMM*, 2012.

[104] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proc. of ACM SIGCOMM*, 2015.

[105] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *Proc. of ACM CoNEXT*, 2014.

[106] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Scalable symbolic execution for modern networks. In *Proc. of ACM SIGCOMM*, 2016.

[107] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *Proc.of USENIX NSDI*, 2018.

[108] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. Stroboscope: Declarative network monitoring on a budget. In *Proc. of USENIX NSDI*, 2018.

[109] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Proc. of ACM SOSR*, 2018.

[110] Brendan Tschaen, Ying Zhang, Theo Benson, Sujata Banerjee, Jeongkeun Lee, and Joon-Myung Kang. Sfc-checker: Checking the correct forwarding behavior of service function chaining. In *Proc. of IEEE NFV-SDN*, 2016.

[111] A. Vahidi. Jdd, a pure java bdd and z-bdd library. `http://javaddlib.sourceforge.net/jdd/index.html`, 2004.

[112] Huazhe Wang, Chen Qian, Ye Yu, Hongkun Yang, and Simon S Lam. Practical Network-wide Packet Behavior Identification by AP Classifier. In *Proc. of ACM CoNEXT*, 2015.

[113] Wenfei Wu and Ying Zhang. Network function modeling and its applications. *IEEE Internet Computing*, (4):82–86, 2017.

[114] Wenfei Wu, Ying Zhang, and Sujata Banerjee. Automatic synthesis of nf models by program analysis. In *Proc. of ACM HotNets*, 2016.

[115] G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of IP networks. In *Proc. of IEEE INFOCOM*, 2005.

[116] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, 2015.

[117] Hongkun Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. Technical Report TR-13-15, The Univ. of Texas at Austin, Dept. of Computer Science, Aug. 2013.

[118] Hongkun Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *Proc. of IEEE ICNP*, 2013, extended version in *IEEE/ACM Transactions on Networking*.

[119] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *Proc. of USENIX NSDI*, 2013.

[120] Ye Yu, Chen Qian, and Xin Li. Distributed collaborative monitoring in software defined networks. In *Proc. of ACM HotSDN*, 2014.

[121] Xingliang Yuan, Huayi Duan, and Cong Wang. Bringing practical execution assurances to outsourced middleboxes. In *Proc. of IEEE ICNP*, 2016.

[122] Xingliang Yuan, Xinyu Wang, Jianxiong Lin, and Cong Wang. Privacy-preserving deep packet inspection in outsourced middleboxes. In *Proc. of IEEE INFOCOM*, 2016.

[123] Hongyi Zeng, Peyman Kazemiany, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proc. of ACM CoNEXT*, 2012.

[124] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, and Carsten Lund. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. In *Proc. of ACM IMC*, 2004.

[125] Ying Zhang, Neda Beheshti, Ludovic Beliveau, Geoffrey Lefebvre, Ramesh Mishra, Ritun Patney, Erik Rubow, Ramesh Subrahmaniam, Ravi Manghirmalani, Meral Shirazipour, Catherine Truchan, and Mallik Tatipamula. StEERING: A Software-Defined Networking for Inline Service Chaining. In *Proc. of IEEE ICNP*, 2013.

[126] Ying Zhang, Wenfei Wu, Sujata Banerjee, Joon-Myung Kang, and Mario A Sanchez. Sla-verifier: Stateful and quantitative verification for service chaining. In *Proc. of IEEE INFOCOM*, 2017.

[127] Yu Zhao, Huazhe Wang, Xin Lin, Tingting Yu, and Chen Qian. Pronto: Efficient test packet generation for dynamic network data planes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 13–22. IEEE, 2017.