FLOW TABLE MANAGEMENT IN PROGRAMMABLE NETWORK DATA PLANES

A Dissertation

by

LUKE ANDREW MCHALE

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Paul V. Gratz
Co-Chair of Committee,	Alex Sprintson
Committee Members,	Riccardo Bettati
	Krishna Narayanan
	Stavros Kalafatis
Head of Department,	Miroslav M. Begovic

December 2022

Major Subject: Computer Engineering

ABSTRACT

The design-space of network devices is constantly evolving, driven by the continual demand for increased global inter-connectivity, intelligent orchestration, and distributed computation between cloud and edge resources. Modern businesses are increasingly reliant on a connected world for a competitive advantage as well as essential operations. Meanwhile, there is an increasing number of attacks on critical communication infrastructure from a variety of malicious actors. Thus, there is an increasing urgency to improve all aspects of security in data communication networks.

Additionally, Software-Defined Networking (SDN) has increasingly gained traction and utility across data centers and network administration. SDN concepts enable increased flexibility for network operators, including the ability to implement a broad class of custom network functions for real-time diagnostics as well as traffic management. While SDN has notable advantages over traditional network appliances, current implementations are often more susceptible to malicious attacks due to increased complexity and abstractions imposed on packet classification and table management.

This dissertation investigates architectural techniques to improve the reliability and performance of data plane processing hardware. Our techniques are applicable to both traditional packet processing as well as SDN data plane architectures. The contributions of this research include two novel and complementary techniques to improve data plane performance through optimizing the use of available packet classification resources. By leveraging storage-efficient stochastic data structures and machine learning inspired replacement policies, our techniques improve data plane processing efficiency and predictability.

The first technique leverages a Bloom Filter to prioritize established traffic and prevent malicious starvation of expensive packet classification resources. This *Pre-Classification* technique is general enough to be considered for any classification pipeline with non-uniform processing requirements. The second technique, originally developed for speculative microprocessors, adapts the Hashed Perceptron binary classifier to flow table cache management. The proposed *Flow*

Correlator mechanism leverages the Hashed Perceptron to correlate flow activity with temporal patterns and transport/network layer hints. This technique demonstrates the viability of associating temporal patterns to network flows enabling improvements in flow table cache management. Amenable to hardware implementations, both *Flow Correlator* and *Pre-Classification* techniques show promise in improving the reliability and performance of flow-centric packet processing architectures.

ACKNOWLEDGEMENTS

First and foremost, I want to thank my advisors Paul Gratz and Alex Sprintson for their dedication to my personal as well as technical growth. I especially want to thank Paul for his patience, support, and guidance throughout my graduate research. I particularly want to thank Alex for his encouragement and foresight, always enabling opportunities from teaching fellowships to research collaborations. Without the guidance and support of both Paul and Alex, this dissertation would not have been possible. I also would like to sincerely thank my committee members for their feedback and insights throughout my graduate studies.

I also sincerely appreciate the friendship of Andrew Targhetta, Braden Obrzut, Joseph Bozzay, Eliot Edling, Matthew Hamer, Michael Starr, Ping Wang, Gino Chacon, and Priya Madhu – providing notable encouragement as well as essential breaks between the many hours of research. Finally, I especially would like to thank my parents Timothy and Nancy as well as my sisters Tabitha and KayLeigh for their support and encouragement throughout this journey.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Paul V. Gratz (chair), Alex Sprintson (co-chair), Krishna Narayanan, and Stavros Kalafatis of Texas A&M's Department of Electrical and Computer Engineering as well as Professor Riccardo Bettati of Texas A&M's Department of Computer Science and Engineering.

Chapter 2 was a collaboration with Jasson Casey, including assistance from Eric Garfinkle. Discussions with Boris Hanin, Andrew Sutton, and Jasson Casey provided insight which helped scope the research direction of Chapter 3. Prior works from Andrew Sutton, Michael Gruesen, and Hoang Nguyen were leveraged during this research. Passive network datasets provided by Equinix and CAIDA [1] enabled the ability to perform this research. All research conducted for this dissertation was completed by the student.

Funding Sources

This work was funded in part through National Science Foundation grant CNS-1423322[1] as well as U.S. Air Force Research Lab grants FA8650-05-D-1912[2] and FA8650-13-C-5800[3].

This work was also partially funded by fellowships as well as research and teaching assistantships provided by Texas A&M's Department of Electrical and Computer Engineering including the 2012 B. Pat and Frieda Ebsenberger Doctoral Fellowship, 2018 Graduate Teaching Fellowship, 2020 Summer Research Fellowship, as well as network equipment gifts provided by Freescale Semiconductor in 2015.

[1] "Tools for Design and Analysis of Provably Correct Networking Systems", National Science Foundation, PI: A. Sprintson, Co-PIs: G. Dos-Reis, A. Sutton, Oct 2014 - Sep 2018, award number CNS-1423322.

[2] "Minority Leaders Program: Cyber-Security Research for Distributed Sensory Systems and Cloud Applications", U.S. Air Force Research Lab, Clarkson Aerospace, PI: Dr. Sejun Song, Co-PI: Dr. Alex Sprintson, Nov 2012 - Feb 2014, award number FA8650-05-D-1912.

[3] "AFRL Collaboration Program: Sensors Research: Conformance Verification and Software-Defined Flow Control Frameworks in Cloud Networks and High Power Waveguide Amplifiers for EO", U.S. Air Force Research Lab, Clarkson Aerospace, PI: Alex Sprintson, Co-PI: Guofei Gu, Oct 2014 - May 2018, award number FA8650-13-C-5800.

NOMENCLATURE

ACL	Access Control List
AMAT	Average Memory Access Time
ASIC	Application-Specific Integrated Circuit
CDF	Cumulative Distribution Function
DoS	Denial-of-Service
DDoS	Distributed Denial-of-Service
HP	Hashed Perceptron
IG	Information Gain
IP	Internet Protocol
IXP	Internet Exchange Point
LRU	Least Recently Used
MCC	Matthew's Correlation Coefficient
ML	Machine Learning
MRU	Most Recently Used
MTU	Maximum Transmission Unit
NFV	Network Function Virtualization
PCAP	Packet Capture
PCC	Pearson's Correlation Coefficient
RTT	Round Trip Time
SDN	Software Defined Networking
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

The design-space of network devices is constantly evolving, driven by the continual demand for increased global inter-connectivity, intelligent orchestration, and distribution of computation between cloud and edge resources. Modern businesses are increasingly reliant on a connected world for a competitive advantage as well as essential operations. Additionally, there is an increasing urgency to improve the security of critical infrastructure, emphasizing the importance of reliable devices and ultimately resilient networks.

1.1 Packet Processing Architectures

Researchers and industry continue to push the boundaries of packet processing performance and device reconfigurability. Trade-offs in performance, flexibility, and programmability are being explored with the aim of improving efficiency, reliability, and overall quality of networks. Networking hardware and software is notably complex, accompanied a high barrier to entry. It is also difficult, while notably important to guarantee performance across diverse network scenarios.

Packet processing devices and frameworks trade-off programmability with target performance, ranging from networking ASICs [2, 3], network processors [4], to general purpose systems [5]. A common architectural advantage of network processors is to provide hardware-assisted event scheduling and table lookup extensions for both on-chip and memory-backed tables. Many network functions require large flow tables, often consisting of millions of entries and managed across both on-chip and off-chip table resources [6]. To meet performance and cost requirements, caching mechanisms are often employed. Caching a flow table using on-chip table resources preserve memory bandwidth and reduce latency, but introduces the complexity of effective cache management.

1.2 Software Defined Networking

Software Defined Networking (SDN) is an exploration into the formal separation of control plane and data plane constructs. Motivations include commoditization of data plane hardware (commonly referred to as whitebox switches), reconfigurable flexibility, and longer device life-

cycles through software updates and a third party network function application ecosystem. There has been simultaneous resistance and desire to integrate SDN capabilities into networking devices [7]. OpenFlow [8] is a recent effort towards standardizing abstractions and control protocols for data plane manipulation. Abstractions introduced by this distributed model require careful engineering to reduce associated overheads [9].

SDN has demonstrated utility, particularly as a diagnostic tool for complex network operations [10]. The complexities inherent in customizing data plane processing is out of reach to all but the largest enterprises. The complexity of the constructing SDN systems is notably higher than that of the traditional systems [11]. In particular, it is difficult to verify data plane modification do not result in unintended consequences [12, 13]. In practice, data plane programmability can introduce data paths that violate protocol definitions and various expectation across networking standards [14, 15, 16]. Moreover, SDN applications present unique safety challenges above and beyond traditional programming language libraries and run-times [17, 18, 19, 20].

1.3 Packet Classification

Packet classification is a fundamental component of networking hardware, generally growing in complexity with network size [21]. It is well known that packet classification is expensive and this is a well studied problem in the traditional network hardware domain [22, 23, 24, 25]. Complexity of packet classification remains a processing bottleneck, especially in the context of SDN [26, 27].

Networking devices have a long history of highly specialized designs along with notable service-based market segmentation. Partly due to the complexity of underlying protocols, but also intricacies involved with ensuring a device meets reliability expectations. Researchers and enterprises have long been experimenting with trade-offs of increasing the exposed programmability of networking devices. Specialized applications will continue to demand the ability to categorize, filter, and manipulate flows in increasingly complex ways.

As network reachability and uptime becomes increasingly crucial to modern infrastructure and safety, it is necessary to develop architectural techniques to reduce the impact of Denial-of-Service

(DoS). Not limited to hosted services, underlying network infrastructure is also often vulnerable to DoS [28].

Non-uniform processing requirements are a prime target for DoS. Packet classification is inherently complex with variable processing requirements. Worse yet, Access Control Lists (ACLs) tend to be focused around allow lists with unauthorized traffic tending to require full ACL traversal, opening the door for DoS attacks on classification. Accordingly there is a critical need to explore architectures that can improve the resilience of networks.

Switching and routing functions are a backbone of all networks; however, an increasing number of network functionality requires tracking flow state for transiting connections. Modern stateful firewalls, security monitoring devices, and corporate edge services rely on efficient implementations of stateful flow tables. These flow tables often grow much larger than can be expected to fit within on-chip memory, requiring a managed caching layer to meet capacity and performance requirements [29]. Depending on the network function and performance target, the flow table and caching layers may be distributed across both hardware and software components. Complicating observability, this caching layer is often obfuscated and embedded between the logical control and data plane layers [30].

1.4 Dissertation Statement

Hardware techniques leveraging temporal patterns can be used to improve the performance and reliability of packet processing architectures.

1.5 Contributions

In this dissertation we examine how stochastic and speculative hardware techniques can be used to further leverage temporal locality in order to improve performance over existing networking hardware. The goal of this research is to improve the performance and reliability of packet processing systems through architectural techniques.

The contributions of this dissertation are summarized as follows:

i. Novel use of a stochastic data structure to decouple the impact of malicious packets on estab-

lished flows.

ii. Design and evaluation of a DDoS mitigation strategy to protect classification pipelines with non-uniform execution paths.

iii. Flow table cache management limit study using Belady's MIN on CAIDA network exchange traffic.

iv. Application of the Hashed Perceptron binary classifier to network flow table cache management.

v. Iterative approach to feature selection and ranking in the context of Hashed Perceptron binary classifiers.

vi. Discussion of feature roles, granting further perspective into the dynamics of a Hashed Perceptron binary classifier.

1.6 Dissertation Organization

Chapter 2 introduces *Stochastic Pre-Classification*, an architectural technique to harden network functions against Distributed Denial-of-Service (DDoS). Chapter 3 advances stateful flow table cache management, introducing *Flow Correlator*, a feature-driven temporal flow activity prediction mechanism. Finally, Chapter 4 wraps up this dissertation reviewing conclusions and notable future work.

2. STOCHASTIC PRE-CLASSIFICATION*

2.1 Introduction

Packet classification is a fundamental component of network hardware. Generally, the problem of packet classification expands in complexity as the number of rules grow. For example, typical firewalls have access control lists (ACLs) with thousands of matching rules. It is well known that packet classification is expensive and this is a well studied problem in the traditional network hardware domain [23, 24, 22]. With the move to Software Defined Networking (SDN), the complexity of packet classification is expected to grow dramatically due to the increased number of matching fields, the push to support a large number of features, and the larger degree of flexibility that SDNs encompass. While brute force hardware approaches, such as techniques that leverage ternary content addressable memories (TCAMs), can be used to improve the throughput [2], these approaches incur significant costs and result in increased power consumption.

Increasing packet classification complexity in turn increases data planes' vulnerability to Denial of Service (DoS) and, in particular, Distributed Denial of Service (DDoS) attacks. Malicious packet flows, such as those seen during a DDoS attack, interfere with authorized flows by consuming valuable data plane resources. Accordingly there is a critical need to explore architectures that can decouple the impact of potentially low throughput malicious traffic from high throughput authorized traffic in SDN data planes. This malicious traffic may consist of multiple well coordinated flows that come from a potentially large number of sources and can cause significant disruption (in terms of latency and packet loss) of the authorized traffic.

The goal of this chapter is to improve the throughput and latency of packet classification for known/authorized traffic within the SDN data plane. The main idea is to pre-classify known authorized traffic in SDN data planes, separating them from unknown or malicious traffic, thereby reducing the impact of malicious traffic on known flows.

*©2014 IEEE. Reprinted, with permission from, L. McHale, J. Casey, P. V. Gratz, and A. Sprintson, "Stochastic Pre-classification for SDN Data Plane Matching", 2014 IEEE 22nd International Conference on Network Protocols, October 2014 [31].

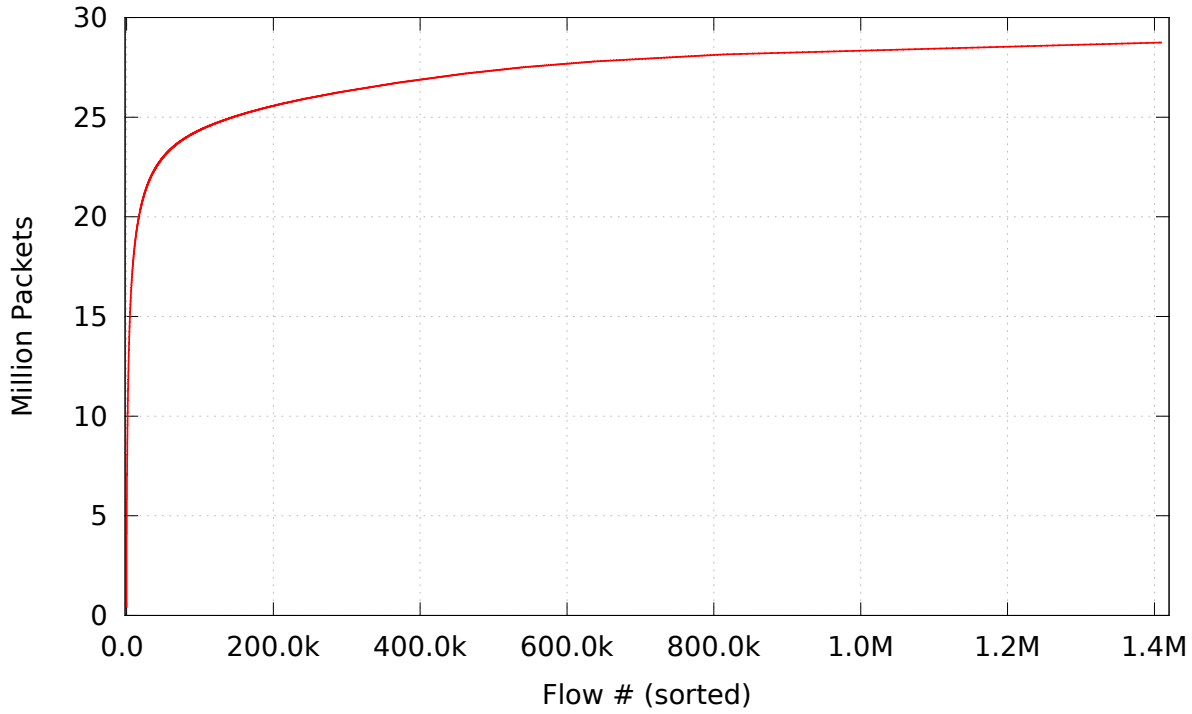


Figure 2.1: Cumulative Distribution of Packets per Flow*

Sorted CDF of packets (y-axis) associated with each flow (x-axis) for CAIDA trace equinix-sanjose.dirA.20120119-125903.

*©2014 IEEE. Reprinted with permission from L. McHale, J. Casey, P. V. Gratz, and A. Sprintson [31].

As a means to accelerate packet classification, we propose to leverage the locality of known, authorized flows to enable a pre-classification stage within the SDN data plane. For example, Figure 2.1 is a Cumulative Distribution Function (CDF) of packets per flow from a one minute CAIDA trace [32]. The CDF shows a non-uniform distribution of packets amongst flows, with a small subset responsible for a majority of the traffic. Specifically, just 3% of flows are responsible for approximately 80% of packets observed. Additionally, in just 35% of flows, 95% of all packets are covered; meanwhile, the remaining 65% of flows contribute to only 5% of the observed packets.

There is significant locality to be leveraged in packet classification. Some traditional approaches to accelerating packet classification, such as ACL caching [21, 33, 23] leverage this locality, however these approaches are truly able to decouple the effects of highly defuse mali-

cious traffic as seen under DDoS attacks. To address these challenges, we propose to use a Bloom filter [34] as a hardware pre-classification stage, improving on a prior software implementation [35].

Our goal in this work to enable a decoupling of known authorized traffic from unknown and/or malicious traffic within the SDN data plane. The contributions of this chapter are as follows:

i. Novel use of a stochastic data structure to decouple the impact of malicious packets on established flows.

ii. A combined architectural approach that protects established flows from DoS/DDoS.

iii. Design and evaluation of a DDoS mitigation strategy to protect classification pipelines with non-uniform execution paths.

iv. A simulation study to evaluate the dynamics of an SDN data plane under attack.

The remainder of the chapter is organized as follows: Section 2.2 discusses the background on hardware data planes for use in SDNs. It also examines prior work in packet classification. Section 2.3 introduces our proposed hardware design. Section 2.4 evaluates our design for its ability to achieve our goal of decoupling malicious traffic's effect on known good traffic. Finally, in Section 2.5 we present conclusions and directions for future research.

2.2 Background

In this work we start with a baseline OpenFlow data plane architecture and explore design permutations focused on improving overall data plane performance while under heavy and potentially malicious traffic. The baseline OpenFlow architecture and protocols are maintained by the Open Networking Foundation [8]. These specifications outline semantics for an abstract packet processing machine. While these documents are not precise, they have been successful in outlining a basic packet processing pipeline along with a control interface for manipulating the pipeline's state.

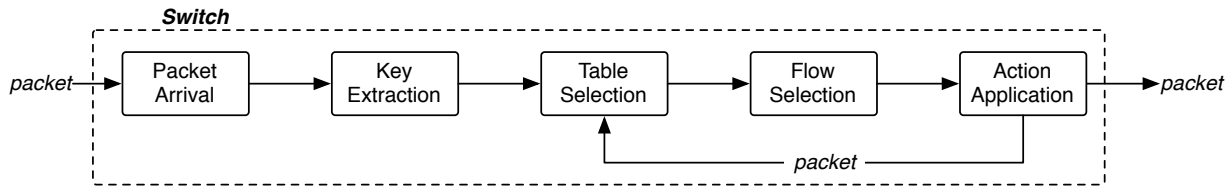


Figure 2.2: OpenFlow Data Plane*

*©2014 IEEE. Reprinted with permission from L. McHale, J. Casey, P. V. Gratz, and A. Sprintson [31].

2.2.0.1 OpenFlow

OpenFlow describes a simplified data plane for packet processing, this data plane pipeline is pictured in Figure 2.2. Packet processing happens in five stages. The first stage, packet arrival, is concerned with writing a packet to memory and sending certain bits of metadata about the packet into the pipeline such as: address in memory, packet size in bytes, arrival port id, etc. The second stage involves decoding enough of the packet's header to construct a key. The packet key is a tuple formed from a subset of packet header fields. The third stage, table selection, selects the appropriate flow table for indexing the packet's key. The first time a packet transits the pipeline, it always selects the first flow table; however, subsequent traversals can choose different tables. The fourth stage, flow selection, will use the packet's key to choose a specific entry in the flow table. The entry will contain a policy, or set of actions, to be applied to all matching packets. Finally, the fifth stage applies the selected policy to the packet. This could result in the modification of data within the packet, copying the packet, traversing the pipeline again, directing the packet towards the controller, or egressing the system.

2.2.1 Packet Classification

Packet classification is a fundamental activity in the core of any packet processing data plane. The basic process of classification involves forming a key that represents a packet and finding an entry in a classifier table that matches. The key is formed by extracting a set of values from a packet's protocol header fields. These values are concatenated to form a bit string that represents

the packet. Classifier tables usually match the key bit string against a pair of bit strings, where the first element of the pair is a value to be matched and the second element masks bits that are important to the table entry.

There are several techniques for addressing matching problems: hash tables, tries, hierarchical tries, etc. [23, 36], but once you move to two or more dimension prefix bit string matching solutions become expensive in terms of time or memory. Varghese [36] provides a comprehensive survey of the classification problems as well as approaches for their solution. For our discussion we are interested in software and low cost hardware solutions, which are likely constraints on low cost pervasive networking devices supporting SDN protocols. To further complicate the problem certain types of attack traffic can drastically reduce the performance of the classification stage in a data plane pipeline. The simple reason is that DoS/DDoS traffic cannot be classified as malicious until the classification activities have completed. Attackers have shown their ability to generate large volumes of attack traffic, in excess of 300 Gbps, congesting major Internet links [28]. Performing multidimensional classification in software is expensive, therefore it would be highly desirable to prevent malicious traffic from consuming classification resources.

2.3 Design

In this section, we describe a phenomenon inherent in network traffic. We further suggest two techniques that provide unique advantages to classification.

2.3.1 Motivation

Network packet classification is difficult, in part, due to the inherit randomness in packet arrival. In order to ensure no packets are discarded during classification, the network node must be able to process all packets at line rate. However, there are many circumstances where this is not possible due to classification complexity, throughput requirements, and available computational capability.

2.3.1.1 Pre-Classification

Our goal is to treat classification as a finite resource and prioritize packets for classification based on whether or not the packet belongs to a known, trusted flow. By partitioning incoming

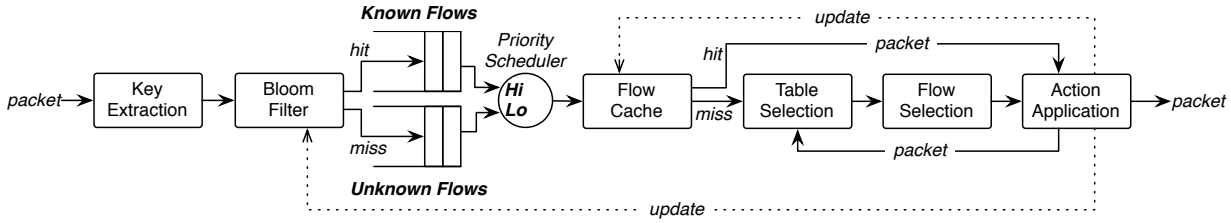


Figure 2.3: Resilient Classification Data Plane*

*©2014 IEEE. Reprinted with permission from L. McHale, J. Casey, P. V. Gratz, and A. Sprintson [31].

packets thus, we isolate the consumption of classification resources from malicious traffic. We propose to place a pre-classification stage before flow selection in the SDN data plane. The aim of this pre-classification stage is to leverage the locality in known, authorized flows in order to isolate performance of known flows from adverse effects caused by large bursts of unknown traffic.

2.3.1.2 Flow Locality

Within a period of time, old flows retire and new flows are established. The number of active flows, while highly variable, is finite and at worst equal to the number of packets. However, the number of flows is often much less than the total number of packets. This trend can be observed in Figure 2.1. Over this particular trace, there exists 1.4 million unique flows contained within 27.3 million packets – an order of magnitude difference in flow count versus packet count. When looking at received bytes, 95% of the aggregate throughput occurs from just 35% of the 1.4 million flows. We refer to this phenomenon as *flow locality*. We further define the *active-flow window* as a measure of flow locality over a given period of time. In this work we leverage flow locality within the given active-flow window to improve classification throughput via active flow action caching.

We propose to leverage both pre-classification and flow locality in our proposed design. As we will show in our results, techniques that take advantage of these properties provide orthogonal benefit and their effect is additive.

2.3.2 Architecture

Figure 2.3 depicts the block diagram of our general architecture for an OpenFlow data plane which leverages both pre-classification and flow locality. As shown, the OpenFlow abstractions from Figure 2.2 exist in this design as well. Differing from Figure 2.2, we first note that packets are partitioned by the *Bloom Filter* into two queues: *Known Flows* and *Unknown Flows* – corresponding to the desired pre-classification traffic classes. These flows are scheduled for processing in the remainder of the data plane by the *Priority Scheduler*, with *Known Flows* always processed ahead of *Unknown Flows* (*i.e.*, packets in the *Unknown Flows* queue will wait for processing until no packets remain in the *Known Flow* queue). The final addition to this data plane is the *Flow Cache*, which exploits flow locality within the active-flow window to cache the desired actions for a given flow. The remainder of the data plane correspond to the baseline architecture of Figure 2.2. We now discuss each of the proposed components in detail.

2.3.2.1 Flow-Identifiers and the Bloom Filter

The Bloom Filter [34], a constant time stochastic containment data structure, is the critical component enabling quick and efficient pre-classification. Bloom Filters are a well-known, space-efficient, data structure which approximates the behavior of a conventional hash table for testing whether an element is within a set. In hardware, Bloom Filters store up to 10X more elements in the same space, and/or are many times faster to access than a traditional hash table. In our proposed design, the Bloom Filter serves as a space-efficient container to track flow identifiers. The Bloom Filter provides the ability to quickly and arbitrarily segregate packets, effectively decoupling flows into two logical partitions – *Known Flows* and *Unknown Flows*. Once incoming flows have been partitioned, they are fed into two different queues prior to proceeding through the rest of the data plane. The *Known Flows* queue has a higher priority than the *Unknown Flows* queue, decoupling classification performance of known flows from the effect of potentially large numbers of packets from previously unclassified, unknown flows.

In a standard SDN data plane, a unique key is extracted from the incoming packet to be used

in packet classification. Typically, this key is then directly used by the *Table Selection* and *Flow Selection* stages, shown in Figure 2.2 to classify the packet. Here, we use this key first as input to the *Bloom Filter* stage, as shown in Figure 2.4.

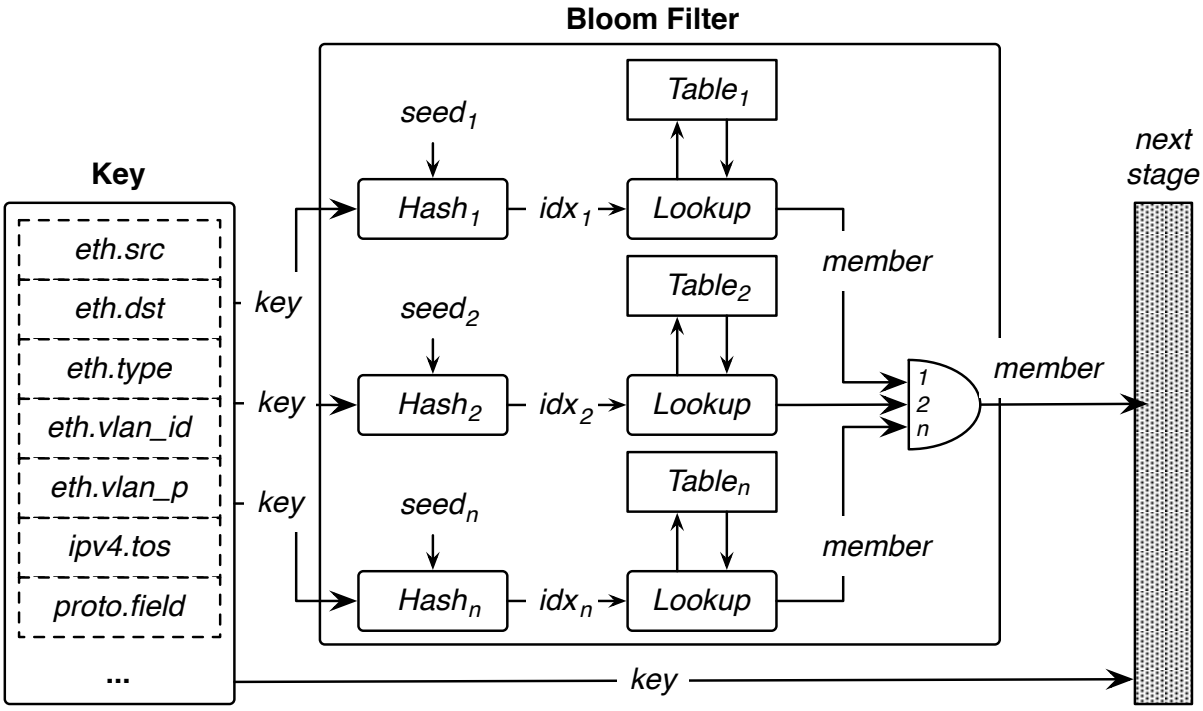


Figure 2.4: Bloom Filter Stage*

*©2014 IEEE. Reprinted with permission from L. McHale, J. Casey, P. V. Gratz, and A. Sprintson [31].

To map an arbitrary k-bit tuple (*Key*) to a w-bit flow identifier used by both the *Bloom Filter* and *Flow Cache* stages, the Key is reduced using an *XOR tree* as the hash function ($Hash_n$). Each of the n XOR trees fold the k-bit Key into a w-bit flow identifier, where w is log_2 of the Bloom Filter table size. Each w-bit flow identifier is then XOR'ed with a w-bit random number ($seed_n$) to obfuscate the hash, resulting in the Bloom Filter table index (idx_n). On every clearing interval, each seed is refreshed with a new random number.

In order to avoid n read/write ports on a single table, the Bloom Filter's table is split into n

sets. Each set is managed by a single hash function. The resulting membership is simply the logical AND of the memberships determined by each set. Even though each hash performs essentially the same operation (reducing the k-bit Key to a w-bit flow identifier), the XOR combination must be unique to reduce collisions caused by compression.

In order to design a sufficient XOR tree, the binary entropy was analyzed for each bit in the Key over the length of the trace. In general, binary entropy decreases from LSB to MSB of IP and Port fields. The payload type field offers less entropy to the Key since the bits are highly correlated due to the popularity of IPv4 and UDP protocols.. The relative entropy of each bit in the Key may vary depending on the type of network traffic. In order to increase the entropy of the resulting flow identifier, the XOR tree was constructed to avoid the chance of combining bits that are highly correlated.

We segregated the bits from the Key (sorted by measured entropy) into w-bit levels. The order of the w bits within each level was randomized for each of the n hashes at design time. Finally, each of the w flow identifier bits is the column compression (XOR) through each level. This effectively randomizes the stride and reduces the chance of combining bits that are highly correlated.

Bloom Filters are stochastic data structures which have a low, but non-zero probability of false positive matches. In our design, false positives indicate the associated packet is falsely classified as known. Too many false positives will diminish the benefits of decoupling known flows and thus can lead to some performance degradation should the occurrence be too high. The probability of false positive classification increases as more items are added to the Bloom Filter. To ensure that false positive probability remains low, our design clears the Bloom Filter after I_{CLR} insertions. After clearing, the Bloom Filter must effectively relearn the locality of flows, thus the clearing interval I_{CLR} must be infrequent enough to reduce the likelihood of cold misses, yet frequent enough to reduce the likelihood of false positives. Real-time monitoring of false positives within the data plane is possible, however we simply use a pre-determined constant clearing interval. In our prior work involving exploration of a software-implemented Bloom Filter in a Linux system [35] showed that simply cold clearing is effective, but more intelligent clearing mechanisms are

possible.

2.3.2.2 Flow Locality and the Flow Cache

As discussed in Section 2.3.1 and demonstrated in Figure 2.1, within typical packet traces there exists a high degree of flow locality. Our design leverages this flow locality with the *Flow Cache*, shown in Figure 2.3. Here, the purpose of the *Flow Cache* is to cache the actions to be performed upon a given flow, reducing the burden upon the *Table Selection* and *Flow Selection* stages of the SDN data plane. In our design, the *Flow Cache* is indexed with the same flow identifier used to index the *Bloom Filter* stage (*i.e.*, w-bit identifier, XOR'ed down from the k-bit flow key). To ensure a deterministic match, each entry in the cache contains a full k-bit key for definitive tag match against the flow in question, together with a 32-bit "action" field for storing a pointer to the action-set associated with that flow entry.

2.3.2.3 Bloom Filter Flow Learning

Here we proceed to describe the learning process for unknown flows. After key extraction, the packet is first checked against the Bloom Filter. When a miss occurs from the Bloom Filter, the packet enters the *Unknown Flows* queue. When no packets are currently in the *Known Flows* queue, the packet is then searched for in the *Flow Cache* (note, while it is unlikely that a flow would match in the *Flow Cache* after missing in the *Bloom Filter*, it is not impossible given the clearing interval I_{CLR} of the *Bloom Filter* and the *Flow Cache* replacement policy). Assuming no match in the *Flow Cache*, the packet then proceeds through the *Table Selection*, *Flow Selection*, and *Action Application* stages for packet classification and action-set application.

The update path to the Bloom Filter and Flow Cache can potentially be definable by the application. Applications could decide that certain flows should not be pre-classified using a *pre-classification* bit and/or cached using a *cacheable* bit.

Additionally, a cacheable or priority instruction could be integrated to offer more fine-grained control. Upon classification, the *Action Application* stage may prioritize the flow by inserting it into the *Bloom Filter* and/or improve classification performance by inserting it into the *Flow*

Cache. Once the flow has been prioritized, all future packets matching the flow-identifier are pre-classified are then forwarded to the *Known Flow* queue. The action execution stage may also choose to process the packet without updating the *Bloom Filter*/*Flow Cache* selectively or only after a threshold is reached. While we always update the *Bloom Filter* and *Flow Cache* for every flow in our test application, configurable behavior may be desirable for low-throughput or low-priority flows.

2.4 Evaluation

In this section we first describe our experimental methodology and design implementation details. We then examine the performance of our design for varying combinations of malicious traffic and interface speeds.

2.4.1 Methodology

All experiments presented were modeled using a cycle accurate SDN data plane simulator developed in-house. SDN data plane models were developed for the following architectures:

- **Baseline:** Basic data plane architecture shown in Figure 2.2.
- **Partition+Caching:** Proposed architecture shown in Figure 2.3.
- **Partition:** Data Plane architecture with *Bloom Filter* and *Priority Scheduler* stages, but no *Flow Cache*.
- **Caching:** Data Plane architecture with a *Flow Cache* stage, but no *Bloom Filter* and *Priority Scheduler*.

Table 2.1 shows the microarchitectural implementation details for the designs under test (except where noted elsewhere). The Bloom Filter and Flow Cache sizes and clearing interval were set at the size empirically determined to be the point of diminishing returns in performance benefit. The data plane frequency was set to be equal to the access time of the slowest memory array defined within the system as determined by the SRAM array modeling tool, Cacti [37].

Data Plane Frequency	2 GHz
Data Plane Queue Depth	2 high, 2 low
Bloom Filter Size	320Kb (5 arrays, each 64Kb)
Bloom Filter Clearing Interval (I_{CLR})	60K insertions
Flow Cache Size	69Kb (512 138-bit entries)
Flow Cache Organization	2-way set associative, LRU
Flow Selection	8,000 entries

Table 2.1: Summary of Pre-Classification Architecture*

*©2014 IEEE. Reprinted with permission from L. McHale, J. Casey, P. V. Gratz, and A. Sprintson [31].

The workload examined here consists of captured traffic through an internet core switch provided by CAIDA [32]. These traces were captured on a 10 Gbps line card with a median 3 Gbps throughput during a 60 second time window. For privacy reasons, the trace was anonymized by CAIDA. The the associated ports, protocols, and relative flows were left intact. In order to observe how each architecture scales with throughput requirements, the packet-arrival time was expanded or compressed linearly to emulate 1 Gbps, 40 Gpbs, and 100 Gbps line cards.

For flow classification, using the CAIDA trace, we synthesized a set of eight-thousand classification rules utilizing protocol, IP, and port fields for both source and destination. In these experiments, the only action was either accept or drop, emulating a basic firewall application. To generate the rules we implemented a heuristic rule generator to synthesize a set of rules (8,000 in this case) for an arbitrary PCAP trace with a target rate of authorized ($\text{Total}_{\text{auth}}$) versus unauthorized ($\text{Total}_{\text{reject}}$) traffic. The generator distributes matches evenly across all 8,000 rules (as much as possible). Rule sets were generated for the following three scenarios (labeled accordingly in the experimental figures):

- **95%** authorized traffic: representing a network in nominal conditions.
- **60%** authorized traffic: representing a network with a high ratio of unauthorized traffic.
- **20%** authorized traffic: representing a network under a DDoS attack.

2.4.2 Experimental Results

In this section, we compare the three classification architectures, *Partition*, *Caching*, and combined *Partition+Caching*, against a baseline SDN data plane configuration. We evaluate the effect of each approach on data plane performance by analyzing throughput, mean latency, and jitter (mean standard deviation of latency) for a range of interface speeds and authorized to unauthorized traffic ratios.

2.4.2.1 Classification Throughput

Figure 2.5 shows classification throughput for each architecture. Here, throughput is measured as authorized packets accepted ($\text{Accepted}_{\text{auth}}$) normalized against the total number of authorized packets ($\text{Total}_{\text{auth}}$). Throughput is normalized to indicate the ratio of authorized packets that were classified and not dropped. Normalizing allowing comparing throughput across each attack scenario.

Generally for interface speeds above 10 Gbps, the baseline classifier is no longer able to keep up with the packet arrival rate. Since baseline is indiscriminately dropping packets, the throughput is further reduced proportional to the ratio of adversarial traffic. The effect of each architecture is revealed as the classifier is further stressed with 40 Gbps.

Interestingly, we find that both architectures which contain pre-classification (*Partition* and *Partition+Cache*) actually achieve higher relative throughput when the volume of unauthorized traffic is high as opposed to nominal (*i.e.*, 95% authorized). This is as opposed to the *Baseline* and *Cache* architectures where a higher ratio of unauthorized traffic leads to worse throughput. This highlights the pre-classifier's effective quality of service toward known flows.

Generally, in the figure we see that the *Caching* architecture provides a significant boost to classification throughput over baseline. Similar to the baseline architecture, throughput is reduced proportional to adversarial traffic. By combining both components, the *Partition+Caching* architecture leverages the benefits of both pre-classification and flow locality to provide consistent classification throughput in both nominal and adversarial network conditions. In addition, the *Par-*

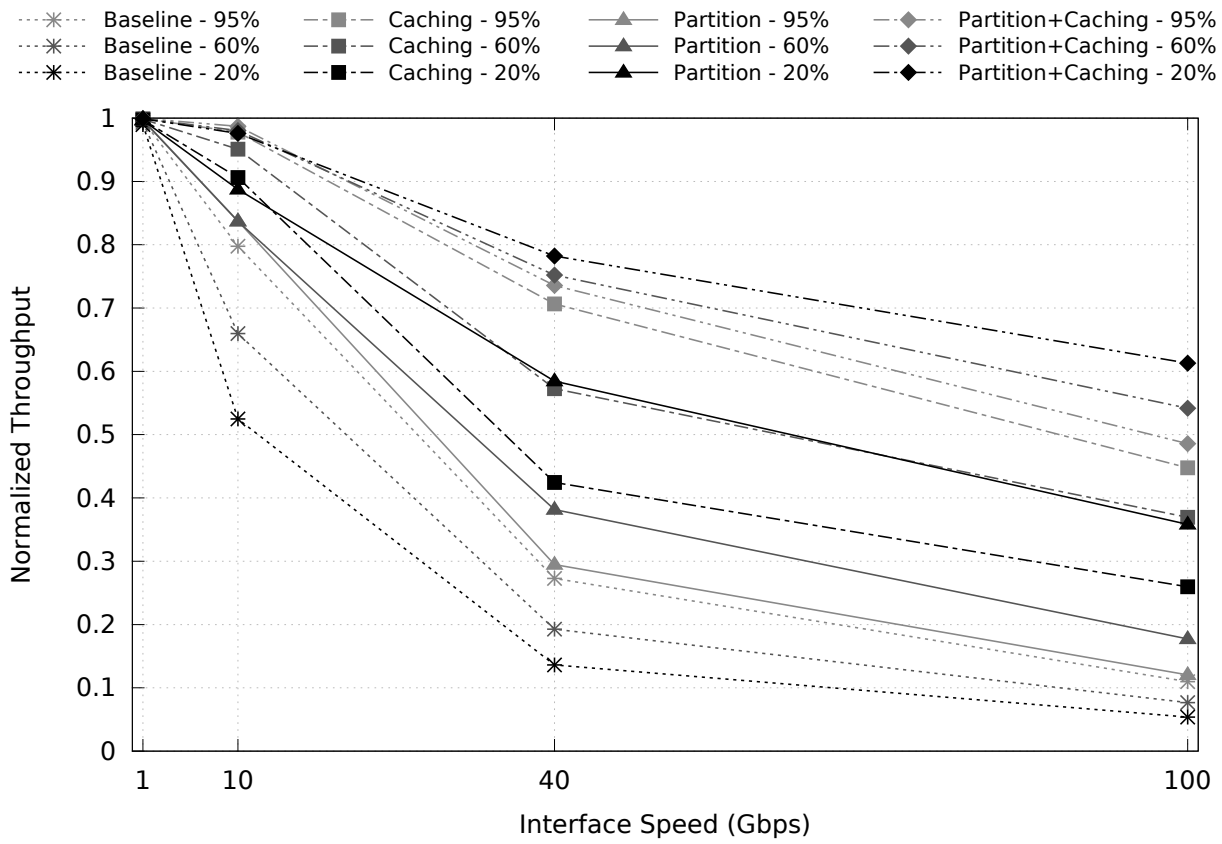


Figure 2.5: Measured Throughput*

Throughput of $\text{Accepted}_{\text{auth}}$ normalized to $\text{Total}_{\text{auth}}$ for each classification architecture.

*©2014 IEEE. Reprinted with permission from L. McHale, J. Casey, P. V. Gratz, and A. Sprintson [31].

tition+Caching architecture scales much better compared to the baseline as interface speed, or similarly, classification requirements increase.

2.4.2.2 Classification Latency

Figure 2.6 shows the mean latency through the SDN data plane for each of the architectures evaluated. In the figure we see that the latency results roughly map to the throughput results above. Generally, the data plane becomes stressed at 10 Gbps and saturated at 40 Gbps. Notice the *Partition* architecture behaves identical to the baseline when queues are rarely full (1 Gbps); however, the Partition mechanism maintains consistent behavior at 10 Gbps even when classifier resources are stressed.

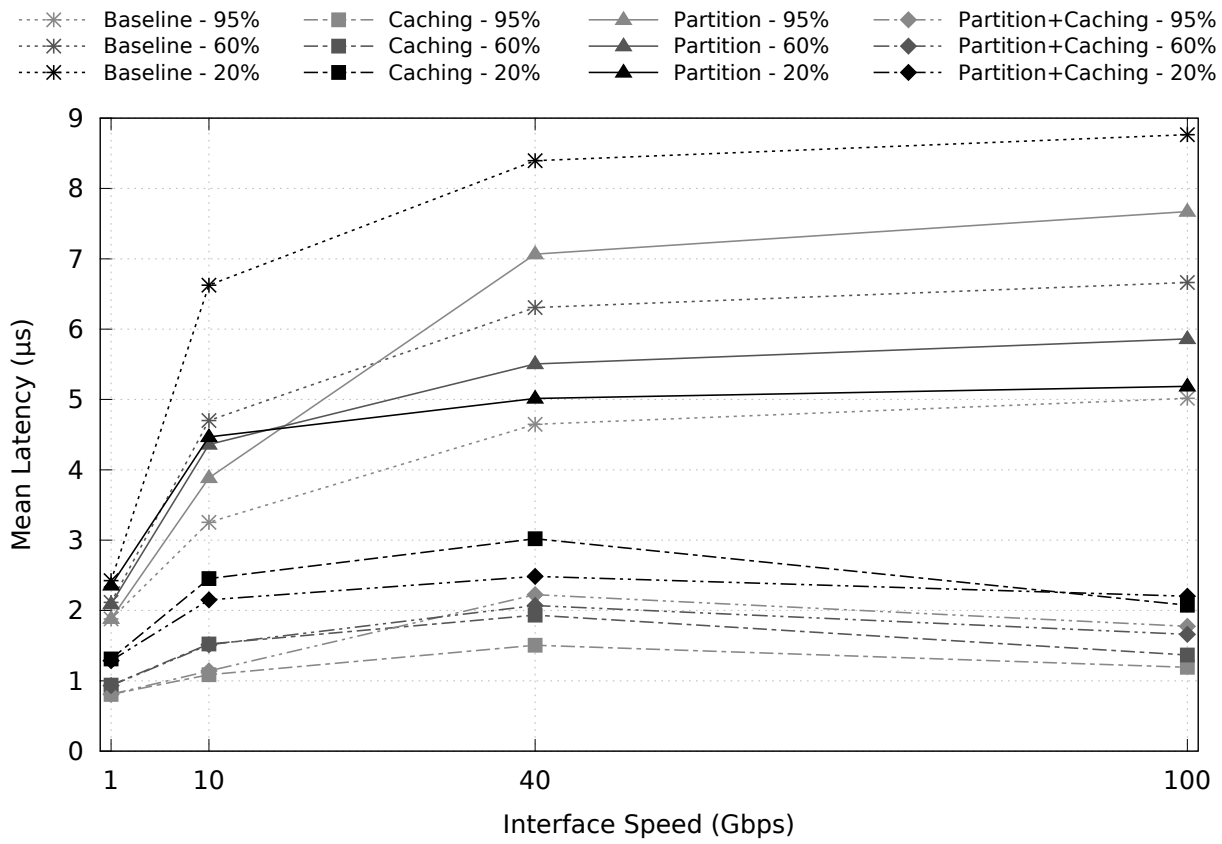


Figure 2.6: Measured Latency*

Mean Latency of $\text{Accepted}_{\text{auth}}$ for each classification architecture compared to baseline.

*©2014 IEEE. Reprinted with permission from L. McHale, J. Casey, P. V. Gratz, and A. Sprintson [31].

The increase in mean latency for the *Partition* architecture under nominal traffic conditions is caused by a longer delay-until-service for authorized packets in the unknown flow queue. The *Partition+Caching* architecture is able to achieve a consistent average latency, even when faced with adversarial traffic.

2.4.2.3 Classification Jitter

Figure 2.7 shows the latency jitter (*i.e.*, Standard Deviation of Latency) for each architecture. While the *Partition* architecture provides increased protection from DDoS attacks, it shows some increase in jitter, due to the priority mechanism. Note that jitter here is averaged across all flows and the increase is caused by the longer time-to-service of packets in the unknown flow queue. Once

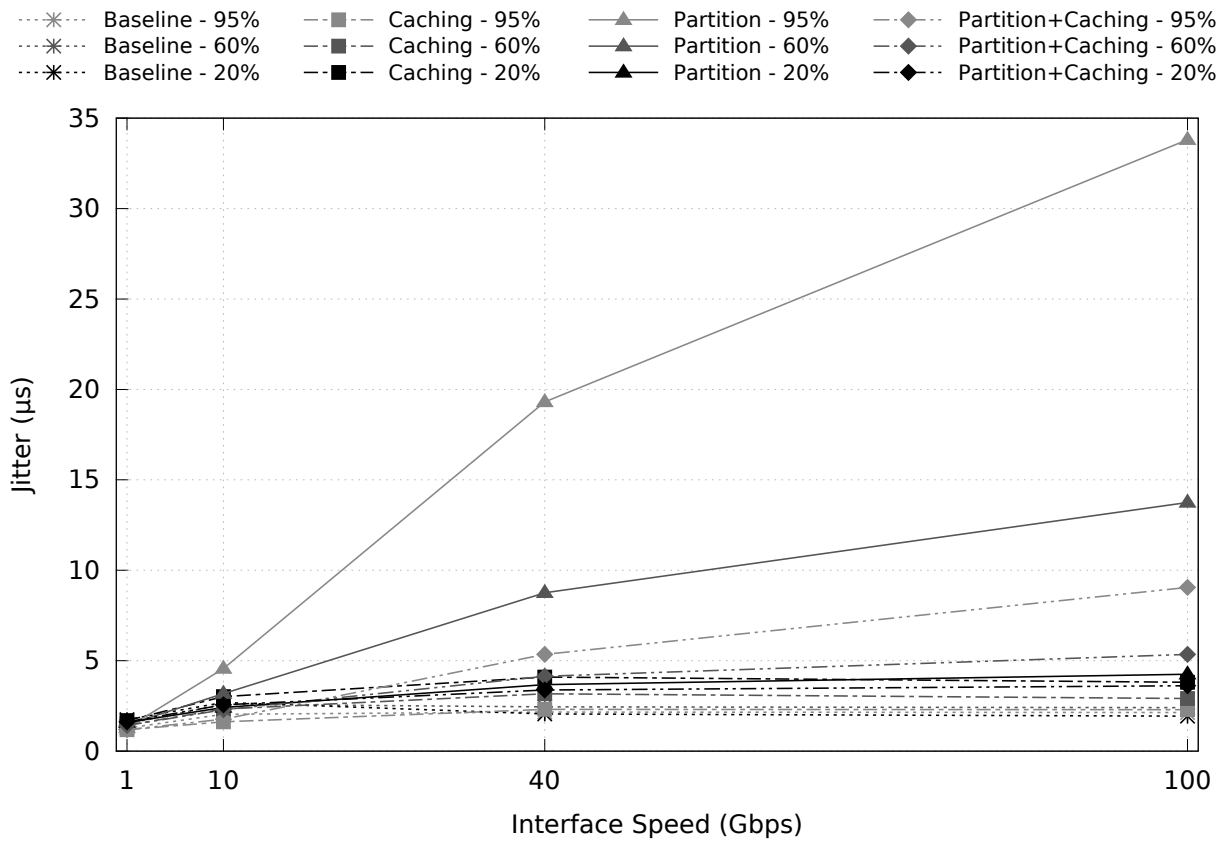


Figure 2.7: Measured Jitter*

Mean Standard Deviation of Latency ($\text{Accepted}_{\text{auth}}$) for each classification architecture compared to baseline.

*©2014 IEEE. Reprinted with permission from L. McHale, J. Casey, P. V. Gratz, and A. Sprintson [31].

the flow is learned, however, the flow's jitter will be consistent with the *Caching* architecture.

The *Partition+Caching* architecture significantly reduces the observed jitter compared to the *Partition* architecture; maintaining jitter comparable to the baseline. This self-metering attribute of the *Partition+Caching* architecture allows the data plane to provide higher effective quality-of-service to known flows, avoiding over-commitment of data plane resources in addition to and improving performance overall.

We kept queue depth small to minimize the chance of packet reordering by the priority mechanism. While raw packet reordering could occur whenever a priority mechanism is implemented, we observed zero actual packet reorders within a flow.

2.5 Summary

While SDN provides many advantages, their abstractions dramatically impact the design trade-offs of network appliances and the underlying architectures. As the complexity of SDN applications increase, data planes are becoming more susceptible to DoS attacks which can result in underlying network reliability and availability concerns. Thus, there is a strong need for data plane architectures, particularly in the context of SDN, that continue to operate efficiently in the presence of malicious traffic.

This chapter presents a novel approach to improve the consistency and reliability of SDN data plane classification. We validated our approach by examining an SDN network firewall application as a fundamental component to any security appliance. For this application, our architecture dramatically reduces the impact of malicious flows on established, authorized flows.

While the motivation of this chapter is centered around SDN, the approach presented in this chapter is general enough for flow-centric packet processing data plane architecture. Through leveraging a probabilistic data structure to pre-classify flows, the proposed architecture was able to decouple legitimate traffic from malicious traffic in the context of DDoS pressure. As a complementary component to stateful flow table cache management, *Pre-Classification* was able to reduce the impact of malicious traffic on a classification pipeline.

3. FLOW TABLE CACHE MANAGEMENT

3.1 Introduction

Caching is an important component of many network devices. Switching and routing functions are a backbone of all networks; however, an increasing amount of network functionality requires maintaining flow state for transiting connections. Modern stateful firewalls, security monitoring devices, and software defined networking require maintaining stateful flow tables. These flow tables often grow much larger than can be expected to fit within on-chip memory, requiring a managed caching layer to maintain performance. This caching layer is often obfuscated and embedded between the control and data plane layers.

The design-space of network devices is continually evolving with trade-offs in performance and reconfigurability. A common architectural advantage of network processors is to provide hardware-assisted table lookup extensions for both on-chip and memory-backed tables. Network functions requiring large flow tables, often consisting of millions of entries, require managing both on-chip and off-chip table resources. To meet performance and cost requirements, caching mechanisms are commonly employed. Caching a flow table using on-chip table resources preserves memory bandwidth and reduces latency, but introduces the complexity of effective cache management.

In processor microarchitecture, it has long been known that software inherently exhibits locality that can be leveraged at run-time. Cache management for network flow tables has historically seen less research interest than in processor microarchitecture. Insight into state of the art flow table management tends to be particularly shrouded in proprietary implementations. Existing flow table cache management heuristics are primarily based on the Least Recently Used (LRU) replacement as well as explicit protocol-based bypass mechanisms.

3.1.1 Contributions

This chapter aims to explore if locality in network traffic can be leveraged with more advanced cache management techniques. We review several modern cache management approaches in mi-

croarchitecture, highlighting our progress exploring flow table cache management. The contributions of this chapter include:

i. Flow table cache management limit study using Belady's MIN replacement algorithm on CAIDA network exchange traffic.

ii. First work to apply a Hashed Perceptron binary classifier to network flow table cache management.

iii. Iterative approach to feature selection and ranking in the context of Hashed Perceptron binary classifiers.

iv. Discussion of feature roles, granting further perspective into the dynamics of a Hashed Perceptron binary classifier.

3.2 Motivation

To bring perspective to the objective metrics used in this chapter, Section 3.2.1 first describes the significance of cache hit-rate to system performance. In order to understand the upper bound on room for improvement to cache hit-rate, we start with an optimality study in Section 3.2.2. Cacheable packet inter-arrival patterns and their timescales are further motivated in Section 3.2.3. Section 3.2.4 reviews well known stack-based cache management algorithms. Limitations to traditional approaches adapted to caching in networking are outlined in Section 3.2.4.1. Finally, we wrap up by introducing modern multi-perspective approaches to cache management using the hashed perceptron binary classifier in Section 3.2.5.

3.2.1 Significance of Cache Hit-Rate

System architects need to understand the impacts and potential drawbacks of caching, including estimations of processing latency and throughput. Caching allows mitigating slow-path processing requirements by taking advantage of locality. The actual performance benefit of a proposed caching implementation depends on the ability to capture the effective working set available in

the workload. It is important to quickly assess the potential expected performance improvement granted by a proposed caching architecture.

$$APPT \approx AMAT = T_{fast} + MissRate \times T_{slow} \tag{3.1}$$

Equation 3.1 rephrases average memory access time (AMAT) to approximate the sensitivity of cache hit-rate on average packet processing time (APPT). As accumulated packet processing effort roughly tracks processing latency, APPT can be approximated through a first-order processing latency estimation. In the context of packet processing, T_{slow} represents the average latency (or effort) exerted while traversing the slow-path. T_{fast} represents the average latency (or effort) consumed when a cache hit enables fast-path processing. It is expected that the processing effort between fast and slow paths could be an order or two apart – further motivating the desire to maximize cache hit-rate.

The wide verity of stateful network functions as well as packet processing architectures make it difficult to settle on a specific known ratio between T_{fast} and T_{slow}. For simplicity, this first-order sensitivity estimation is scoped to a single data plane packet processing slice. A more detailed analysis approach may be needed to take into account intricacies of parallel processing pipelines.

As is common in processor microarchitecture cache management, small percentile average hit-rate improvements often translate into notably significant ten-percentile performance gains. While the sensitivity estimation is ultimately left to the designer, it is conceivable that single-digit hit-rate improvements presented in this chapter may translate to a multiple-digit performance speedup for stateful network functions reliant on caching.

3.2.2 Cache Optimality Study

Originally developed to study optimal page replacement for virtual memory systems, Belady's Minimal page replacement algorithm (MIN) has been applied to cache replacement in processor microarchitecture [38, 39, 40]. Belady's MIN algorithm effectively orders the replacement stack by next furthest access, maximizing cache hit-rate.

Outside of a few specialized domains with bounded working sets or known access patterns, an optimal replacement ranking is usually impractical to build in hardware. Unpredictable random events will be capturable by MIN, but elusive to practical implementations which rely on pattern history. As an upper bound, Belady's MIN provides useful insight to cache replacement as well as potential headroom to improve cache hit-rate.

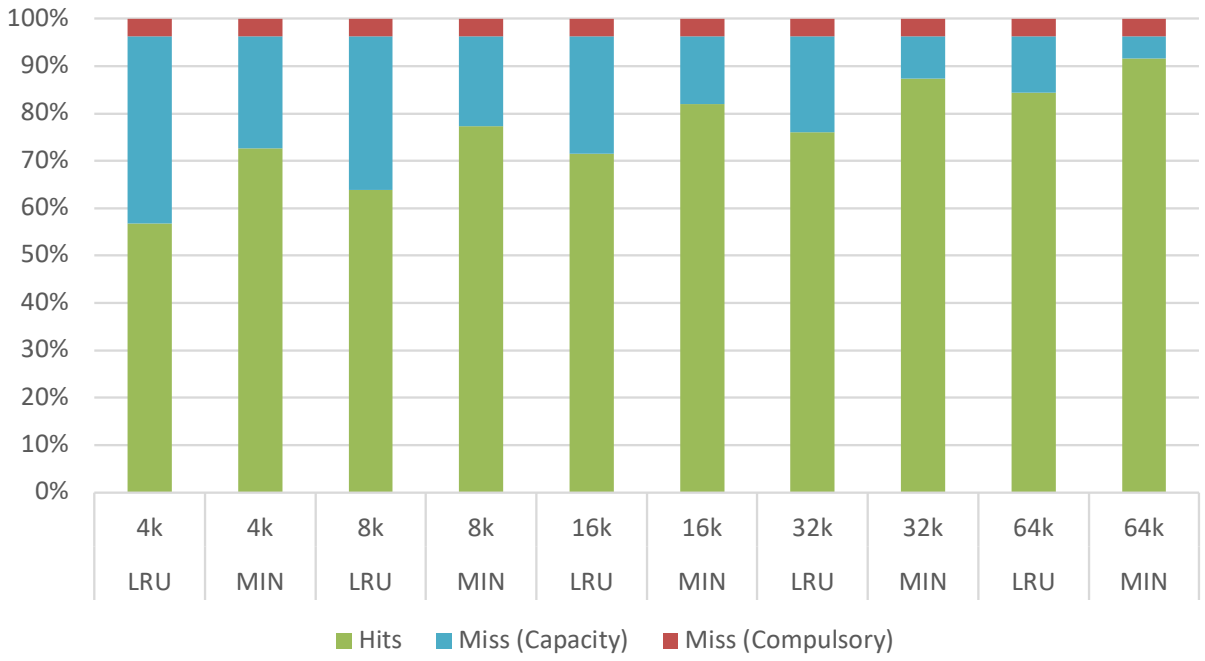


Figure 3.1: Belady's MIN Optimal Replacement

Fully associative flow table cache hit-rate simulation. CAIDA Equinix Sanjose January 2012 dataset.

Figure 3.1 shows the cache hit-rate for both the oracle Belady's MIN algorithm and a baseline fully-associative LRU implementation. While compulsory misses (misses caused by new flows) are unavoidable, MIN's capacity misses are entirely limited by the cache capacity and not also the replacement algorithm. The hit-rate delta between MIN and LRU is an upper bound to best possible hit-rate improvement achievable by improving the cache management algorithms.

At 4k flow entries, LRU achieved 15.8% fewer hits relative to MIN, with 23.7% true capacity misses remaining. This gap reduced to 7.2% fewer hits at 64k entries, with only 4.7% capacity misses remaining. It is expected that there will always be a gap between realistic cache management algorithms and oracle algorithms. However, the following sections aim to investigate the potential for improvement.

This limit study suggests there might be a potential cacheable working-set [41] larger than what LRU is able to capture. However, there is no guarantee that the gap between MIN and LRU is practically achievable. Intuitively, MIN also confirms greater potential improvement as cache size decreases. Section 3.4.3 further discusses the significance of hit-rate including the expected impact on overall system performance.

3.2.3 Flow Patterns

While network traffic, particularly transport layer protocol behavior, is relatively well understood and modeled [42]; we theorize that network flows also exhibit locality driven by program phases on connection endpoints. Examples of side-channel attacks [43] further support inherent end-host program behavior embedded within packet inter-arrival patterns.

Figure 3.2 showcases four hand-selected flows extracted from the 2012 Sanjose CAIDA dataset[1] in order to showcase a few representative multi-modal patterns. These plots should be interpreted not strictly as a linear time-series, but similar to a waterfall plot drawn across the x-axis, where the y-axis is a probability density of delays between packets.

Clusters of packet inter-arrival below the approximate transport layer Round Trip Time (RTT)[2] can be considered packet bursts. Similarly, inter-arrival delays around RTT can be roughly attributed to short-term transport-layer behavior. Finally, inter-arrival delays far exceeding RTT processing are likely attributable strictly to application level behavior, with an upper bound on flow keep-alive[3] delays.

[1] CAIDA Equinix Sanjose January 2012 pcap trace: equinix-sanjose.20120119.

[2] RTT is a measure of round-trip latency between two end-hosts – commonly on the order of 10ms to 100ms for Internet traffic.

[3] Keep-Alive is a common transport-layer timeout concept, primarily used to ensure a dormant flow remains tracked in stateful Flow Tables – usually on the order of seconds.

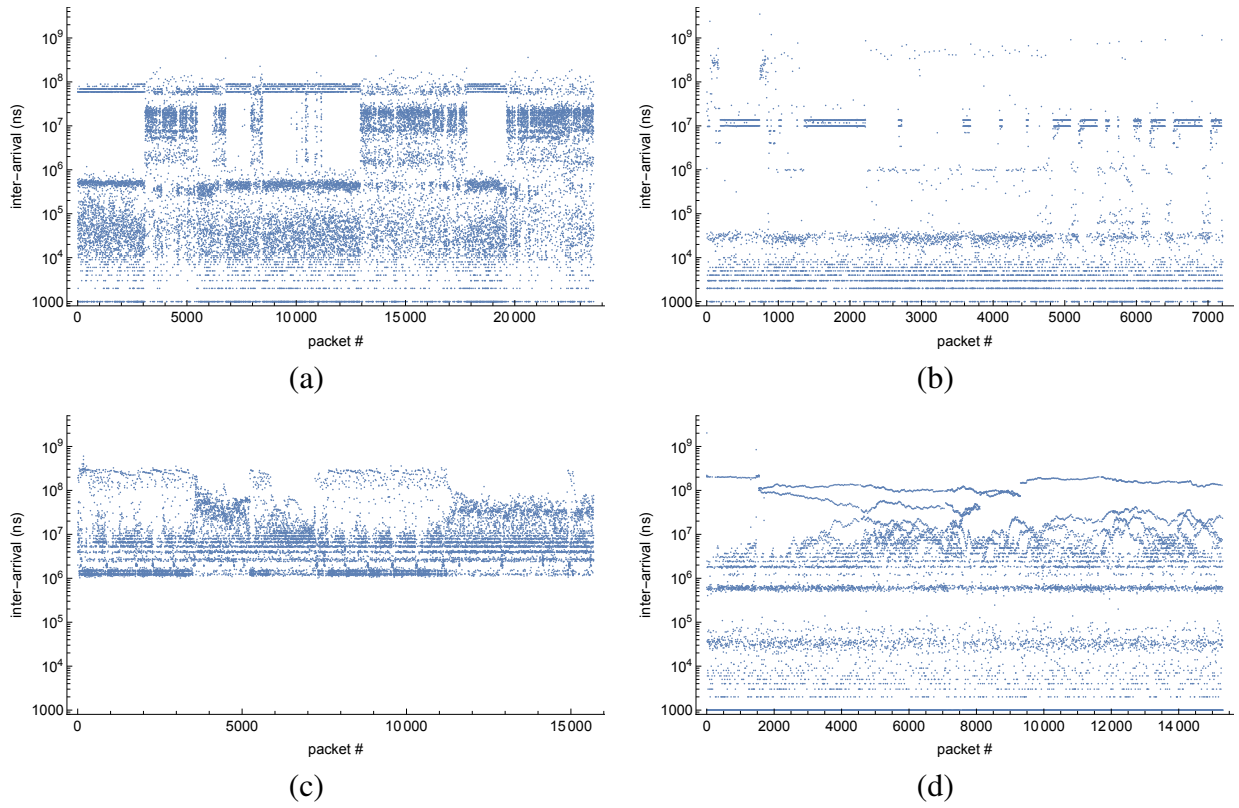


Figure 3.2: Interesting Packet Inter-arrival Patterns by Flow

The log-scale y-axis shows inter-arrival delay between packets (in nanoseconds), while the x-axis is packets since start of the flow.

While packet inter-arrival are inherently noisy, these flows appear to exhibit potential predictable, modal, almost cyclic patterns. We theorize that the large discrete shifts in packet inter-arrival delays are driven by end-host application behavior and are not simply transients in network conditions. In the context of cache management, these shifts in application behavior most notably impact packet burst duration and inter-frequency. Here we aim to explore whether there are cacheable patterns above what the canonical LRU replacement policy is able to identify.

3.2.4 Stack-based Algorithms

Cache management algorithms that maintain ranked order for replacement or insertion are formally referred to as stack-based algorithms. The Least Recently Used (LRU) algorithm intuitively maintains a replacement stack ordered by last access time. Specifically, LRU is analgous to Be-

lady's MIN reversed in time – ranking past instead of future accesses. Ultimately, LRU is effective when past history leads to a good prediction of the future.

LRU assumes reuse is always likely, inserting all new items in the most-recent position. Adaptive insertion policies such as Dual Insertion Policy (DIP) [44], allow insertion into either the top or bottom of the LRU recency stack. DIP is similar to bypassing the cache entirely, but does grant the demoted insertion a chance to become promoted. Adjusting the promotion/demotion logic of LRU can improve cache efficiency in the context of unequal probability of reuse [45].

Replacement policies based on reference counts have proven useful as a means to capture temporal patterns. Cache Bursts [46] introduced both RefCount and BurstCount as mechanisms estimate reuse predictions from cache entry hit counters. Reference interval counting mechanisms continue to be researched and improved, leading to Re-Reference Interval Prediction (RRIP) [47] and Signature-based Hit Predictor (SHiP) [48].

Branch prediction research has long been combining multiple information sources, leading to mechanisms to combine or prioritize multiple predictors. The TAGE predictor is similarly able to combine multiple predictions in a ranking mechanism to select a most likely accurate prediction [49]. While the Hashed Perceptron mechanism has been steadily gaining momentum as a robust means to consider multiple potentially indicative features together as a whole [50, 51].

The Hashed Perceptron technique originated in branch prediction, but has been successfully applied to cache management [52, 53, 54, 55]. Recent research has also leverage the Hashed Perceptron structure as a filtering mechanism to improve prefetching quality [56]. Similar efforts towards prefetching filtering were also accomplished using Bloom Filters [57].

Several research efforts have applied offline deep learning techniques towards cache replacement [58]. Similarly, Hawkeye [59] leveraging a time-delayed belady's min algorithm as an online replacement prediction feedback mechanism.

3.2.4.1 Preliminary Exploration

In processor microarchitecture, access patterns are constantly shifting, depending on program behavior. Similarly in networking, flows also exhibit modes of operation depending on the protocol

transport layer as well as end-host program phases. In the context of a flow table cache replacement policy, LRU does a decent job capturing short-term packet bursts. However, LRU's ranking can also be polluted by dormant, low-bandwidth flows competing for cache resources. LRU has no way to differentiate reuse probability, treating all flows as equally likely.

The insertion policy is especially critical to flow table cache management due to the short lifetime of most flows. Most flows are short lived with the majority of bandwidth attributed to a small subset of flows [60]. However, it is particularly challenging to predict which flows will be short lived early in the flow life-cycle.

Inspired by prior works in CacheBursts [46] and SHiP [48], cache entry hit counters seemed well suited as a means to track packet bursts. However, early attempts at leveraging CacheBursts and SHiP directly as cache management strategies resulted in performance degradation compared to baseline LRU implementations when applied to flow table cache management. That said, CacheBursts ended up being a notably informative feature component in our proposed final design.

Our single-perspective cache management approaches were not able to outperform LRU replacement when applied to managing flow tables. Packet arrival variability inherent in networking adds significant noise to simple pattern predictors such that simple predictor tables alone could not outperform a baseline LRU implementation. It became evident that a more robust pattern correlation mechanism is needed to rank reuse probability amongst noisy inter-arrival patterns.

3.2.5 Hashed Perceptron Binary Classifier

The Hashed Perceptron binary classifier has been successfully used in processor microarchitecture as an approach to improve cache management heuristics. While research on how to integrate and apply this prediction technique has extended over a decade, the technique is now being shipped in modern processors in recent years.

Much like modern machine learning approaches, the Hashed Perceptron improves overall reuse prediction quality by combining multiple predictive features into a singular weighted prediction. While conceptually similar to perceptrons in a single-layer neural network, the Hashed Perceptron

has the significant advantage of being efficient to implement in both hardware and software. The Hashed Perceptron is also paired with an adaptive training technique to adjust to changing run-time behavior.

Differing from a traditional Perceptron model which scale analog values, requiring integral meaning for each perceptron input. The Hashed Perceptron is able to map disjoint binary inputs to a corresponding correlation using the feature's table. The correlation output can be interpreted as an integral value, expressing a historical correlation of a particular feature to the inference question at hand. During inference, combining multiple features' independent correlations result in a strengthened overall prediction.

The processor microarchitecture community has demonstrated the Hashed Perceptron is well suited for reuse prediction, performing especially well in noisy multicore cache hierarchies. This chapter outlines our exploration into adapting the Hashed Perceptron mechanism to flow table cache management.

Our early exploration discovered that relying on any single feature is fragile, especially when applied to network flow table cache management. Traditional single-feature approaches failed to surpass the consistent performance of LRU when applied to network caching. The variety of traffic patterns encountered in network cache management demands a general approach to consistently approximate a good working set.

Caching will continue to be a fundamental component of network data planes as a means to amortize costly control path decisions. The trend in networking to support programmability of generic network functions increases control path complexity. Advances in cache management strategies leveraged in modern processor microarchitecture are absent in the networking domain. Clearly there is a gap between Belady and LRU, can we do better?

3.3 Design

One of the goals of this work is to explore a generalized approach to flow table cache management. Stateful flow tables are core components in the classification pipeline for firewalls and edge security devices. However, caching can be applied generically to any large table supporting a clas-

sification pipeline. While the focus of this chapter is on improving cache management for stateful flow tables, the techniques we propose can be applied to assist any data plane table considered for caching.

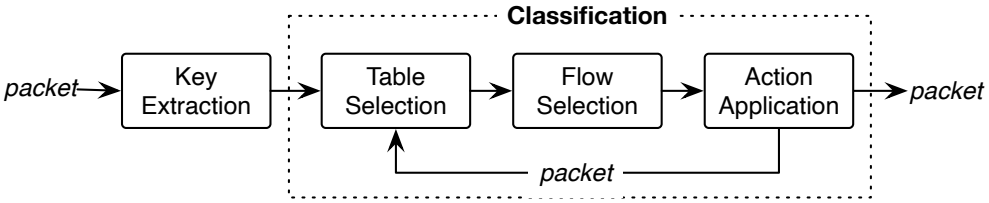


Figure 3.3: Data Plane Classification Processing

Figure 3.3 portrays an abstract cycle of table selection, flow selection, and action application as the underlying fundamental classification abstraction. Network functions are composed by chaining multiple logical rounds of classification across relevant flow-state tables. Network functions that also manipulate packets often also queue data plane and packet actions[4]. Fundamentally, SDNs aim to expose table management and packet manipulation for a composable packet processing pipeline.

Figure 3.4 shows a classification pipeline focused around a flow cache and a backing stateful flow table. It is useful to reiterate that a stateful flow table often resides in off-chip memory due to the number of concurrent flows tracked. Stateful flow tables are managed by the control plane and a cache is often instantiated in the data plane to meet performance requirements. There are two high-level scenarios encountered in stateful flow table management:

- **New Flow:** Requires classification and a new reservation in the stateful flow table.
- **Existing Flow:** Track flow state in relevant table entry for associated packet.

A flow cache and its management algorithm must then contend with three scenarios triggered by incoming packets:

[4]OpenFlow specifies certain actions apply immediately, while others may be queued in an action-set with notable complications.

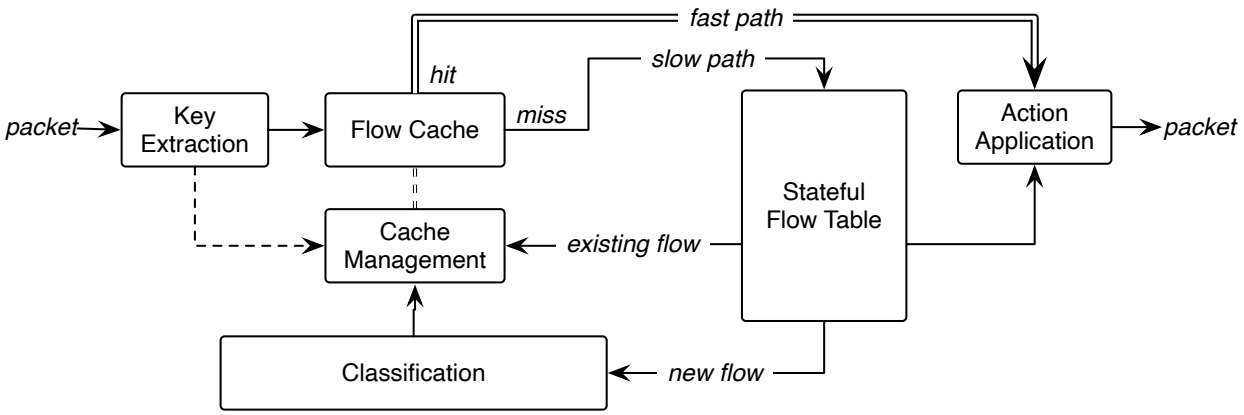


Figure 3.4: Flow Cache Management

- **Compulsory Miss:** *New flow*, not yet tracked in the stateful flow table.
- **Capacity/Conflict Miss:** *Existing flow*, not currently cached.
- **Cache Hit:** *Existing flow*, cached.

Cache management algorithms consist of two distinct components: the insertion and replacement policy. A dynamic insertion policy may choose to bypass cache insertion if the flow is predicted to have a low probability of reuse before eviction. In networking, bypassing flows based on static protocol heuristics is somewhat common, for example ICMP messages. However, there is additional opportunity for an insertion algorithm to bypass tracked flows that have a low probability of cache reuse. Both bypass and early replacement require a means to estimate reuse probability.

Additionally, a dynamic replacement policy may choose to evict a cache entry earlier than a static replacement policy otherwise would. Since static replacement policies grant all entries equal opportunity for reuse, some flow entries are stuck waiting for eviction, taking up valuable cache resources even when the probability for reuse is markedly lower. Dynamic replacement policy with an early eviction mechanism provides an additional opportunity to improve cache efficiency by removing stale entries sooner.

A packet without an associated flow table entry is considered a new flow, requiring classification. Once the flow is classified, the a flow table entry is reserved and the cache management

algorithm may be consulted on whether to cache the entry (insert or bypass). From the perspective of the flow cache, new flows triggering classification are compulsory misses – an unavoidable slow-path.

Once a flow is tracked by the stateful flow table, the classification pipeline no longer needs to be consulted. However, the effectiveness of the limited cache resources is ultimately up to the cache management algorithm. This design aims to minimize the cache misses encountered by existing flows by exploring better cache management techniques.

Packets that encounter hits in the flow cache may simply update the cached flow table entry without needing to access off-chip flow table resources. A more intelligent cache management algorithm has the potential to significantly improve the flow cache hit-rate and thus data plane processing performance.

The remaining section outlines the design details of the flow correlator mechanism. Apart from design-time feature selection, the hashed perceptron flow correlation approach does not require global weight training. The hashed perceptron feedback mechanism resembles an online training technique, adjusting correlation tables dynamically at run-time.

The adaptation of hashed perceptron cache management as a flow correlation mechanism is described in Section 3.3.2. As with many machine learning approaches, selecting useful Features at design-time is non-trivial and requires artful exploration. Our approach to feature design and metrics are covered in Section 3.3.3. Since feature selection consumed a majority of our effort during design iteration, we developed an iterative optimization approach generically applicable to tuning Hashed Perceptron based cache management. Section 3.3.4.2 describes this methodical approach to approximating information gained by differential performance analysis.

3.3.1 Classifier Metrics

Early in feature exploration, we heavily leveraged average and standard deviations of both pure and mixed features to help gain insight into input sparsity. However, simply analyzing input and output distributions alone isn't sufficient to distinguish predictive features.

Analysis of binary classification problems is well studied, applicable across many research

domains. The aptly named *Confusion Matrix* is commonly used to summarize the behavior of binary-outcome systems. Table 3.1 depicts a common representation of a confusion matrix.

		Predicted	
		Positive	Negative
Actual	Positive	True Positive	False Negative
	Negative	False Positive	True Negative

Table 3.1: Confusion Matrix

While simple ratio metrics are heavily used in certain domains, they aren't sufficiently robust for this work. *Accuracy* (Eq. 3.2) considers both positive and negative cases; however it is easily swayed by even moderately unbalanced datasets. The *F1* score (Eq. 3.3) is similarly unsuited as it is essentially *accuracy* focused only on the positive set – used primarily as a positive detection metric.

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.2}$$

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{3.3}$$

Matthew's Correlation Coefficient (MCC) is a particularly useful metric to compare binary classifier performance [61]. MCC (Eq. 3.4) is derived from Pearson's correlation coefficient applied specifically to binary classification. MCC provides a balanced view of the confusion matrix, reducing decision bias from skewing the perceived accuracy. Appendix A.1 provides further intuition on the interpretation of MCC as defined in Equation 3.4

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{3.4}$$

As a correlation coefficient, MCC is bound between -1 and 1. A positive MCC indicates a correlation to correct predictions; while a negative MCC indicates anti-correlation. An MCC near zero indicates a weak correlation, hinting at a poor prediction confidence resembling a random process. As MCC approaches ± 1, the predictor is deemed to have a relatively higher prediction confidence. Ultimately MCC provides a balanced view of binary classifier performance, particularly useful when operating within reasonable bias limitations [62, 63].

		Predicted	
		Active	Dormant
Actual	Active	Active Correct (TP)	Dormant Incorrect (FN)
	Dormant	Active Incorrect (FP)	Dormant Correct (TN)

Table 3.2: Flow Correlation Confusion Matrix

To assist in subsequent sections, Table 3.2 specializes the confusion matrix around predictions of flow activity in the context of cache management. Section 3.3.2.2 will connect each case in the confusion matrix to the hashed perceptron training algorithm.

3.3.2 Flow Correlator Design

The hashed perceptron technique provides a structured mechanism to leverage multiple prediction tables for improved overall accuracy and resiliency. The hashed perceptron training algorithm plays a crucial role in keeping these predictor tables balanced across multiple features. Apart from design-time feature selection, the hashed perceptron flow correlation approach does not require global weight training. The hashed perceptron feedback mechanism resembles an online training technique, adjusting correlation tables dynamically at run-time.

The hashed perceptron consists of correlation tables (feature tables) consulted during inference. Each correlation table is indexed by a feature and contains saturating counters (weights) resembling

typical prediction tables. The Hashed Perceptron differs from single-perspective predictors by combining the predictions from multiple feature tables to produce a final combined decision. The magic of the hashed perceptron centers around the ability to manage multiple tables cohesively using a prediction feedback mechanism (the hashed perceptron training algorithm).

Flow table cache management relies on accurately predicting reuse for a given flow. This ultimately boils down to predicting packet inter-arrival patterns for each active Flow. More specifically, predicting the likelihood of a subsequent packet arrival for a given flow within the relative time window allotted by the cache's working set.

The Hashed Perceptron mechanism is conceptually split into two phases: inference and reinforcement. Section 3.3.2.1 describes the process of generating a reuse prediction for every packet arrival. Section 3.3.2.2 describes the Hashed Perceptron training algorithm as the basis for maintaining prediction tables.

3.3.2.1 Inference

Cache management predictions are generated in real-time by the flow correlator inference mechanism. Every packet event triggers an inference prediction for cache management, thus it is crucial that inference is able to meet the packet processing requirement of the underlying design target.

Cache management handles two scenarios depending on the cached state of the flow entry. First, a flow cache miss triggers a cache management decision to insert or bypass the flow entry. Secondly, cache management seeks to identify optimal scenarios for early eviction. An early eviction, or conversely a reuse prediction, is performed on a flow cache hit. In both scenarios, cache management consults flow correlation inference to estimate the likelyhood of flow entry reuse relative to overall cache pressure.

Both predictions are essentially re-framing an underlying flow entry reuse prediction around a relative ranking of flow activity. Thus, the flow correlator inference and training pipelines are organized around the concept of *active* or *dormant* flows. Both bypass and early eviction share table resources, interpreting positive table weights as likely to be *active* and negative weights as

likely *dormant*.

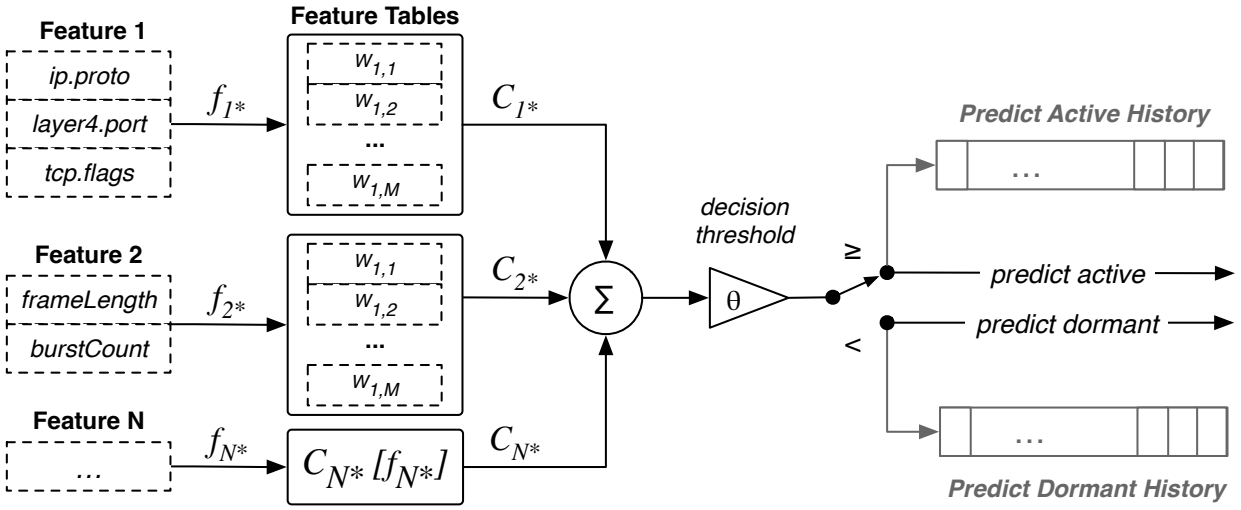


Figure 3.5: Flow Correlator Inference

Figure 3.5 outlines the flow correlator's inference pipeline. In order for cache management to generate an inference, the feature components must first be gathered from the packet header bitfields and the relevant flow table entry. Stateless protocol feature components originate from the packet headers and are readily available by preexisting classification *Key Extraction* operations. Stateful feature components (e.g. burstCount) are pulled from a cached flow table entry during a cache hit, or simply zero during a cache miss.

Each assembled feature (f_{N*}) indexes into a corresponding feature table providing a mapping to independent correlations (C_{N*}). The decision threshold (θ) is applied to the sum of each feature's correlation. Since θ is zero for this flow correlator design, the magnitude of the accumulated correlations can be interpreted as a confidence and the sign represents the decision.

On inference, each feature contributes an opinion as a correlation count. The hashed perceptron structure accumulates the correlations from all features, thresholding to create a reuse prediction. The confidence of the feature's opinion, can be inferred by the magnitude of the correlation count. Similarly, the confidence of the overall prediction is the magnitude of all accumulated correlations.

Feature tables map a disjoint input feature-space to cache reuse correlations. Loosely resembling a hash, features are often orthogonal combinations of several bit fields. These feature table lockups are performed in parallel. Each feature table holds saturating correlation counters, commonly referred as weights. Each feature table's length is 2^m entries where m refers to the bit-width of the corresponding feature assembly. The implementation requirements of each feature is then simply $w * 2^m$ bits, where w refers to the saturating counter bit-width.

Flow entries marked for early eviction by the flow correlation mechanism are in the cache, but prioritized for replacement. When a flow entry needs to be inserted and no entries are marked for early eviction, replacement falls back to LRU. If a flow entry marked for early replacement encounters a hit before eviction, the prediction is corrected and a training event is triggered.

3.3.2.2 Reinforcement

The hashed perceptron mechanism consists of an online training algorithm organized around prediction feedback. Applied to flow correlation, prediction feedback seeks correlations of flow entry reuse across feature vectors. With respect to the available cache resources, the flow correlation predictor aims to rank flow entries by likelihood of reuse within the available cache working set.

Active flow predictions are inserted into the active history queue, while *dormant* flow predictions are inserted in the dormant history queue. In order to reinforce cache management decisions, these feedback queues must be searchable by a unique flow identifier. The flow identifier implementation leveraged by the flow cache may be borrowed for this purpose.

Hashed perceptron training reinforcement consists of four potential scenarios, each corresponding to entries in the confusion matrix. *Active* prediction reinforcement is shown in Figure 3.6 and Figure 3.7 covering the *true positive* and *false positive* cases, respectively.

In Figure 3.6, the example flow F_3 has a prior *active* flow prediction awaiting confirmation in the *active history* queue. Since the prior reuse prediction is now proved correct, the feature vector, $\langle f_3 \rangle$, associated with the previous prediction is popped off of the queue. If the prediction confidence associated with $\langle f_3 \rangle$ do not surpass the training threshold, the corresponding feature

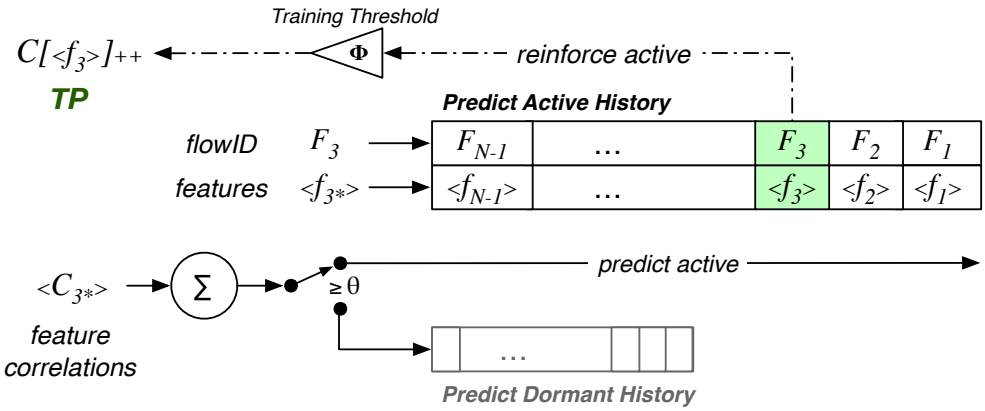


Figure 3.6: Active Correct (TP) Reinforcement

table correlation counts are incremented. The new reuse prediction associated with F_3 is also pushed into the *active history* queue along with the associated new feature vector, $\langle f_{3*} \rangle$.

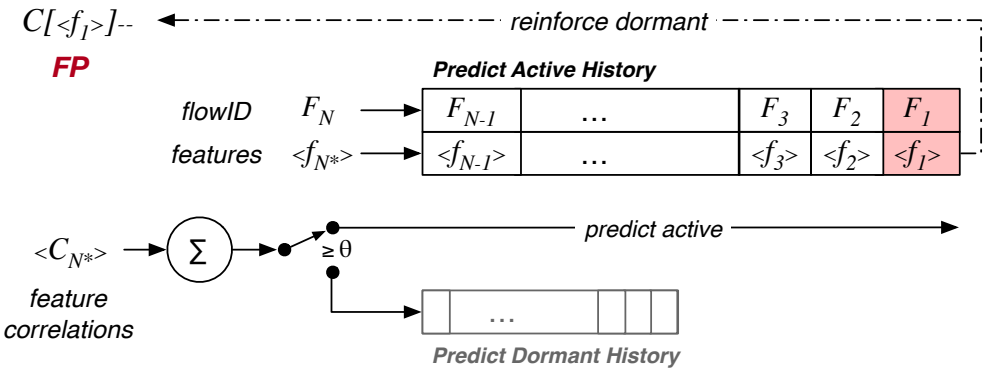


Figure 3.7: Active Incorrect (FP) Training

Figure 3.7 outlines how incorrect *active* flow predictions are determined. Example flow F_N does not have a previous reuse prediction, yet is predicted active by inference. In order to free a slot, the oldest prediction is popped off the end of the *active history* queue and considered incorrect. The example flow F_1 and feature table correlation counts associated with feature vector $\langle f_1 \rangle$ are decremented. Note the training threshold is omitted since the *active* flow prediction associated

with F_1 was determined to be incorrect. The new reuse prediction associated with F_N is pushed into the *active history* queue along with its feature vector, $\langle f_{N*} \rangle$.

The next two figures illustrate *dormant* flow prediction reinforcement covering the *true negative* and *false negative* cases of the confusion matrix, respectively. Conversely to *active history*, entries popped off the end of the *dormant history* queue are considered correct.

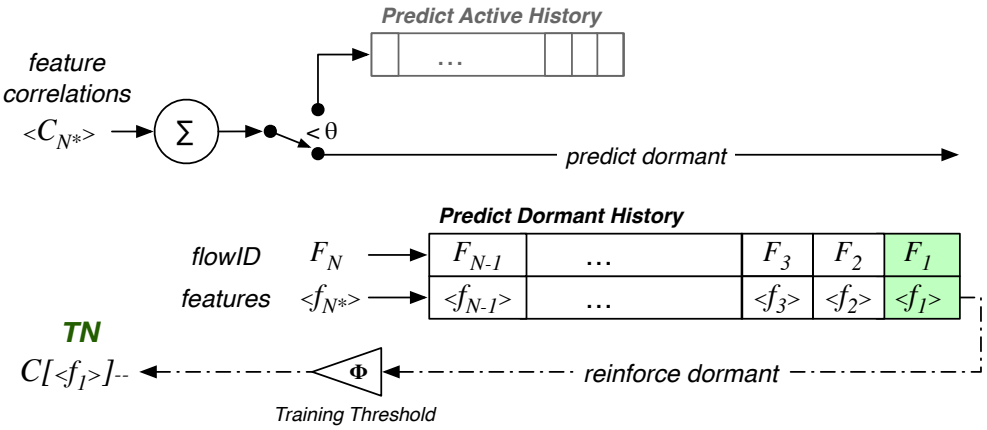


Figure 3.8: Dormant Correct (TN) Reinforcement

Figure 3.8 illustrates example flow F_N predicted *dormant* during inference. Finding no prior prediction associated with F_N, the oldest prediction in the *dormant history* queue is popped off the end. If the prediction confidence associated with $\langle f_1 \rangle$ do not exceed the training threshold, the corresponding feature table correlation counts are decremented – reinforcing the correct *dormant* prediction. The new *dormant* prediction associated with F_N is pushed into the *dormant history* queue along with its feature vector, $\langle f_{N*} \rangle$.

In Figure 3.9, the example flow F_3 has a prior *dormant* flow prediction awaiting confirmation in the *dormant history* queue. The prior *dormant* prediction was caught before falling off the end of the queue, indicating that feature vector $\langle f_3 \rangle$ may contain hints towards flow reuse. The existing F_3 entry and associated feature vector, $\langle f_3 \rangle$, is removed from the queue and the respective feature table correlation counts incremented. Note once again that the training threshold is omitted

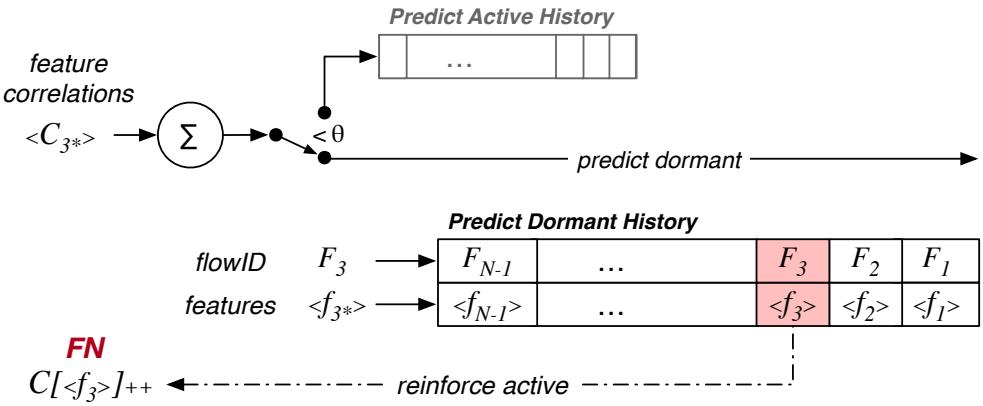


Figure 3.9: Dormant Incorrect (FN) Training

since the *dormant* flow prediction associated with F_3 was determined to be incorrect. Finally, the new reuse prediction associated with F_3 is pushed into the *dormant history* queue along with the associated new feature vector, $\langle f_{3*} \rangle$.

Seznec developed a dynamic training thresholding mechanism for the GEometric History Length branch predictor [64]. We leveraged Seznec adaptive mechanism for the *training threshold* (Φ), which also requires the *decision threshold* (θ) to be centered around zero. Through experimentation, we confirmed that Seznec's target $1:1$ ratio of correct to incorrect predictions resulted in the best overall predictor accuracy, matching our experimentally tuned parameters.

While the optimal Φ depends on the cache pressure, we observed a notably stable threshold once the predictor warmed up and aggressiveness balanced. It is also notably advantageous to keep the decision threshold (θ) intuitively at zero – preserving the numeric symmetry of weights as well as accumulated correlations.

3.3.3 Feature Design

Table 3.3 lists features explored during the design process with remarks on approximate intent behind each combination. There is limited information available from fields contained within each packet. However, we found notable value in associating temporal patterns with hints from packet fields. The features outlined in this table are nowhere near an exhaustive exploration, but rather a

first pass to identify useful combinations.

$f_{\#}$	**Feature Type**	**Feature Components**	**Remarks**
0	Control	$UniformRandom$	Uncorrelated: map to random entry
1	Pure Mix	$ipProto \wedge min(srcPort, dstPort)$	Protocol hints
2	Pure Mix	$ipv4Dst[31{:}16] \wedge dstPort$	Destination service
3	Pure Mix	$ipv4Src[31{:}16] \wedge srcPort$	Source service
4	Pure	$flags$	TCP/IP flags
5	Pure	$srcPort \wedge dstPort$	Bi-directional TCP/UDP ports
6	Pure Mix	$f_{TCP} \ll 7 \wedge f_1$	Mix TCP flags with protocol hint
7	Pure	$(ipv4Dst \wedge ipv4Src)[31{:}16]$	Bi-directional upper 16-bits IP addresses
8	Pure	$(ipv4Dst \wedge ipv4Src)[23{:}8]$	Bi-directional middle 16-bits IP addresses
9	Pure	$(ipv4Dst \wedge ipv4Src)[15{:}0]$	Bi-directional lower 16-bits of IP addresses
10	Pattern	$\lceil flowPackets \rceil_{16}$	Packets since start of flow
11	Pure	$ipLength$	IP frame length
12	Pure	$flowID$	5-tuple flow identifier
13	Pure Mix	$f_{12} \wedge f_{11}$	Unroll frame length over flowID
14	Pattern	$\lceil refCount \rceil_{16}$	Hit count while inserted into cache
15	Pattern	$\lceil burstCount \rceil_{16}$	Hit count while in MRU position
16	Pattern Mix	$\{\lceil f_{14} \rceil_8, \lceil f_{15} \rceil_8\}$	Concatenate RefCount and BurstCount
17	Pure + Pattern	$f_{16} \wedge f_{12}$	Unroll Burst/RefCount over flowID
18	Pure + Pattern	$f_{16} \wedge f_7$	Unroll Burst/RefCount over upper IP
19	Pure + Pattern	$f_{16} \wedge f_8$	Unroll Burst/RefCount over middle IP
20	Pure + Pattern	$f_{16} \wedge f_9$	Unroll Burst/RefCount over lower IP
21	Pure + Pattern	$f_{11} \wedge f_7$	Unroll frame length over upper IP
22	Pure + Pattern	$f_{11} \wedge f_8$	Unroll frame length over middle IP
23	Pure + Pattern	$f_{11} \wedge f_9$	Unroll frame length over lower IP
24	Pure Mix	$ipFragOffset \wedge f_4 \wedge f_{11} \wedge f_{12}$	Unroll fragment offset, flags, and frame length over flowID
25	Pure Mix	$f_7 \wedge f_4$	Unroll flags over upper IP
26	Control	$NULL$	Sparsity: map to single entry
27	Pure + Pattern	$\{f_{11}, \lceil f_{15} \rceil_8\}$	Concatenate BurstCount with frame length
28	Pure + Pattern	$f_{16} \wedge f_{11}$	Unroll Burst/RefCount over frame length

Table 3.3: Complete List of Explored Features

Concatenation syntax is represented as $\{A, B\}$. $\lceil A \rceil_N$ refers to the ceiling function: $min(A, N)$. The XOR operation is indicated by $\wedge$. $A \ll N$ represents left shift left by N-bits. Bit-field selection is indicated using the syntax $[M{:}N]$, where bit positions M through N are extracted.

The *Feature Type* column approximately categorizes each feature by information sources.

There are two potential sources of information: *Pure* (stateless) and *Pattern* (stateful). Pure feature components originate from packet headers. Pattern feature components introduce temporal information, not already available within packet headers. *Pure Mix* are simple combinations of packet header bit-fields. *Pattern Mix* refers to combinations of temporal state. Finally, *Pure + Pattern* are combinations of both packet header bit-fields and temporal state. Appendix A.2 provides all feature table weight distributions for comparison alongside the descriptions below.

3.3.3.1 Pure Features

There are really only five pure feature components originating from packet headers explored in this study. These include IP source and destination addresses, IP protocol identifier, TCP/UDP port numbers, IP frame length, and few grouped IP/TCP flags. The IP flags assembled in f_4 are: DF and MF. The TCP flags assembled in f_4 and f_6 are: SYN, FIN, RST, PSH, ACK, URG.

3.3.3.2 Temporal Pattern Features

Pattern features are distinguished from Pure features as they require temporal metadata in the flow table entry. These pattern feature components do not originate from packet headers, but offer the ability to associate sequences or temporal patterns between packets. Pattern based featured proved to be valuable sources of information, but required mixing to unlock their potential.

The flowPackets feature (f_{10}) is simply the packet count since the flow started, a counter often already present in stateful flow tables. RefCount (f_{14}) and BurstCount (f_{15}) on the other hand are similar counters, but unique to the lifetime of the cached flow table entry. RefCount is simply the packets observed since the flow was inserted into the cache, while BurstCount is the packets observed while the entry remains in the MRU position.

3.3.3.3 Combined Features

There are endless ways to assemble features with complementary pure and pattern-based components. There were two conceptual approaches when attempting to constructively combine features components: *orthogonal combination* and *correlation unrolling*.

Orthogonal combinations allows for better utilization of feature table resources through a union

of independent feature components. For example: some feature components are useful for bypass predictions, while others provide hints toward early eviction. Usually we observed a trade-off in feature information density at the cost of collisions and reduced confidence. However, we also observed potential for coincidental benefit – correlations simultaneously occurring, strengthening feature confidence.

Feature unrolling, as an approach towards enabling constructive feature combinations aiming to distribute global correlations across subsets of flow designators such that opportunities for independent correlations may arise. For example: temporal patterns tend to perform better when associated with partial IP address subsets, protocol, or port designators. Correlation unrolling is a balance and can either benefit or detract from feature performance.

Pattern-based features are particularly complex in the context of networking. Subsets of flows may exhibit similar patterns, but all flows together tend to be quite noisy with conflicting inter-arrival patterns. Thus, we observed notable utility in correlating pattern-based feature components across protocol differentiating components. It is also important to conceptualize that each packet in a given flow maps to different flow table entries throughout the sequence. The inter-arrival access patterns is thus encoded across multiple entries in any given table.

There is certainly a trade-off ranging from targeting independent per-flow correlations to strategic groupings sharing correlator entries. Unlike bloom filter designs where hashes are desired to be as uniquely identifying and free of collisions as possible, there is a balance in hashed perceptron feature construction where carefully selected subsets improve correlation potential.

The designer can choose to combine orthogonal features, or unroll features by spreading potential correlations out over subsets of flows. Since necessary unrolling can increase a features effective training latency, there is a non-obvious trade-off between the concept of sharing correlations across subsets of flows as well as enabling flows to maintain independent correlations. The features presented in this work are almost certainly non-optimal, but do provide insight towards successful (as well as unsuccessful) combinations.

3.3.3.4 Control Features

To help understand the natural bias of the system, two control features were included in the study: UniformRandom (f_0) and NULL (f_{26}). These were only used in the feature exploration process as a likeness comparison UniformRandom simply chooses a table entry randomly based on a uniform random distribution, providing insight into the hysteresis of the system. NULL is effectively a single table entry (5-bit saturating counter), representing sparse feature behavior.

Combining too many components reduces the opportunities for correlations. We have observed that heavily mixed features, such as features that combine the full flow identifier (flowID, f_{12}) tend to under-perform features that combine a partial flow identifier such as f_7. All components mixed together into a single feature tends to approach UniformRandom (f_1) while sparse features tend to resemble NULL (f_{26}).

3.3.4 Feature Metrics

Since the hashed perceptron structure consists of multiple independent prediction tables contributing opinions towards an overall prediction, there is a need to analyze feature performance relative to the overall system. Just as MCC is a notably useful classifier metric, it can also be calculated with respect to each feature, independently.

The Hashed Perceptron training feedback is assumed to be ground truth. Absolute truth is non-trivial to ascertain as each misprediction has implications that propagate forward in time. Imperfect, ground truth is a practical compromise that extends to all feature analysis techniques in this work.

3.3.4.1 Feature Influence

While MCC is well suited for relative feature comparisons, it doesn't quite grant insight into the confidence of predictions. Specifically in the context of the hashed perceptron mechanism, each feature has the potential to influence the system prediction equally. However well behaved features self-moderate their confidence by the weight of their contribution. Understanding how a feature contributes to the overall system provides valuable feedback into the behavior of feature

components as well as intuition into successful combinations.

		Predicted	
		Positive	Negative
Actual	Positive	$\frac{\sum \vec{w}_f}{TP}$	$\frac{\sum \vec{w}_f}{FN}$
	Negative	$\frac{-\sum \vec{w}_f}{FP}$	$\frac{-\sum \vec{w}_f}{TN}$

Table 3.4: Influence Matrix

Table 3.4 shows how influence is an extension of the confusion matrix. The influence matrix is categorized with respect to the overall correctness of the system prediction (TP, TN, FP, FN). The vector notation simply indicates that feature weights ($\vec{w_f}$) are accumulated independently, with respect to each system prediction.

Note that influence of a single-feature predictor is simply the average weight with respect to that predictor's confusion matrix. However, the influence metric becomes more than just an average weight when per-feature contributions are tracked with respect overall system decisions. Influence grants insight into the feature's contribution, or lack thereof, relative to the correctness of the system's overall prediction.

Influence can be analyzed with respect to each quadrant of the confusion matrix. As noted above, the influence vector notation $\overrightarrow{INF}$ indicates individual per-feature weight accumulations ($\vec{w}_f$) with respect to system decisions. To simplify analysis, feature influence can be grouped with respect to correct and incorrect system predictions.

$$\overrightarrow{INF}_{correct} = \frac{\sum^{TP} \vec{w}_f}{TP} - \frac{\sum^{TN} \vec{w}_f}{TN} \tag{3.5}$$

Equation 3.5 defines *Correct Influence* as the contributed weight towards correct decisions. The

numerical value of correct influence is the average weight contributed towards correct predictions. Correct Influence can be interpreted as a feature's confidence when deemed correct, providing unique insight alongside prediction accuracy.

$$\overrightarrow{INF}_{incorrect} = \frac{\sum^{FN} \vec{w}_f}{FN} - \frac{\sum^{FP} \vec{w}_f}{FP} \tag{3.6}$$

Similarly, Equation 3.6 defines the *Incorrect Influence* as the contributed weight towards incorrect decisions. Incorrect influence can be interpreted as a feature's overconfident contribution towards incorrect predictions. Overconfident features ultimately detract from an otherwise potentially correct prediction, a notably useful insight unique to influence. To improve interpretation, *Incorrect Influence* maintains the same directionality alongside *Correct Influence*, where weight contributions towards incorrect predictions are negative numeric values, while correct contributions despite being overruled are positive.

The negation of the *FP* and *TN* accumulations map all correct predictions into a positive influence and incorrect predictions into negative influence. Total influence can them be considered the sum of correct and incorrect influence as shown in Equation 3.7.

$$\overrightarrow{INF}_{total} = \overrightarrow{INF}_{correct} + \overrightarrow{INF}_{incorrect} \tag{3.7}$$

Ideally, features would remain impartial until sufficient correlation occurs to form a reliable prediction. In practice, features can be confidently wrong or even marginally correct while still providing value to the system. Independently, features often produce noisy and inaccurate predictions. Considered together with the hashed perceptron structure, feature noise is averaged away while simultaneous correlations combine to improve prediction quality.

The notable difference between MCC and Influence centers around the penalty attributed to mispredictions. MCC treats all feature predictions equally, regardless of the accompanying confidence. Rather, influence provides a perspective into the confidence towards correct decisions as well as overconfidence towards incorrect decisions. MCC provides a robust balanced accuracy,

where influence grants insight into contributions.

Features able to regulate their overconfidence are notably valuable to the system – even if they only occasionally grant unique perspectives. While there will inherently be inaccuracies in any history-based pattern correlation mechanism, understanding feature contributions towards correct/incorrect decisions helps during feature selection and tuning.

3.3.4.2 Inferring Information Gain

Information gain has been influential in decision trees [65] as a tool to estimate mutual information. Decisions trees leverage information gain as a means to estimate mutual information between competing branches when growing as well as trimming decision trees.

Inspired by information gain, we leverage a similar technique comparing the differential performance gain (or loss) between adjacent simulation runs. Algorithm 1 defines a simulation sweep across an ordered set of features. By incrementally adding features in ranked order, information gained by the added feature can be estimated through a resulting change in the simulation's performance metric. Plotting the improvement (or degradation) of each additional feature grants insight into any objectively useful information added to the system.

Algorithm 1 Simulation Feature Sweep

```
1: function SWEEP(R⃗)                          ▷ Simulation hit-rate sweep across feature ranking
2:     for n ← 1 to LENGTH(R⃗) do
3:         r⃗ ← {R⃗[0], ..., R⃗[n]}              ▷ Feature subset to include in simulation run
4:         P⃗[i] ← SIMULATE(r⃗)
5:     return P⃗                               ▷ Simulation performance metric in inclusive ranked order
```

This *feature sweep* estimation is particularly useful as both MCC and influence are inherently unable to identify shared information between competing features. Realizing that competing rankings can be compared by sweeping features, we stumbled on an approach to prune shared information. Algorithm 2 describes our approach to formalize our process of feature ranking in the context

of a hashed perceptron prediction system.

Algorithm 2 Differential Information Gain

```
1:  function IG(F⃗)                ▷ Inferring information gain through differential system improvement
2:      p_0 ← SIMULATE(F⃗)
3:      R⃗_0 ← MCC(p_0)                                  ▷ Initial ranking provided by MCC
4:      i ← 1
5:      while R⃗_i ≠ R⃗_{i−1} do                          ▷ Repeat until ranking is stable
6:          p⃗ ← SWEEP(R⃗_{i−1})
7:          q⃗ ← ADJACENTDIFFERENCE(p⃗)                  ▷ Calculate relative improvement gain
8:          R⃗_i ← SORT(q⃗)
9:          i ← i + 1
10:     return R⃗_i                                      ▷ New ranking sorted by relative improvement
```

Ranking features purely by information gain is tempting, but unfairly favors the features already selected over the differential new feature added. This algorithm greedily optimizes to the nearest local minimum. However, this technique is notably useful in eliminating mutual information from an already decent initial feature ranking provided by MCC.

3.4 Analysis

The primary evaluation goal of this work is to assess the viability of applying the Hashed Perceptron correlation mechanism to stateful flow table cache management. Contrasted with the cache optimality limit study in Section 3.2.2, this feasibility study provides insight into practically capturable locality.

3.4.1 Methodology

The simulator developed for this study allowed for quick exploration of cache management strategies in the context of stateful network flow tables. This work focused on cache hit-rate as the primary performance metric with the expectation that cache hit-rate translates directly into improved throughput and ultimately cache efficiency. While this cache hit-rate simulation does not model the latency intricacies incurred by cache misses as well as flow table management, it

does showcase relative prediction accuracy of competing cache management algorithms. Focusing on cache hit-rate simulation for the viability study allows for quick design iteration.

3.4.1.1 Network Traffic Datasets

It is tempting to leverage synthesized network traffic based on statistical models of packet arrival behavior [66, 67]. However, the random packet inter-arrival assumptions of these models would both hinder the cache management algorithm as well as introduce uncertainly around the accuracy of the synthesized traces. While these models can quite accurately resemble real inter-arrival flow behavior, actual packet arrival behavior is inherently complex and beyond our confidence to model. This viability study hinges on having representative datasets of network traffic to validate the premise of capturable temporal locality through more advanced cache management techniques.

For this study, we chose to leverage network packet capture (PCAP) traces rather than synthetic traffic in order to preserve temporal inter-arrival patterns theorized to be present in Section 3.2.3. CAIDA's offers passive PCAP traces within their Equinix internet exchange[5] data centers. We leveraged PCAP traces from CAIDA Equinix Sanjose, Chicago, and NYC datasets spanning 2012 to 2018 [1]. Traffic flowing through these exchange points should contain a representative sampling network traffic commonly found on the Internet. CAIDA's Equinix datasets were chosen primarily because of the availability and diversity of the traffic provided and available to researchers.

CAIDA Dataset	Equinix Sanjose January 2012
Interface	10 Gbps link (6 Gbps average, bi-directional)
Packet Rate	1M / second
New Flows	30k / second
Active Flows	300k over 4-second window
Trace Length	5 minutes

Table 3.5: Sample of CAIDA Packet Arrival Statistics

[5]Internet Exchange Points (IXPs) are crucial infrastructural components of the Internet where network operators (peers) exchange traffic.

The CAIDA Equinix PCAP traces utilized for this study are captured on 10 Gbps links and taken over one hour intervals, split into 1 minute files. Table 3.5 provide a summary of packet arrival and flow turnover statistics. The CAIDA PCAP files contain pseudo-anonymized network layer IPv4 addresses, preserving subnets while anonymizing exact addressing. Because of the inherent complexities involved, we believe actual traces of representative network traffic to be crucial in exploring the viability of network flow table cache management.

3.4.1.2 Cache Simulation

The simulator models a single inline device maintaining a stateful flow table such a firewall. Maintaining this stateful flow table is the critical-path for achievable throughput. Caching techniques are leveraged to improve device throughput; this study focuses on improving the approach of cache management by borrowing techniques developed within the the computer microarchitecture community.

Flow Cache Entries	4k flow entries
Cache Associativity	8-way
Feature Tables	5 features, each 64k entries 5-bit saturating counters
Feedback History	8 predictions / associative-set
Feature Selection Dataset	CAIDA Equinix Sanjose January 2012
Validation Datasets	CAIDA Equinix Sanjose March 2012 CAIDA Equinix Chicago March 2014 CAIDA Equinix Chicago March 2016 CAIDA Equinix NYC March 2018

Table 3.6: Hashed Perceptron Cache Simulation Parameters

Table 3.6 summarizes the cache simulation parameters as well as feature selection and validation datasets. The simulator replays PCAP files, preserving relative arrival order to maintain packet inter-arrival patterns. The packet headers are parsed, stateful flow identifiers maintained throughout the simulation to track and evaluate cache management algorithm's relative performance. In

addition to cache hit-rates, the simulator gathers evaluation metrics helpful to gain insight during early design exploration of the *Flow Correlator* cache management mechanism described in Section 3.3.2.

We first provide insight into our feature exploration, evaluating several competing feature ranking mechanisms in Section 3.4.2. Second, we evaluate the robustness of the final selected features across datasets in Section 3.4.3. In Section 3.4.4 we infer changes in cache entry lifetime by analyzing cache efficiency. Finally, Section 3.4.5 recaps some of the dynamics between the chosen features in the context of bypass and reuse cache management predictions.

3.4.2 Feature Exploration

While performance estimation is the ultimate goal of any architectural design-space exploration, quickly ranking useful features early in the design process is immensely valuable. Feature design, ranking, and selection is a time-consuming component of design exploration – adding yet another dimension to an already complex optimization problem.

Since nearly all of the features explored in Table 3.3 perform poorly in isolation, it is crucial to compare feature performance as a system. Further, it becomes non-trivial to select the best combinations of feature components as candidate combinations inherently share mutual information. Therefore, there is a need for a reliable ranking mechanism that allows the architect to identify top performing feature combinations early in the design phase.

The two competing feature ranking techniques analyzed in this section are Matthews Correlation Coefficient (MCC) and Total Influence (INF). Randomly ordered features (RND) were also included to gauge the effectiveness of the mechanisms in context. Finally, we describe a differential analysis technique developed in this work to trim mutual information, resulting in the final feature ranking.

3.4.2.1 Feature Design

Influence was discovered out of a desire for feedback during feature design. In particular, hints at strengths and weaknesses of competing features help identify successful mixing strategies early

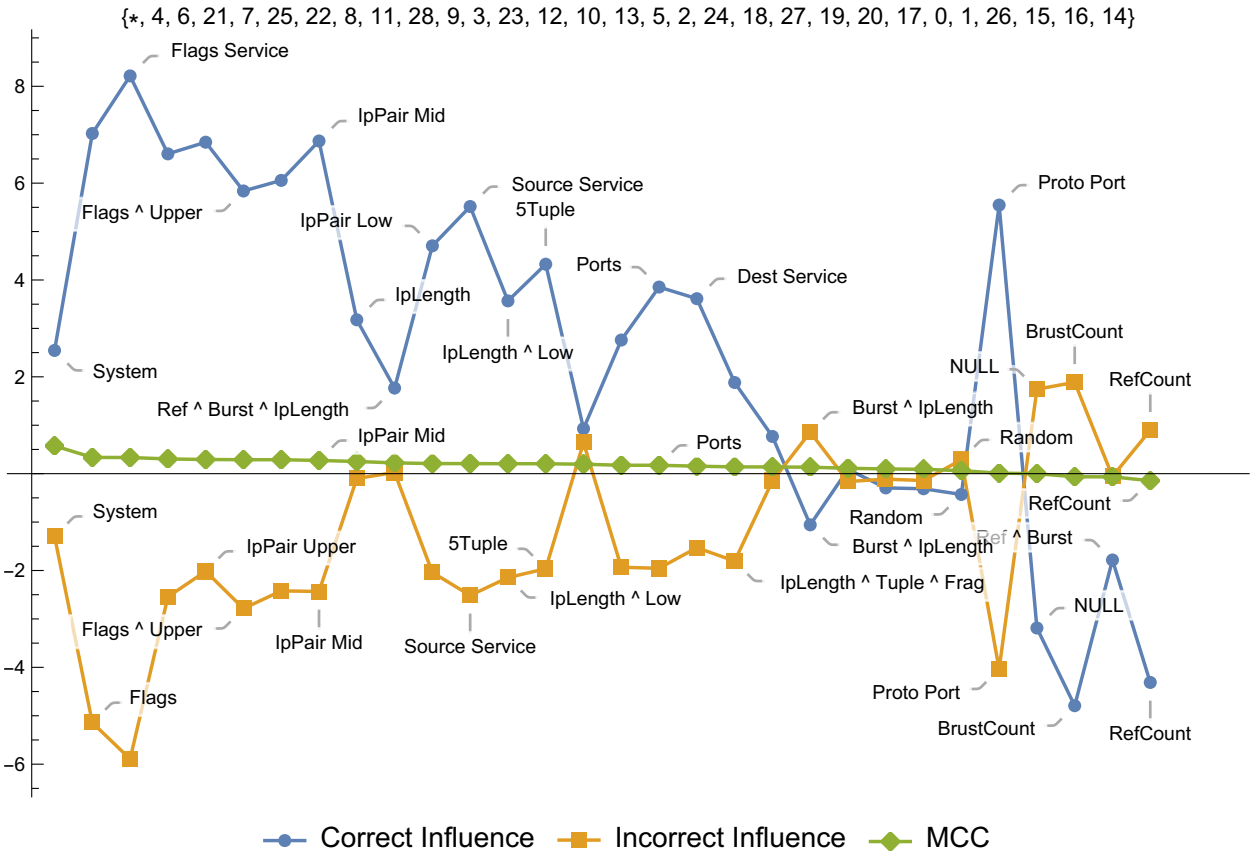


Figure 3.10: Feature Influence on Reuse Predictions

in the design exploration. The intuition behind Influence centers around interpreting the feature's correlation output (weight) as a confidence. Influence then contrasts these contribution confidences against the overall correctness of the system, grouped by prediction type (Reuse or Bypass).

Figure 3.10 provides insight into feature contributions towards flow entry reuse predictions, relative to MCC's balanced prediction accuracy. Per-feature MCC and Influence metrics are tracked across all candidates in a single Hashed Perceptron *Flow Correlation* simulation. Features are sorted in decreasing MCC order with the corresponding *Correct Influence* and *Incorrect Influence* presented for relative comparison.

The first entry, *System*, represents the overall flow entry reuse prediction exhibiting a notable improvement in MCC compared to any individual feature. The contributions towards correct predictions (*Correct Influence*) as well as contributions towards incorrect predictions (*Incorrect In-*

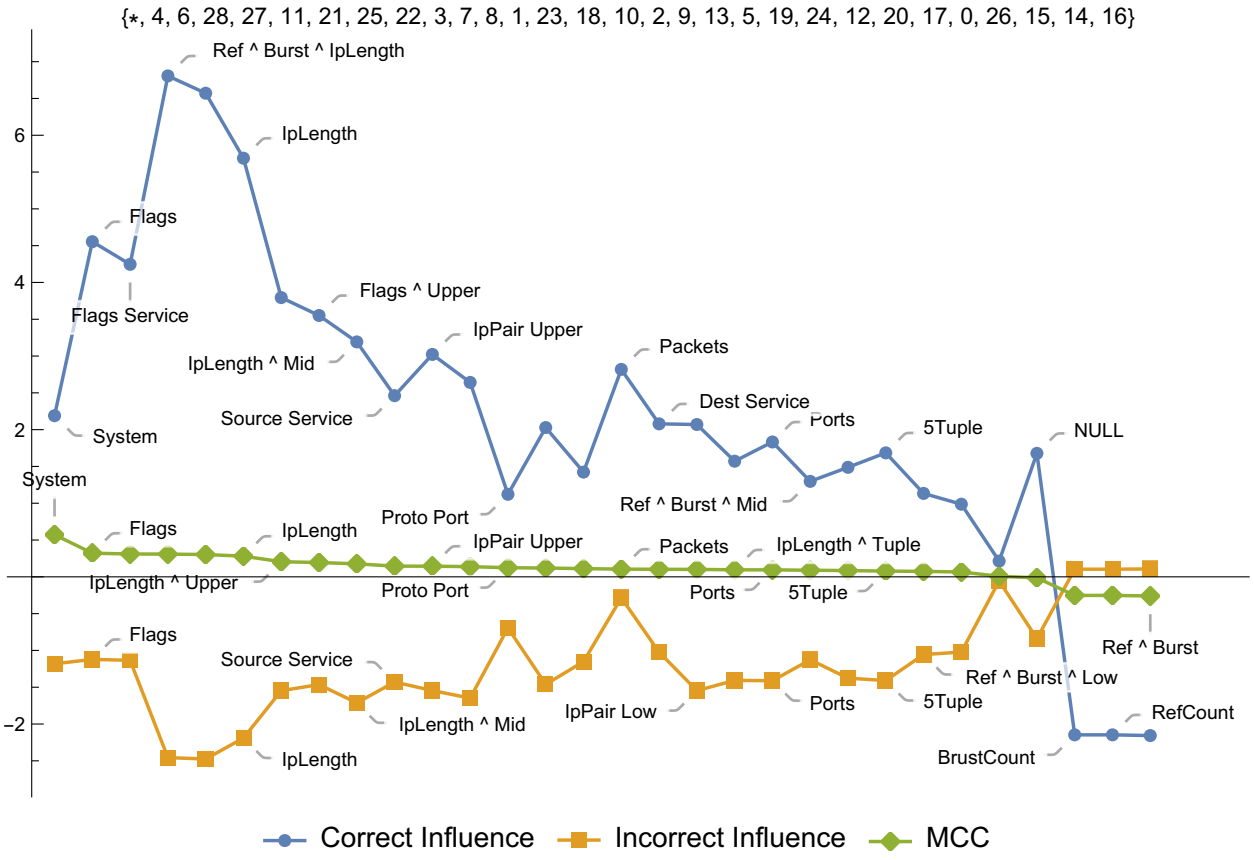


Figure 3.11: Feature Influence on Bypass Predictions

fluence) represent a feature's influence over the predictor. The magnitude represents confidence while the sign represents a positive (or negative) influence on predictor outcomes.

Interestingly, features exhibiting high confidence towards correct predictions tend to also be accompanied by overconfidence towards incorrect predictions. Overconfidence can be tolerated when coupled with a high prediction accuracy (MCC); however, there is a strong preference for features that provide value while also minimally contributing to mispredictions as represented by *Incorrect Influence*.

Figure 3.11 provides insight into feature contributions towards flow entry bypass predictions relative to MCC. It is intuitive that features have strengths towards either Bypass or Reuse predictions. Unlike reuse flow entry predictions, bypass predictions inherently lack some of the temporal indicators tracked in cached flow table entries.

Bypass and Reuse predictions both aim to predict flow entry reuse within the approximation of the useful cache working set. However, they are distinct predictions as reuse probability of a cached and uncached flow table entry are not necessarily equivalent. Bypass predictions occur not only at the start of new flows, but also on transitions from *Dormant* to *Active* for already established flows. Reuse predictions aim to identify *Active* connections amongst established flows, triggering early eviction if predicted to transition from *Active* to *Dormant*.

Burst Count and Reference Count (Pure features f_{14} and f_{15}) exhibit a perplexing anti-correlation as seen by both MCC and Influence. It is notable that Burst and Reference Count rely on unrolled and allowed to decouple from global patterns. Intuitively, it makes sense that temporal patterns need to be grouped (unrolled) in some way to enable correlations. During feature design, Influence enabled insight into how each feature fell into natural cache management roles.

3.4.2.2 Feature Ranking

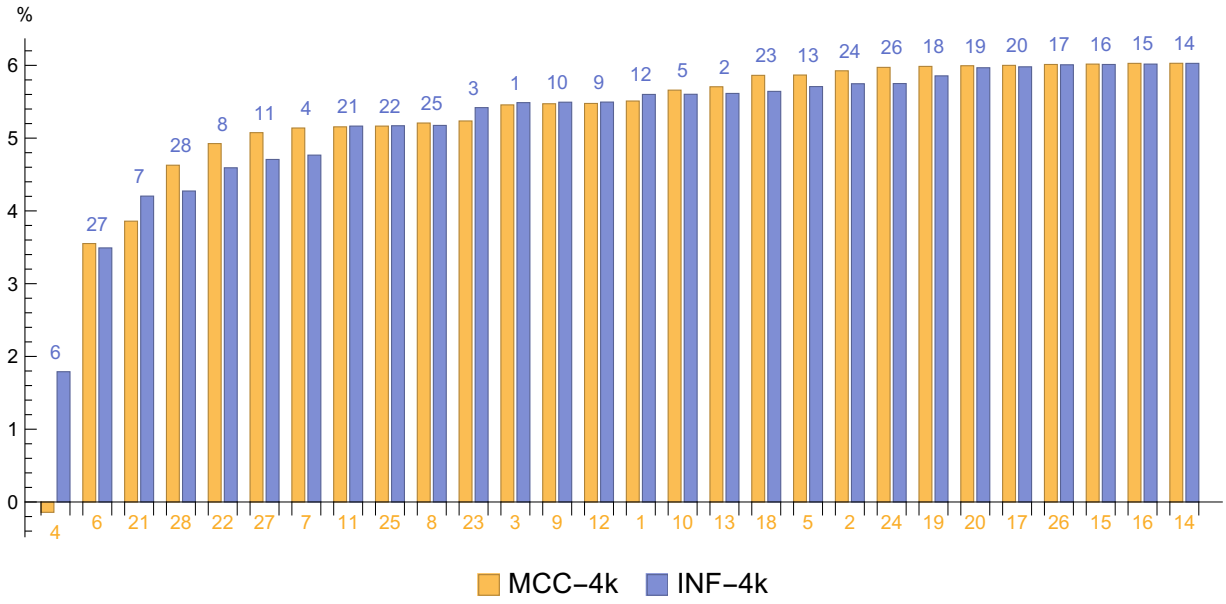


Figure 3.12: Initial Feature Ranking

Hit-rate improvement over baseline LRU comparing MCC-4k below bars in yellow to INF-4k above in blue.

Figure 3.12 shows the relative hit-rate improvement over LRU as baseline. *Total Influence* as defined in 3.7 was used for INF-4k's ranking. The plot is generated by incrementally including features in ranked order using Algorithm 1. This *sweep* plot shows the incremental gain in hit-rate by including additional features as a useful means to compare feature rankings. Spanning from a single feature (left) to all features (right), diminishing returns can be easily identified.

Both MCC and INF provide a relatively similar ranking, with a few notable differences. In particular MCC's top choice, f_4 consists of just TCP flags is unable to stand alone. INF's top choice, f_6, consists of a mix of Port, Protocol, and TCP/IP flags, covering f_4. As a single-table predictor, INF's f_6 manages to provide a modest 2% hit-rate improvement over baseline LRU, while MCC's f_4 actually degrades hit-rate. Ultimately MCC provides a better all-rounded ranking; however, INF does identify a few notably valuable features before MCC.

To help put into perspective the utility of MCC as a feature ranking mechanism, we compare against four randomized rankings. As expected, Figure 3.13 showcases the stunted performance of random rankings amongst features with significant permutation overlap. There are a few notable observations that highlight the behavior of a Hashed Perceptron based predictor.

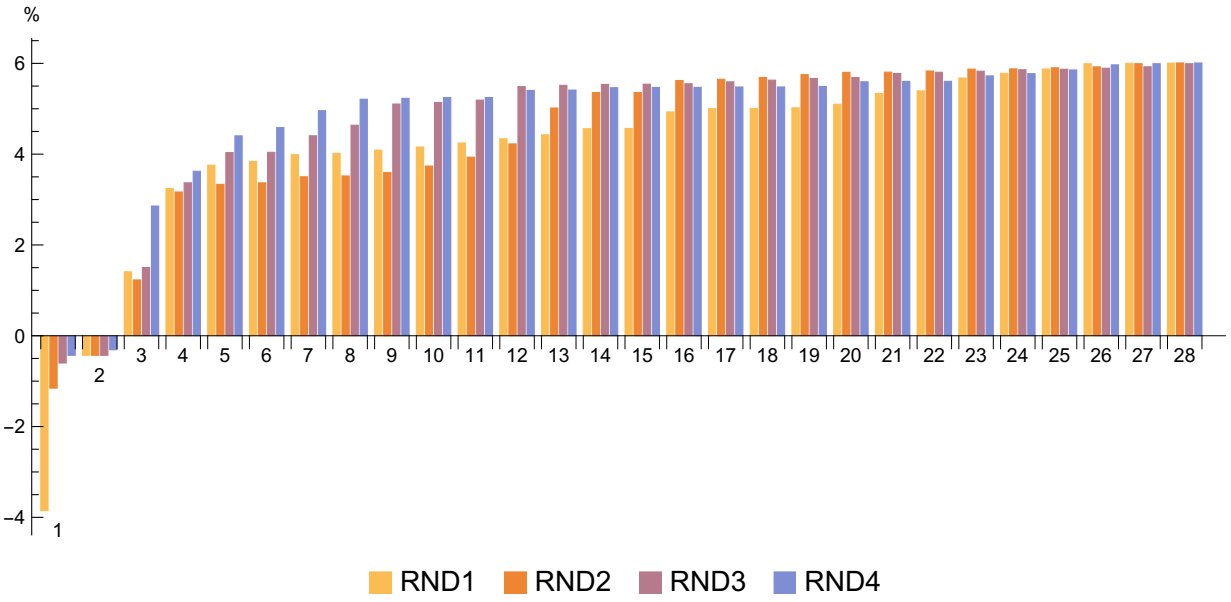


Figure 3.13: Random Feature Ranking

The first notable observation is that predictor effectiveness under-performs baseline LRU until the third randomly selected feature is added. The situation flips with the third random selection able to surpass baseline; achieving half the potential hit-rate improvement by the fourth random selection.

As expected, there is significant shared information across the features in Table 3.3 due to feature permutation similarity. The goal of the ranking algorithm is to identify the best combinations in order to hasten designer iteration. It is clear that MCC and INF are both able to identify a better initial ranking than random selections. However, MCC and INF are both unable to account for shared information, looking only at the feature's own track record in isolation.

3.4.2.3 Iterative Information Gain

Relying on the designer to select the best choice is untenable as system complexity grows. Ultimately we need a method to rank features by incremental value contributed to the system. Inspired by information gain decision tree learning algorithms, we outline our approach to both measure a feature's actual contribution to the system as well as take into account mutual information between features.

Our iterative information gain method outlined in Section 3.3.4.2 improves a MCC ranking by differential analysis sweep. The feature sweep analysis plot is differential analysis, where hit-rate is plotted as features are incrementally added until all features are included. Figure 3.14 shows the incremental change in hit-rate as features are incrementally added to the predictor in MCC-4k rank-order.

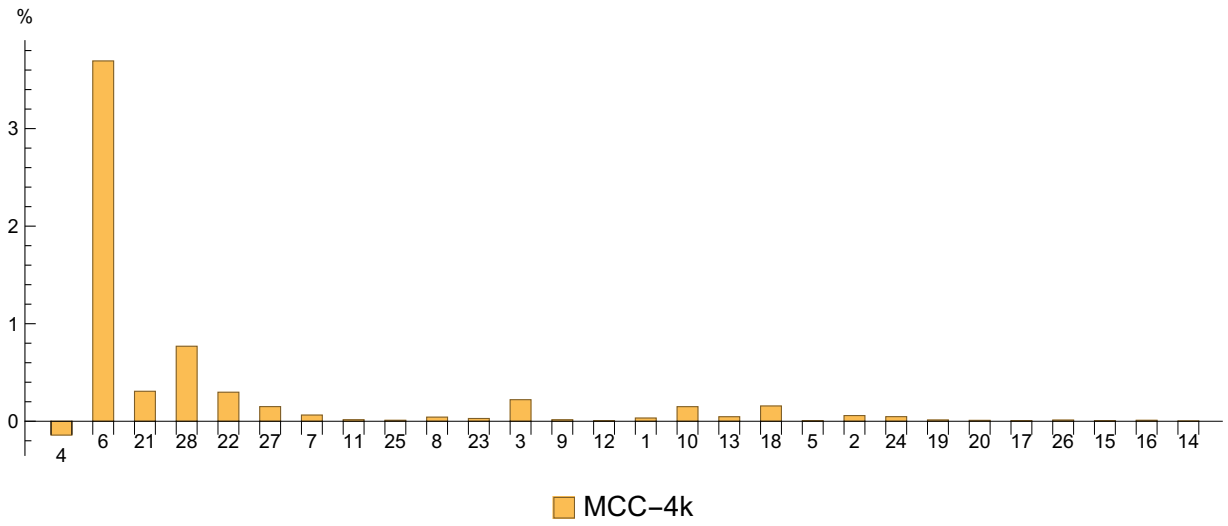


Figure 3.14: Differential Improvement

Per-feature hit-rate differential gain (adjacent difference) of MCC-4k.

Figure 3.14 shows the change in hit-rate as features are added to the predictor from left to right. While early features are likely to bring a larger gain, the goal is to minimize shared information between features.

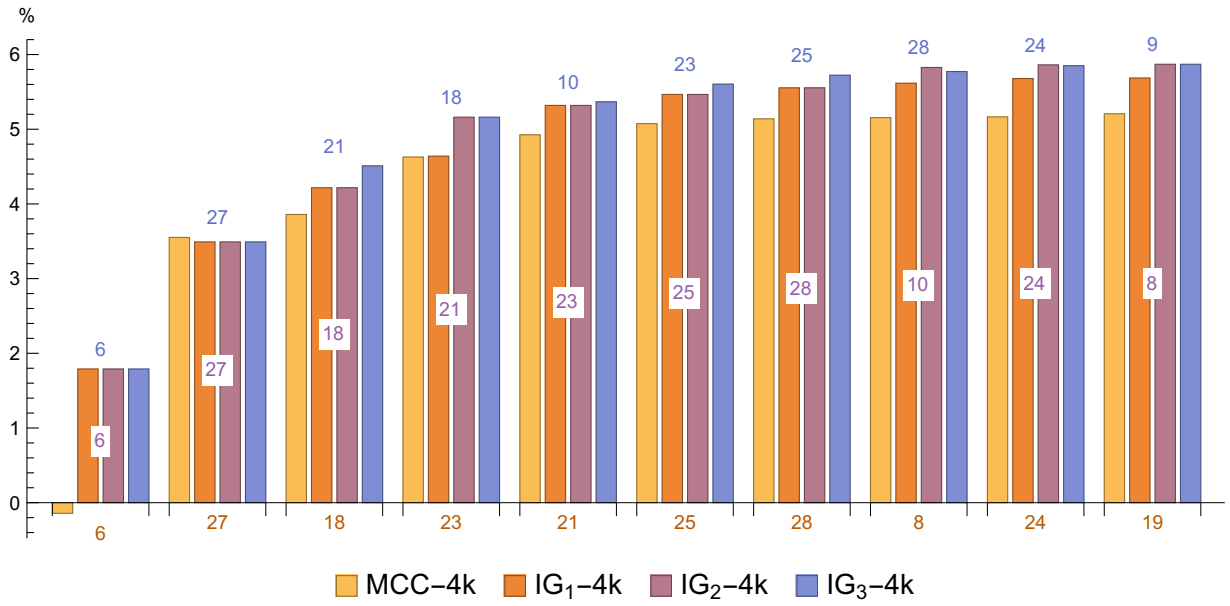


Figure 3.15: Iterative Information Gain Ranking Improvement

Figure 3.15 shows the iterative improvement from the initial *MCC-4k* ranking through three iterations of differential information gain. To improve readability, this figure just showcases the hit-rate improvement for the first ten features. *IG$_1$-4k* through *IG$_3$-4k* represent the improvement in ranking through each differential Information Gain (IG) iteration.

The first notable adjustment made in the first pass (IG$_1$) is identifying that f_6 covers f_4 entirely and translates to a hit-rate improvement as the first feature, despite f_4 achieving a marginally higher prediction accuracy as inferred by MCC.

It is also notable that f_{10} and f_{18} were both overlooked by MCC and INF implying a relatively low prediction accuracy. However, both propagated to the top five ranking in the subsequent two iterations (IG$_2$ and IG$_3$). This differential information gain analysis identified that f_{10} and f_{18} provided notable value to the system, despite the lower prediction accuracy.

The final notable adjustments were marginal, preferring to unroll across upper IP addresses rather than middle or lower variants (f_{18} and f_{21}).

This iterative information gain technique is best fit to trim shared information from an already

decent ranking. While not showed here, attempts to iterate from a random feature selection using just the iterative information gain technique was computationally prohibitive.

We found Influence to be a particularly useful during feature creation, MCC best suited for initial feature ranking, and differential information gain notably helpful to narrow in on the most informative features. Automating feature selection, even if imperfect, allows the system designer to focus on feature creation – in particular introducing novel feature combinations and information sources.

3.4.3 Improvement Validation

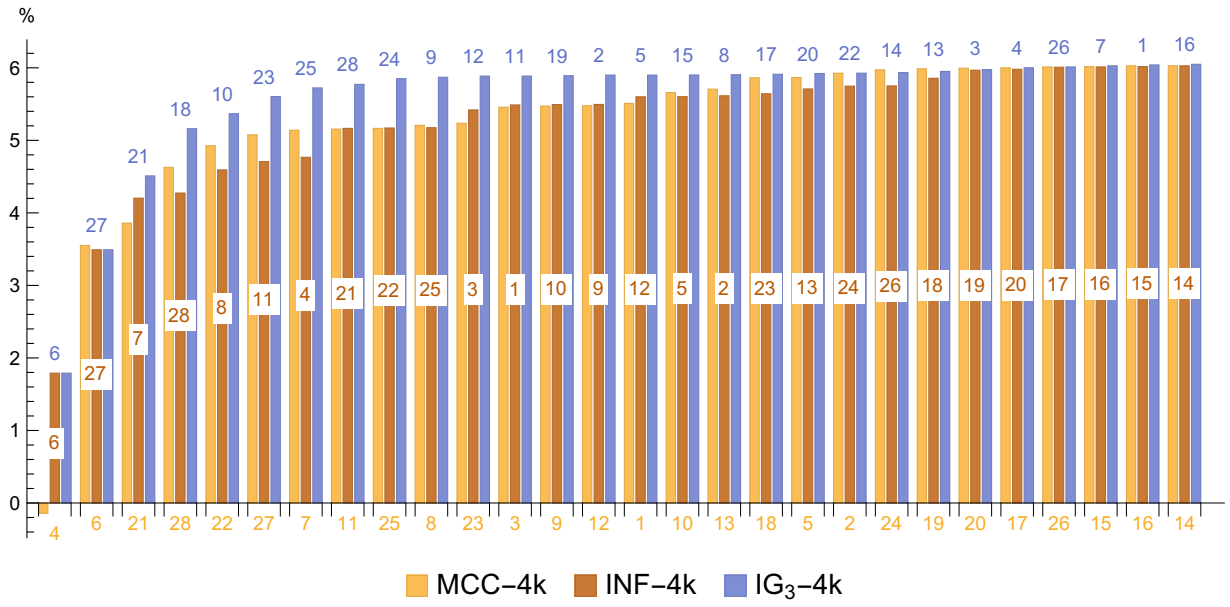


Figure 3.16: Final Feature Ranking Comparison

Figure 3.16 recaps the ranking improvement on the training dataset. Diminishing returns in achievable hit-rate improvements started to occur roughly between four and seven features. For validation, we decided to select the five best features from each ranking.

$f_{\#}$	**Feature Type**	**Feature Components**
6	Pure Mix	$f_{TCP} \ll 7 \wedge ipProto \wedge min(srcPort, dstPort)$
27	Pure + Pattern	$\{ipLength, \lceil burstCount \rceil_8\}$
21	Pure + Pattern	$ipLength \wedge (ipv4Dst \wedge ipv4Src)[31{:}16]$
18	Pure + Pattern	$\{\lceil refCount \rceil_8, \lceil burstCount \rceil_8\} \wedge (ipv4Dst \wedge ipv4Src)[31{:}16]$
10	Pattern	$\lceil flowPackets \rceil_{16}$

Table 3.7: Selected Features (IG_3-4k)

Optimized on flow cache with 4k entries, 8-way set associativity using the CAIDA Equinix Sanjose January 2012 dataset (equinix-sanjose.20120119) during feature selection and ranking.

Table 3.7 list the top five features in ranked order of improved system value after the third iteration of differential information gain (IG_3-4k). It is interesting to note that all *Pure* information sources were utilized in some form in the final set of features. While it is not surprising that protocol and TCP/IP flags proved to be a naturally useful reuse hints, it is certainly interesting that f_6 was preferred over the f_4. A similarly useful reuse hint comes from deviations in IP frame length. While both MCC and INF showed high utility of frame length (f_{11}) as a *Pure* indicator, IG_3 found significantly higher utility by combining frame length and BurstCount (f_{27}).

It appears that both frame length and BurstCount contributed uniquely as a reuse prediction hint, both notably providing additional value being unrolling over upper IP addresses (f_{18} and f_{21}). With IP address combinations seemingly contributing little insight into reuse as *Pure* sources, it is interesting that both frame length and BurstCount found extra utility through unrolling. Appendix A.3 provides further insight into selected features' weight distributions alongside the contributing feature components.

3.4.3.1 Validation Datasets

While it is clear that there is notable improvement performing IG passes within the training dataset, it is crucial that this process did not over-optimize feature selection. The training set consisted of five minutes of PCAP, bi-directional from January 2012 taken at CAIDA Sanjose exchange. The validation set consisted of five minutes of PCAP, bi-directional from March of 2012, 2014, 2016 and 2018 spanning the three available CAIDA exchange locations: Sanjose,

Chicago, and NYC. The validation set will be used to compare the top-five features provided by the MCC-4k, INF-4k, and IG_3-4k rankings.

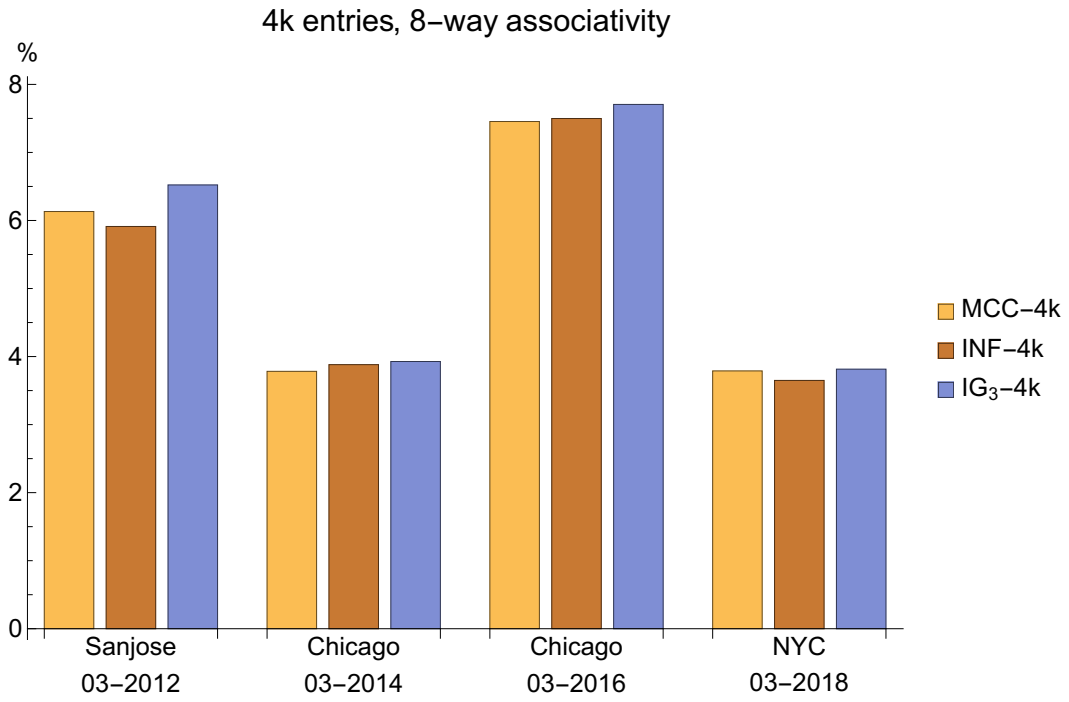


Figure 3.17: Improved Hit-Rate Validation

Figure 3.17 shows consistent improvement across all validation sets. Hit-rate improvements ranged between 4% and just under 8% over baseline LRU. The validation datasets showed no degradation from MCC-4k to IG_3-4k, indicating that iterative Information Gain did not over optimize. However, we suspect IG's greedy optimization requires generality in feature creation – best suited to remove mutual information rather than select the most robust features.

These results attest to the generality of the features chosen, providing value to a generic cache management reuse predictor beyond the original dataset location. It is certainly notable to demonstrate viability towards a flow table cache management approach which translates beyond the original optimization parameters.

3.4.4 Cache Efficiency

Tracking the lifecycle of cache entries can be helpful to understand how cache management techniques impact cache behavior. Figure 3.18 depicts a cache entry's lifecycle from insertion (t_0) to last-access time (t_L) and finally the eviction time (t_E).

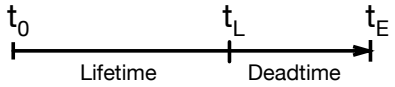


Figure 3.18: Cache Entry Lifecycle
t_0: insertion time, t_L: last-access time, t_E: eviction time

Equation 3.8 defines a cache line's *lifetime* as duration between insertion and last access.

$$lifetime = t_L - t_0 \tag{3.8}$$

Similarly, *deadtime* is the duration between last access and eviction as defined in Equation 3.9.

$$deadtime = t_L - t_E \tag{3.9}$$

Finally, Equation 3.10 defines cache line *efficiency* as the useful *lifetime* over the total time the cache entry was occupied.

$$efficiency = \frac{t_L - t_0}{t_E - t_0} \tag{3.10}$$

Tracking the *lifetime*, *deadtime*, and *efficiency* for all cache lines throughout the simulation provides some insight into the relative impact of cache management algorithms on overall caching behavior.

Figure 3.19 compares the cache line *lifetime* during a cache simulation over a one minute

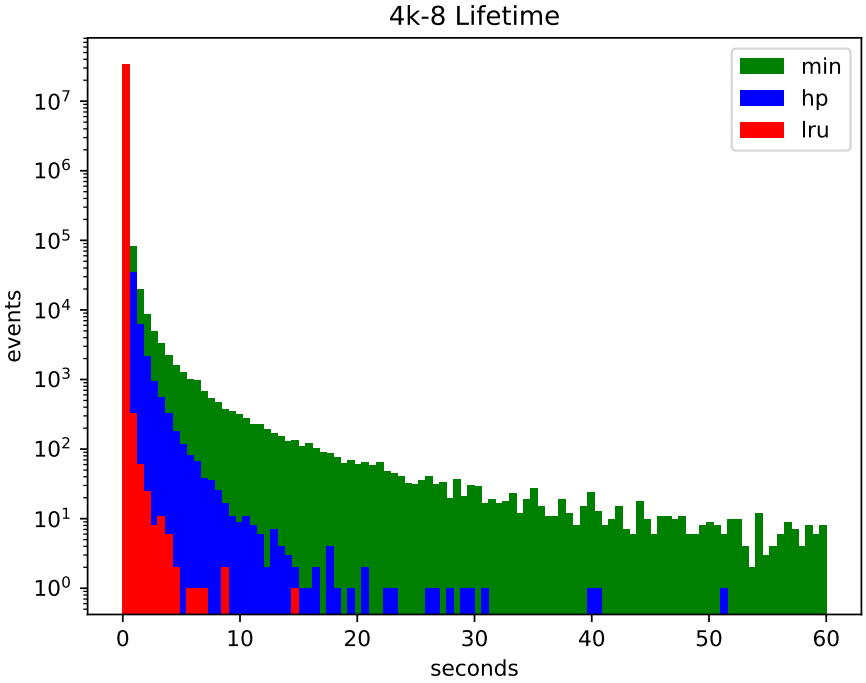


Figure 3.19: Cache Entry Lifetime

pcap trace[1]. While the vast majority of flows entries have a short lifetimes in the cache, the *Flow Correlator* Hashed Perceptron cache management technique (hp) more closely resembles Belady's MIN (min). Holding onto certain bursty flows for a longer duration between pauses potentially reducing entry turn-over.

Figure 3.20 shows the corresponding *deadtime*, implying that the hashed perceptron technique simultaneously has both more and less patience. A significant number of entries are evicted very early (within 10ms), while simultaneously waiting on average twice as long as LRU for others (upwards of 200ms). This adaptability allows hp to triage flows with low probability of reuse, while simultaneous willing to increasing entry *deadtime* in hopes of also enabling an increase in entry *lifetime*.

Figure 3.21 outlines how the hashed perceptron approach generally smoothed out cache *efficiency*. LRU's inherent requirement to wait for demotion from MRU to LRU limits cache effi-

[1]CAIDA Equinix Sanjose January 2012 pcap trace: equinix-sanjose.20120119.

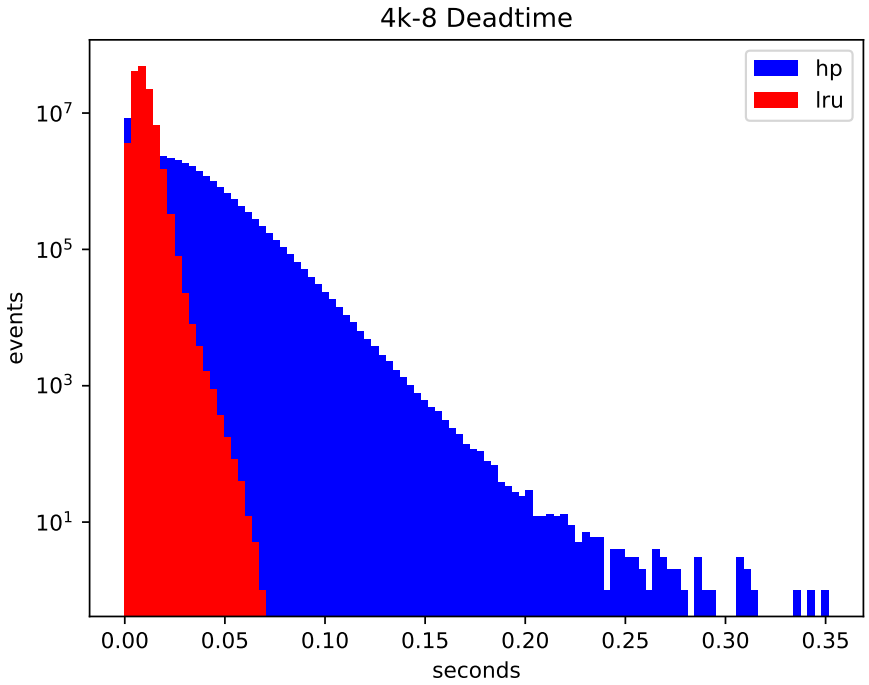


Figure 3.20: Cache Entry Deadtime

ciency. Meanwhile, the hashed perceptron technique is a step towards both triaging flows with a low probability of reuse as well as a tendency to hold onto cache entries for a longer duration.

Cache *efficiency* doesn't grant perspective into the quantity of hits over the entry *lifetime*, just that the entry had reuse. Coupled with an improvement to hit-rate, this cache line efficiency analysis confirms that there is a worthwhile trade-off to holding onto certain flows longer as well as triaging flows early.

3.4.5 Feature Roles

Throughout the feature selection process, we noticed a few noteworthy trends in roles assumed by each feature. Observed primarily through feature influence described in Section 3.3.4.1, features tended to produce a certain prediction bias as measured by average output prediction weight. Originally described as a form of optimism, features tended to vote for eviction (pessimistic), while others tended to lean towards reuse (optimistic). What was observed as a natural feature bias

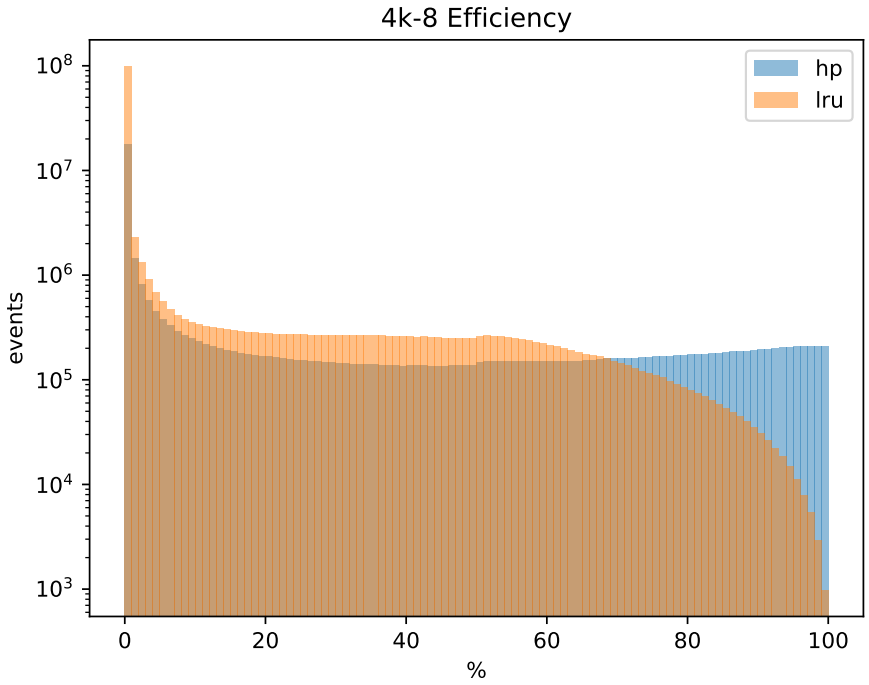


Figure 3.21: Cache Entry Efficiency

towards a particular outcome, in actually reflects the natural predictive utility of the feature.

Certain features are better indicators of early eviction – hinting at a flow's transition from *active* to *dormant*. The temporal pattern feature components (f_{10}, f_{14}, and f_{15}) are clear contributes to bypass role. Other features are best suited to triage incoming flows during bypass – hinting at flow transitions from *dormant* to *active*. Indications of protocol behavior from several pure feature components (f_4, f_6, and f_{11}) were notable contributions to the reuse role.

These two cache management roles (bypass and reuse) tend to uniquely favor subsets of feature components. In nearly all cases, we noticed an overall benefit in combining feature components with orthogonal roles. Additionally, we also noticed that certain orthogonal combinations (f_{27}, f_{28}) resulted in improved predictions accuracy and confidence for both bypass and reuse. IP frame length (f_{11}), while valuable as a pure feature component contributing to both roles, strengthened nearly any feature component it was combined with.

As an artifact of packetizing flows into manageable units bounded by Maximum Transmission

Unit (MTU), IP's frame length field was a notable indicator of packet inter-arrival patterns. It is further noteworthy that while f_{11} was able to stand alone, it also significantly strengthened when combined with role-focused feature components. While not included in this study, the interplay between frame length and received packet size (only differing if IP fragmentation has occurred), is expected to be noteworthy low-hanging fruit in further feature design explorations. The final feature ranking pass of our differential Information Gain approach preferred the information density of role-orthogonal feature components.

3.4.6 Automatic Throttling

One of the critical aspects of cache management is the ability to dynamically adapt prediction aggressiveness in response to changing cache pressure. While not quite hardened against adversarial attacks, the Hash Perceptron flow correlation mechanism presented in this work exhibits inherent self-throttling behavior which adapts to changes in cache pressure. The correlation feedback mechanism reinforces likely good predictions, while predictions with less probability of being correct naturally tend to regress to the natural bias of the system. See Appendix A.4 for additional cache efficiency analysis providing insight into cache pressure dynamics.

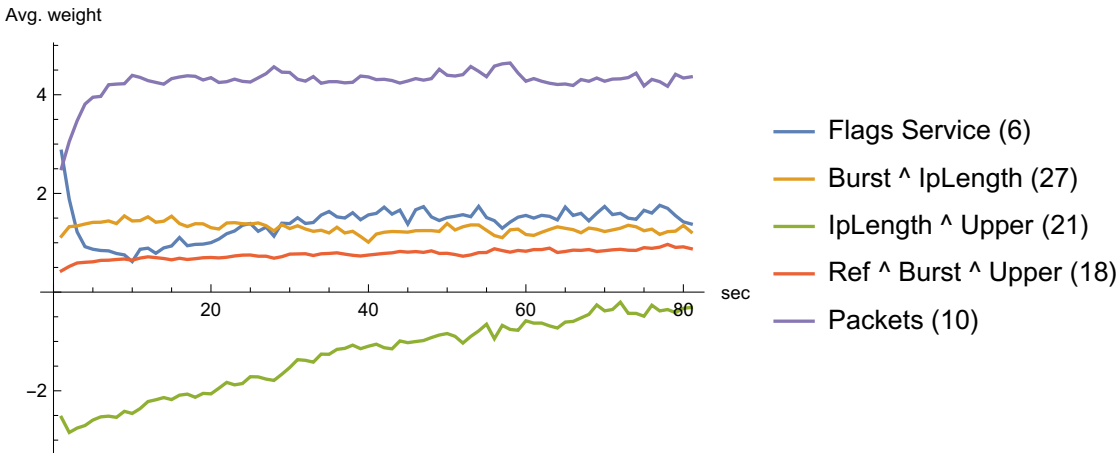


Figure 3.22: Feature Bias

Figure 3.22 shows the feature's prediction bias over epochs of each evaluated feature. Naively,

we would desire all features to be neutral biased; however, this isn't aligned with the objective of discriminating flows based on estimated short-term activity. Features assume a role implicit by the construction of their components, producing a bias skewed towards the utility of their insight. Pessimistic features (net bias of evict) primarily contribute to the role of bypass, effectively filtering flows from entering the cache until sufficient optimism arises from other features. Optimistic features (net bias of keep) primarily contribute to reuses prediction. The adaptive nature of hashed perception mechanism dynamically adjusts each feature to the average observed cache pressure.

3.5 Summary

Cache management for stateful flow tables is a largely unexplored area of research, despite their wide deployment. This work demonstrates viability of the Hashed Perceptron approach to cache management. The Hashed Perceptron correlation technique is uniquely able to cope with noisy information sources, well suited for network flow table cache management.

Feature creation is still an designer's art coupled with both intuitive and counter-intuitive surprises. Early feedback and iteration is essential to enable fast design exploration. Advancements to systematic feature ranking and selection, even if non-optimal, allow designers to focus on system architecture and feature creation first and foremost. Ultimately, any ML approach is inherently reliant on effective feature creation and information sources.

The Hashed Perceptron technique is proving to be an interesting lightweight ML approach where weights are run-time correlations, enabling adaptability beyond comparable offline techniques. When applying the Hashed Perceptron technique to a new domain it became apparent how a multi-feature, consensus-based system significantly improves prediction reliability compared to hand-crafting complex heuristics. A interesting observation in the context of network traffic is the resiliency of the Hashed Perceptron technique to cope with noisy or even misleading features. While it is certainly possible to degrade performance, the Hashed Perceptron's ability to combine multiple features offers stability over any single perspective technique.

Few mechanisms allow combining independent correlations reliably, yet efficiently to improve system performance. This work describes the process of adapting and evaluating the Hashed Per-

ceptron mechanism to an unexplored domain, where the input patterns and vectors available to features bring unique challenges.

4. CONCLUSIONS

As the complexity of networking and network functions increase, data planes are becoming more susceptible to DoS attacks which can result in underlying network reliability and availability concerns. Thus, there is a strong need for data plane architectures, particularly in the context of SDN, that continue to operate efficiently in the presence of malicious traffic. The first technique presented leverages a Bloom Filter to prioritize established traffic and prevent malicious starvation of expensive packet classification resources. This *Pre-Classification* technique is general enough to be considered for any classification pipeline with non-uniform processing requirements.

The second technique explored, originally developed for speculative microprocessors, adapts the Hashed Perceptron binary classifier to flow table cache management. The proposed *Flow Correlator* mechanism leverages the Hashed Perceptron to correlate flow activity with temporal patterns and transport/network layer hints. The *Flow Correlator* technique was able to demonstrate a consistent hit-rate improvements across all validation sets.

This dissertation explored how temporal locality present in flow behavior can be leveraged to improve both the reliability and performance of packet processing pipelines. Providing a unique complement to flow table caching, *Pre-Classification* improves reliability by prioritizing established traffic when classification resources are strained. Amenable to hardware implementations, both *Flow Correlator* and *Pre-Classification* techniques show promise in improving the reliability and performance of flow-centric packet processing architectures.

4.1 Future Work

Exploring graceful Bloom Filter clearing strategies for *Pre-Classification* would help reduce or eliminate the impact of re-learning flows immediately following cold clearing. Using two Bloom Filters with out-of-phase clearing intervals would noticeably help smooth the transition. Alternatively, clearing subsets of a single Bloom Filter's memory arrays in a cyclic manner similar to memory row refreshing may also be sufficient to maintain false positive rate while reducing the

impact from clearing. Further research is needed to understand these trade-offs in a time-decaying Bloom Filter mechanism.

While flow table cache management driven by the Hashed Perceptron binary classifier shows promise, further research is needed to close the gap between feasibility and practical implementations. We believe there remains notable room for improvement beyond the achievable hit-rates demonstrated in this study. The features designed in this exploration were a first pass in order to understand dynamics and viability of applying the Hashed Perceptron binary classifier to flow table cache management. It is clear that follow-on investigations are needed to quantify realizable improvements from advanced cache management techniques.

Additionally, it would be interesting to explore a bounded Belady's algorithm as ground truth and training feedback for a Hashed Perceptron cache management system in a manner similar to Hawkeye [59]. While not leveraged in this exploration, genetic algorithms applicable to automating feature design. Further exploration is needed to improve the iterative information gain technique to remove mutual features alongside partially automated feature creation techniques.

4.2 Need for Standardized Network Benchmarks

Unlike in processor microarchitecture research where standardized test suites are available and well understood, networking research lacks a comparable structure for caching research. One of the critical components of this work was exploring patterns present in real traffic, opposed to attempting to fully capture all the real-world dynamics in synthetic traffic generation. The processor microarchitecture community has enjoyed competitive advancements in caching and branch prediction predictor accuracy enabled by community recognized standardized benchmarks. The networking community would benefit from open competitions enabling improvement of flow table cache management algorithms.

REFERENCES

[1] "The CAIDA UCSD Anonymized Internet Traces - 2012, 2014, 2016, 2018, kc claffy, Dan Andersen, Paul Hick."

[2] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, (New York, NY, USA), pp. 99–110, ACM, 2013.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, July 2014.

[4] B. Wheeler and L. Wirbel. sixteenth ed., 2015.

[5] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vSwitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 117–130, USENIX Association, May 2015.

[6] Y.-K. Chang and H.-C. Chen, "Fast Packet Classification using Recursive Endpoint-Cutting and Bucket Compression on FPGA," *The Computer Journal*, vol. 62, pp. 198–214, 06 2018.

[7] H. Song, "Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, (New York, NY, USA), pp. 127–132, ACM, 2013.

[8] Open Networking Foundation, "Openflow switch specification." `opennetworking.org/software-defined-standards/specifications`.

[9] C. J. Casey, A. Sutton, and A. Sprintson, "tinyNBI: Distilling an API from Essential OpenFlow Abstractions," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, (New York, NY, USA), pp. 37–42, ACM, 2014.

[10] H. Wang, G. Yang, P. Chinprutthiwong, L. Xu, Y. Zhang, and G. Gu, "Towards fine-grained network security forensics and diagnosis in the sdn era," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, (New York, NY, USA), p. 3–16, Association for Computing Machinery, 2018.

[11] A. Shaghaghi, M. A. Kaafar, R. Buyya, and S. Jha, *Software-Defined Network (SDN) Data Plane Security: Issues, Solutions, and Future Directions*, pp. 341–387. Cham: Springer International Publishing, 2020.

[12] P. Zhang, H. Li, C. Hu, L. Hu, L. Xiong, R. Wang, and Y. Zhang, "Mind the gap: Monitoring the control-data plane consistency in software defined networks," in *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, (New York, NY, USA), p. 19–33, Association for Computing Machinery, 2016.

[13] A. Voellmy, J. Wang, Y. Yang, B. Ford, and P. Hudak, "Maple: Simplifying sdn programming using algorithmic policies," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pp. 87–98, 2013.

[14] C. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," in *Proceedings of the 41st annual ACM Symposium on Principles of Programming Languages*, pp. 113–126, 2014.

[15] A. Guha, M. Reitblatt, and N. Foster, "Machine-verified network controllers," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pp. 483–494, 2013.

[16] C. Jasson Casey, A. Sutton, G. Dos Reis, and A. Sprintson, "Eliminating network protocol vulnerabilities through abstraction and systems language design," in *Network Protocols*

(ICNP), 2013 21st IEEE International Conference on, pp. 1–6, Oct 2013.

[17] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[18] A. Voellmy, A. Agarwal, and P. Hudak, "Nettle: Functional reactive programming for openflow networks," tech. rep., DTIC Document, 2010.

[19] M. Lopez, C. Jasson Casey, G. Dos Reis, and C. Chojnacki, "Safer sdn programming through arbiter," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pp. 65–74, ACM, 2015.

[20] V. Jeyakumar, M. Alizadeh, C. Kim, and D. Mazières, "Tiny packet programs for low-latency network control and monitoring," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, (New York, NY, USA), Association for Computing Machinery, 2013.

[21] K. Li, F. Chang, D. Berger, and W. chang Feng, "Architectures for packet classification caching," in *The 11th IEEE International Conference on Networks, 2003. ICON2003.*, pp. 111–117, Sept 2003.

[22] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, pp. 34–41, Jan 2000.

[23] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, pp. 24–32, Mar 2001.

[24] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," *SIGCOMM Comput. Commun. Rev.*, vol. 29, pp. 135–146, Aug. 1999.

[25] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "Efficuts: Optimizing packet classification for memory and throughput," in *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, (New York, NY, USA), p. 207–218, Association for Computing Machinery, 2010.

[26] T. Inoue, T. Mano, K. Mizutani, S.-I. Minato, and O. Akashi, "Rethinking packet classification for global network view of software-defined networking," in *2014 IEEE 22nd International Conference on Network Protocols*, pp. 296–307, 2014.

[27] F. Pacheco, E. Exposito, M. Gineste, C. Baudoin, and J. Aguilar, "Towards the deployment of machine learning solutions in network traffic classification: A systematic survey," *IEEE Communications Surveys and Tutorials*, vol. 21, no. 2, pp. 1988–2014, 2019.

[28] "Biggest ddos attack' did not hurt the global internet – this time," 2014.

[29] A. Ruia, C. J. Casey, S. Saha, and A. Sprintson, "Flowcache: A cache-based approach for improving sdn scalability," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 610–615, 2016.

[30] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," in *Passive and Active Measurement* (J. Mirkovic and Y. Liu, eds.), (Cham), pp. 347–359, Springer International Publishing, 2015.

[31] L. McHale, J. Casey, P. V. Gratz, and A. Sprintson, "Stochastic pre-classification for sdn data plane matching," in *2014 IEEE 22nd International Conference on Network Protocols*, pp. 596–602, 2014.

[32] "The CAIDA UCSD Anonymized Internet Traces - 2012, kc claffy, Dan Andersen, Paul Hick."

[33] I. L. Chvets and M. H. MacGregor, "Multi-zone caches for accelerating ip routing table lookups," in *Workshop on High Performance Switching and Routing, Merging Optical and IP Technologie*, pp. 121–126, 2002.

[34] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[35] P. Ghoshal, C. J. Casey, P. V. Gratz, and A. Sprintson, "Stochastic pre-classification for software defined firewalls," in *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*, pp. 1–8, IEEE, 2013.

[36] G. Varghese, *Network Algorithmics,: An Interdisciplinary Approach to Designing Fast Networked Devices (The Morgan Kaufmann Series in Networking)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[37] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, pp. 22–31, 2009.

[38] L. A. Belady and F. P. Palermo, "On-line Measurement of Paging Behavior by the Multivalued MIN Algorithm," *IBM Journal of Research and Development*, vol. 18, no. 1, pp. 2–19, 1974.

[39] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[40] R. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[41] P. J. Denning, "The Working Set Model for Program Behavior," *Commun. ACM*, vol. 11, p. 323–333, May 1968.

[42] A. Dainotti, A. Pescapé, P. S. Rossi, F. Palmieri, and G. Ventre, "Internet traffic modeling by means of hidden markov models," *Comput. Netw.*, vol. 52, p. 2645–2662, oct 2008.

[43] D. X. Song, D. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on ssh," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, (USA), USENIX Association, 2001.

[44] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive Insertion Policies for High Performance Caching," *SIGARCH Comput. Archit. News*, vol. 35, p. 381–391, June 2007.

[45] D. A. Jiménez, "Insertion and Promotion for Tree-Based PseudoLRU Last-Level Caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), p. 284–296, Association for Computing Machinery, 2013.

[46] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache Bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pp. 222–233, 2008.

[47] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," *SIGARCH Comput. Archit. News*, vol. 38, p. 60–71, June 2010.

[48] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer, "SHiP: Signature-based Hit Predictor for high performance caching," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 430–441, 2011.

[49] A. Seznec, "A new case for the TAGE branch predictor," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 117–127, 2011.

[50] D. A. Jiménez and C. Lin, "Dynamic Branch Prediction with Perceptrons," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, (USA), p. 197, IEEE Computer Society, 2001.

[51] D. Tarjan and K. Skadron, "Merging Path and Gshare Indexing in Perceptron Branch Prediction," *ACM Trans. Archit. Code Optim.*, vol. 2, p. 280–300, Sept. 2005.

[52] S. M. Khan, Y. Tian, and D. A. Jiménez, "Sampling Dead Block Prediction for Last-Level Caches," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 175–186, 2010.

[53] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[54] S. Mirbagher-Ajorpaz, E. Garza, G. Pokam, and D. A. Jiménez, "CHiRP: Control-Flow History Reuse Prediction," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 131–145, 2020.

[55] D. A. Jiménez and E. Teran, "Multiperspective Reuse Prediction," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), p. 436–448, Association for Computing Machinery, 2017.

[56] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-Based Prefetch Filtering," in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), p. 1–13, Association for Computing Machinery, 2019.

[57] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 146–159, 2020.

[58] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying Deep Learning to the Cache Replacement Problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), p. 413–425, Association for Computing Machinery, 2019.

[59] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, p. 78–89, IEEE Press, 2016.

[60] L. McHale, J. Casey, P. V. Gratz, and A. Sprintson, "Stochastic pre-classification for sdn data plane matching," in *2014 IEEE 22nd International Conference on Network Protocols*, pp. 596–602, 2014.

[61] B. W. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA)-Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.

[62] D. Chicco and G. Jurman, "The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation," *BMC genomics*, vol. 21, no. 1, pp. 1–13, 2020.

[63] Q. Zhu, "On the performance of matthews correlation coefficient (mcc) for imbalanced dataset," *Pattern Recognition Letters*, vol. 136, pp. 71–80, 2020.

[64] A. Seznec, "Analysis of the O-GEometric history length branch predictor," in *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 394–405, 2005.

[65] S. Nowozin, "Improved information gain estimates for decision tree induction," in *ICML 2012*, June 2012.

[66] A. Botta, A. Dainotti, and A. Pescapé, "A tool for the generation of realistic network workload for emerging networking scenarios," *Comput. Networks*, vol. 56, pp. 3531–3547, 2012.

[67] S. Molnár, P. Megyesi, and G. Szabó, "How to validate traffic generators?," in *2013 IEEE International Conference on Communications Workshops (ICC)*, pp. 1340–1344, 2013.

A. APPENDIX

A.1 Matthew's Correlation Coefficient

This appendix aims simplify the interpretation of MCC by Equation A.5. Pearson's Correlation Coefficient is defined in Equation A.1.

$$PCC = corr(X, Y) = \frac{cov(X, Y)}{\sigma_X \sigma_Y} \tag{A.1}$$

Equation A.2 is rewritten with respect to covariance and variance:

$$corr(X, Y) = \frac{cov(X, Y)}{\sqrt{var(X)}\sqrt{var(Y)}} \tag{A.2}$$

Equation A.2 is further rewritten in terms of expected value:

$$corr(X, Y) = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sqrt{E(X^2) - E(X)^2}\sqrt{E(Y^2) - E(Y)^2}} \tag{A.3}$$

Matthew discovered restricting Pearson's Correlation Coefficient to binary classes allows for simplification. After manipulation, and reduction, MCC is commonly represented as seen in Equation A.4.

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{A.4}$$

By correlating the predictions against actual, MCC balances the predictor accuracy across both classes. Matthew's Correlation Coefficient can be interpreted as bisections of the confusion matrix:

$$MCC = \frac{correct(X, Y) - incorrect(X, Y)}{\sqrt{actual(X, Y) * predicted(X, Y)}} \tag{A.5}$$

Where the numerator contains the *difference of products* between the correct diagonal (Equation A.6) and vs the incorrect diagonal (Equation A.7).

$$correct(X, Y) = TP * TN \tag{A.6}$$

$$incorrect(X, Y) = FP * FN \tag{A.7}$$

Normalized by a denominator consisting of *product of sums* across actual ground truth rows (Equation A.8) vs prediction columns (Equation A.9).

$$actual(X, Y) = (TP + FN)(TN + FP) \tag{A.8}$$

$$predicted(X, Y) = (TP + FP)(TN + FN) \tag{A.9}$$

A.2 Feature Table Weight Distributions

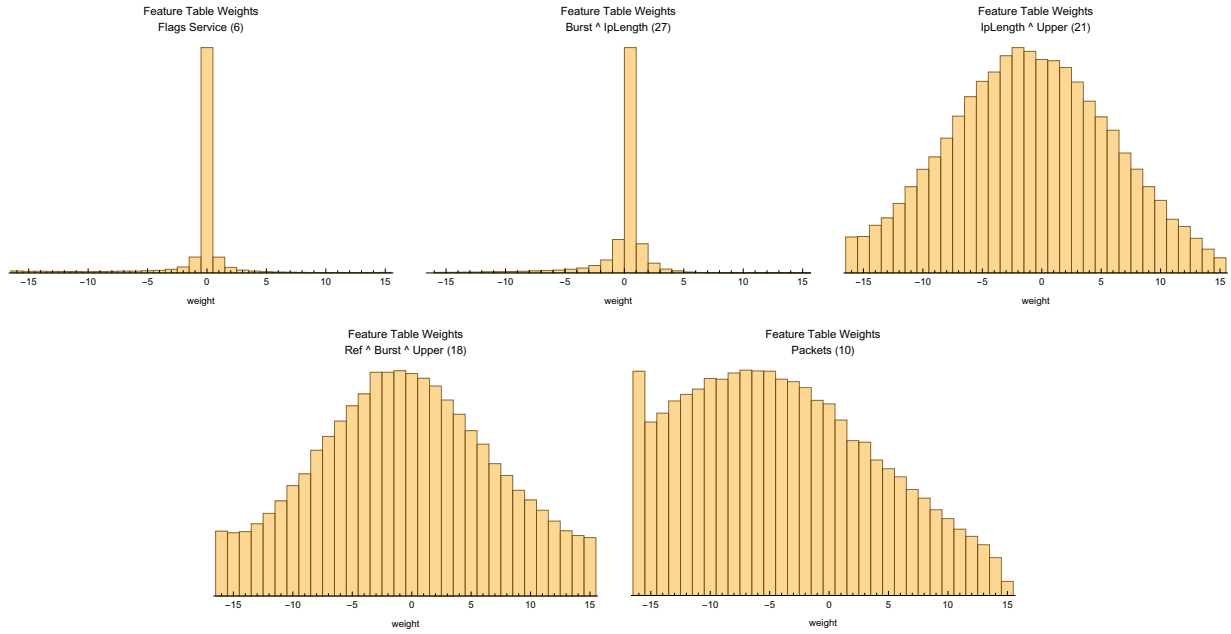


Table A.1: Distributions of Selected Features (in IG_3-*4k* ranked order)

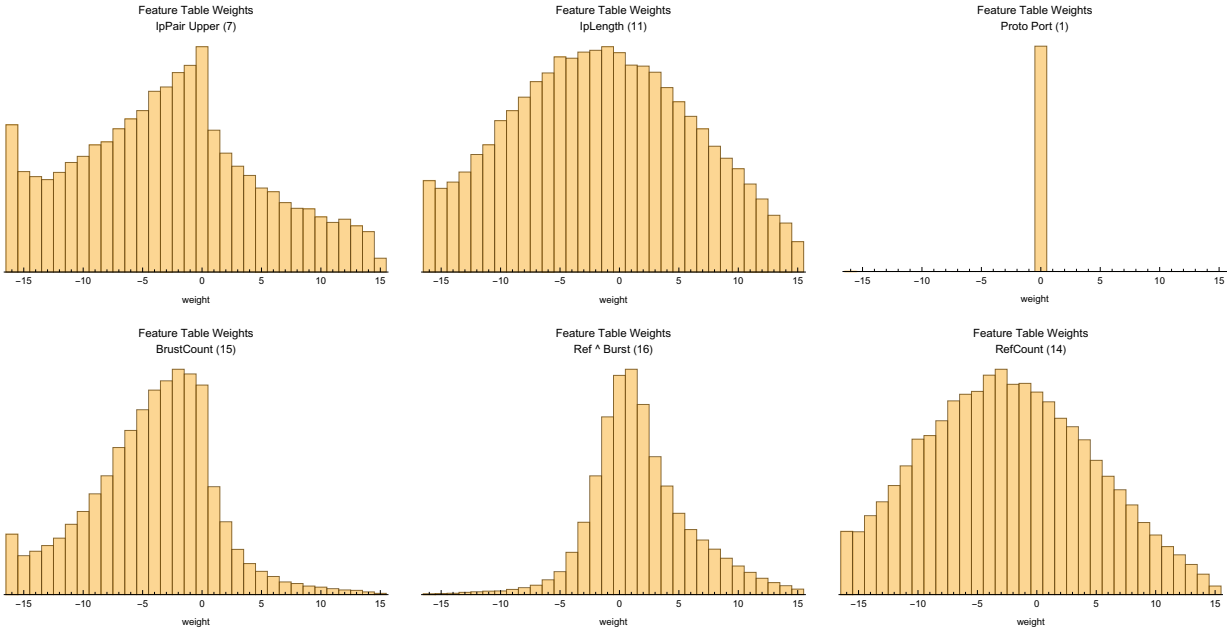


Table A.2: Distributions of Contributing Feature Components (in *MCC-4k* ranked order)

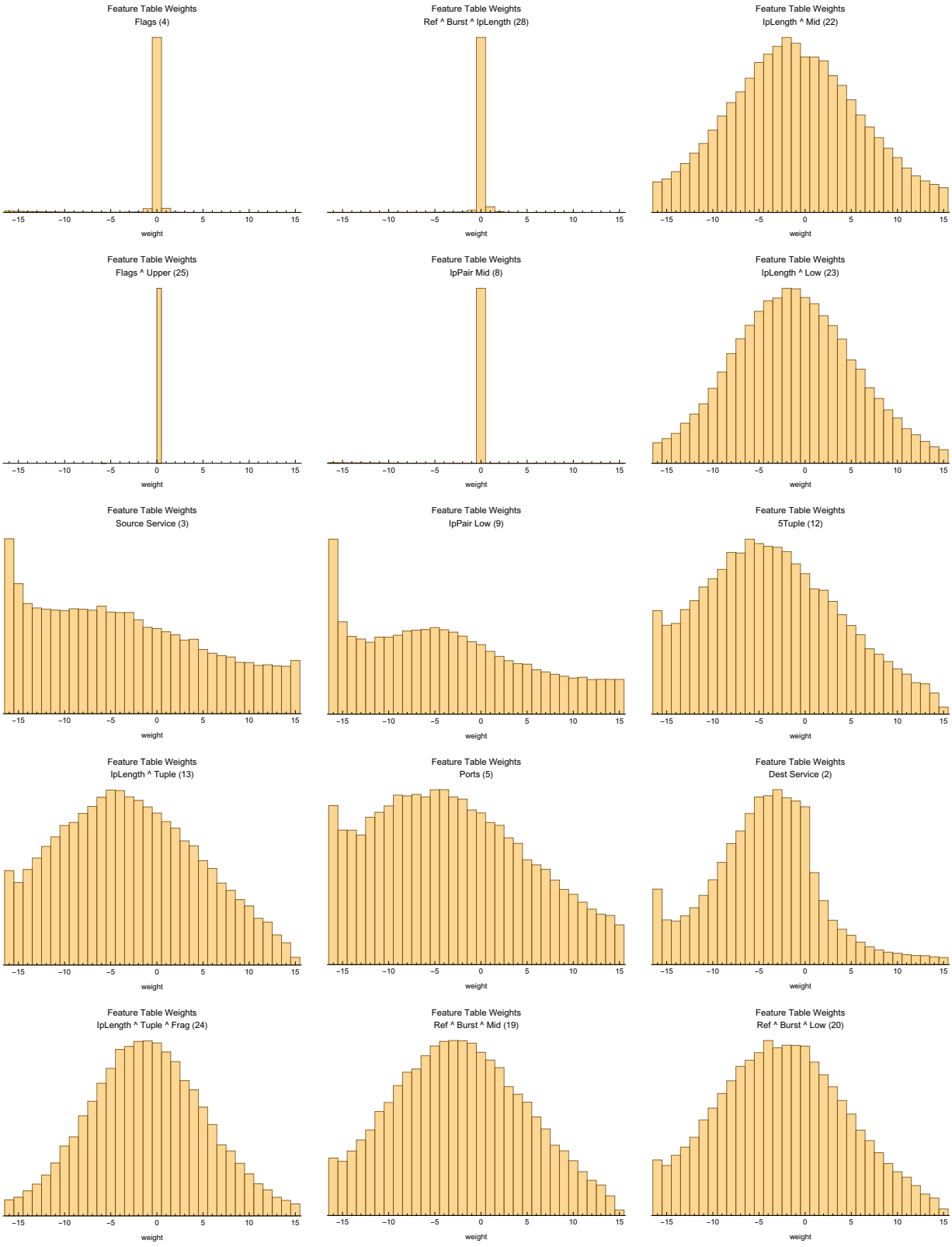


Table A.3: Distributions of Rejected Feature Components (in *MCC-4k* ranked order)

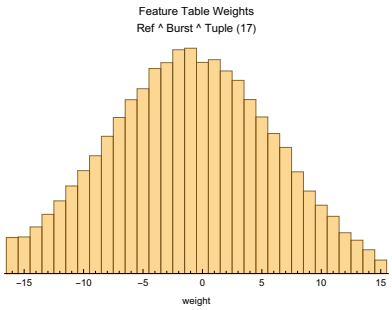


Table A.3 Continued: Distributions of Rejected Feature Components (in *MCC-4k* ranked order)

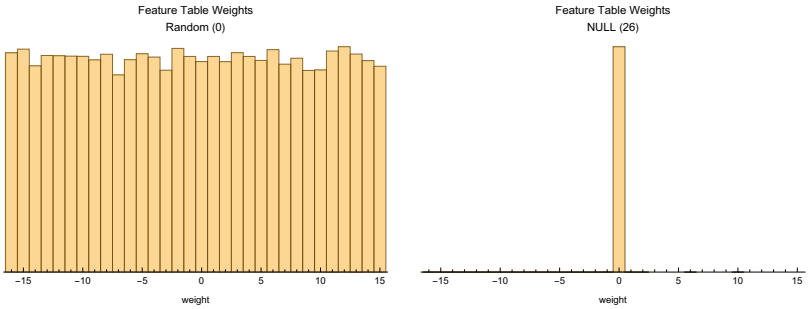


Table A.4: Distributions of Control Features

A.3 Selected Features' Composition

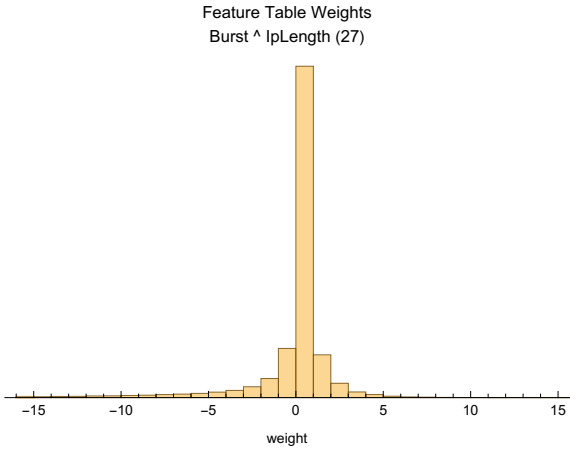


Figure A.1: f_{27} Weight Distribution

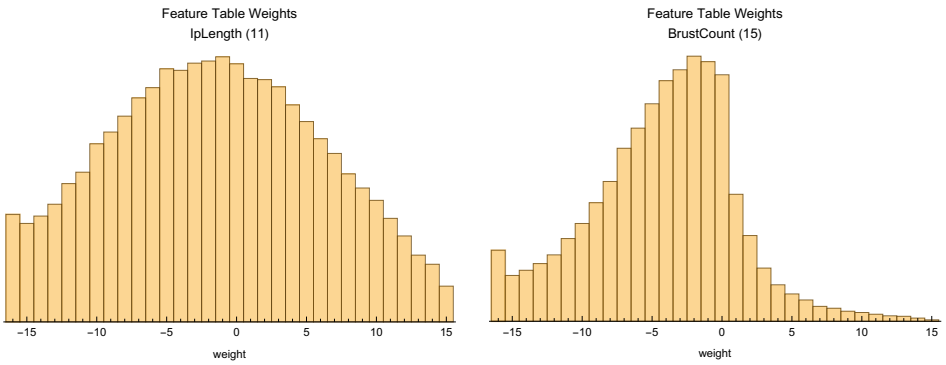


Figure A.2: f_{27} Compositions' Weight Distributions

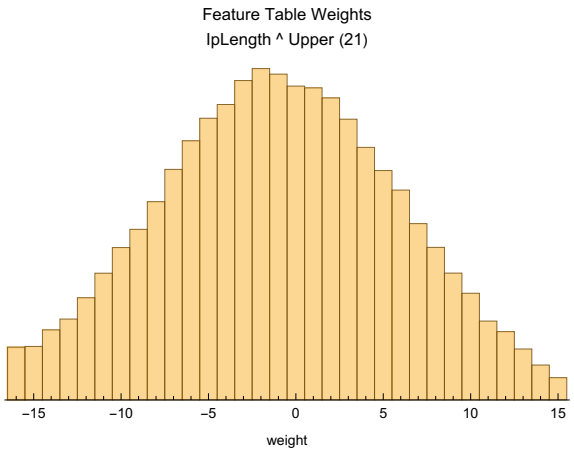


Figure A.3: f_{21} Weight Distribution

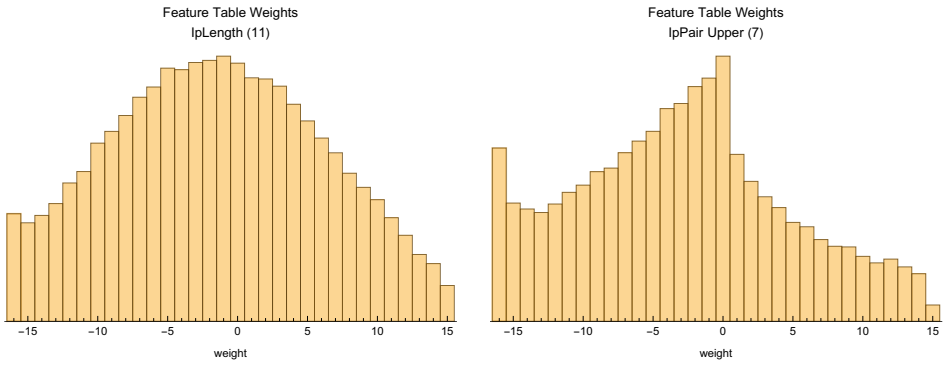


Figure A.4: f_{21} Compositions' Weight Distributions

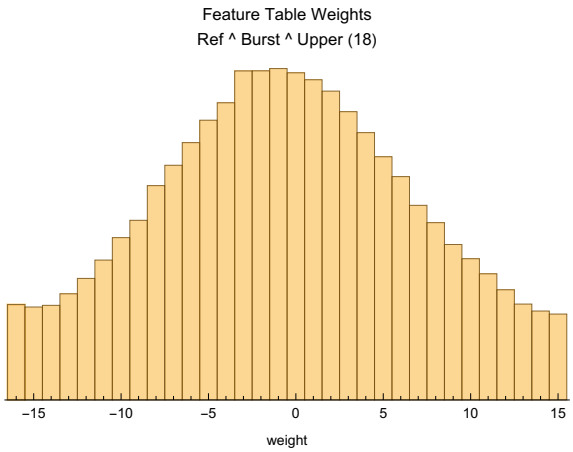


Figure A.5: f_{18} Weight Distribution

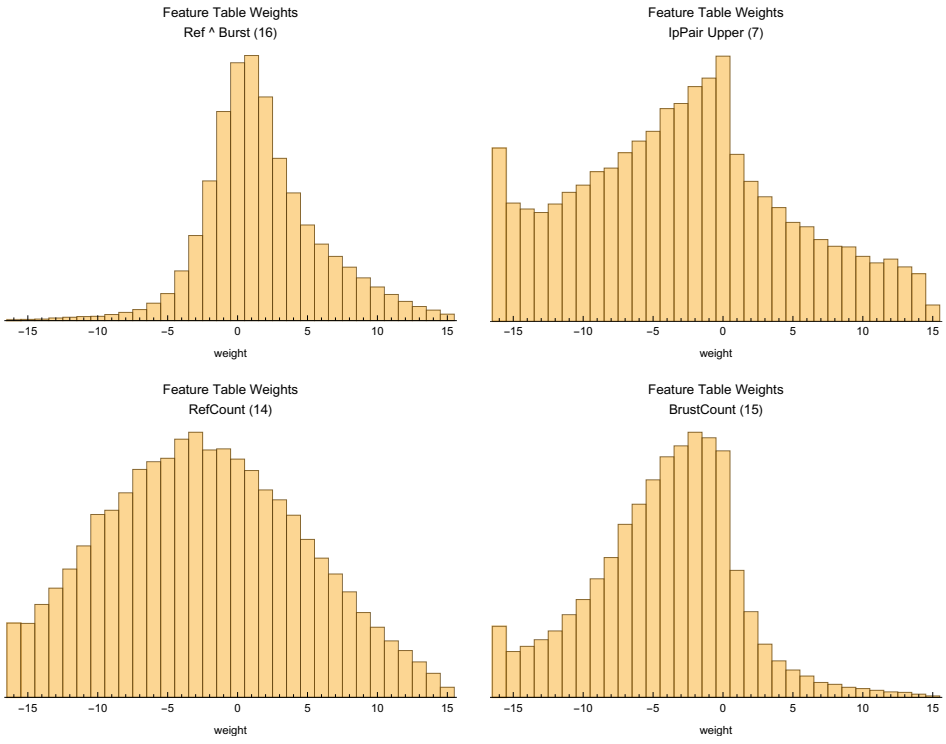
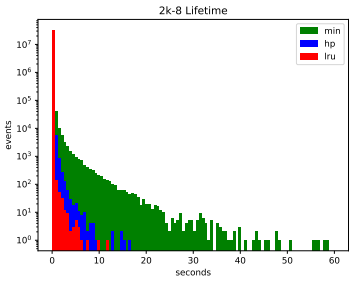
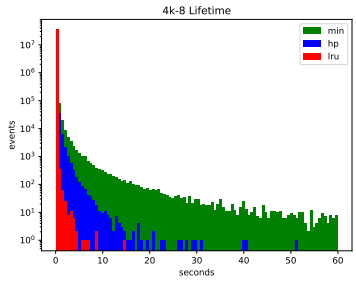


Figure A.6: f_{18} Compositions' Weight Distributions

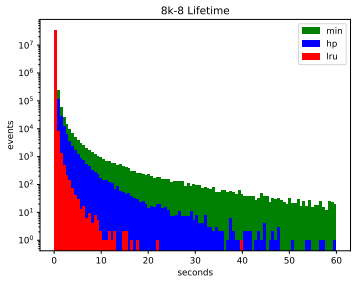
A.4 Cache Pressure Dynamics



2k, 8-way

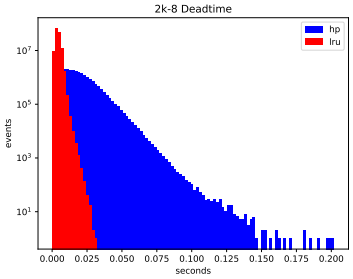


4k, 8-way

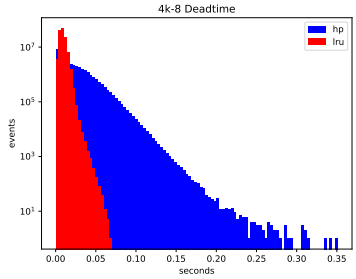


8k, 8-way

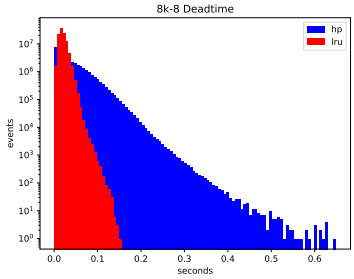
Table A.5: Cache Size vs. Entry Lifetime



2k, 8-way

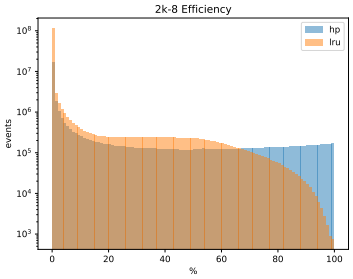


4k, 8-way

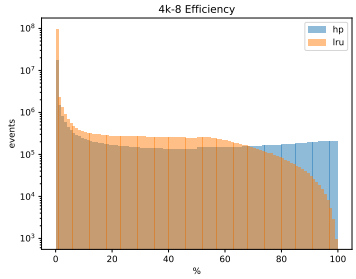


8k, 8-way

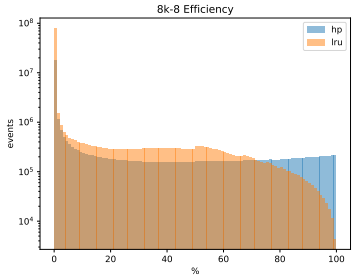
Table A.6: Cache Size vs. Entry Deadtime



2k, 8-way

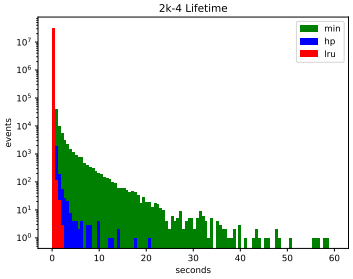


4k, 8-way

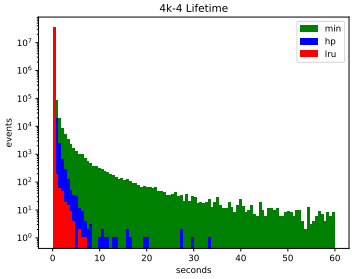


8k, 8-way

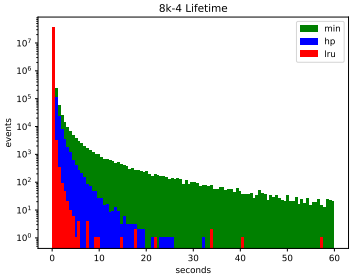
Table A.7: Cache Size vs. Efficiency



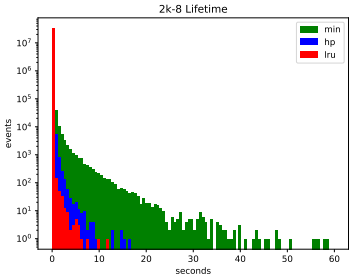
2k, 4-way



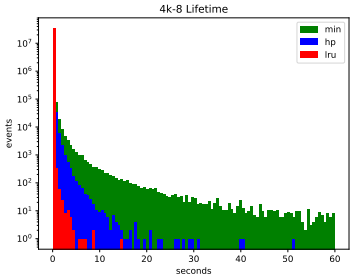
4k, 4-way



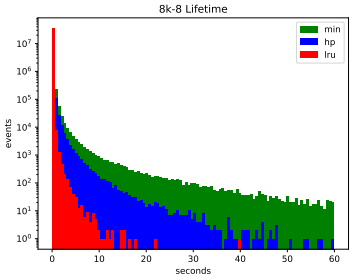
8k, 4-way



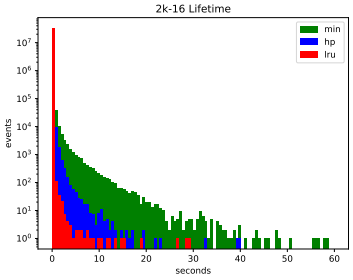
2k, 8-way



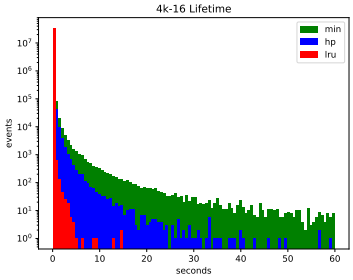
4k, 8-way



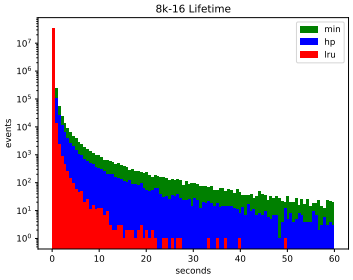
8k, 8-way



2k, 16-way



4k, 16-way



8k, 16-way

Table A.8: Cache Associativity vs. Entry Lifetime