Middleboxes as a Cloud Service

Justine Sherry

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-165
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-165.html

November 19, 2016

Middleboxes as a Cloud Service

By

Justine Marie Sherry

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sylvia Ratnasamy, Chair
Professor Scott Shenker
Professor John Chuang
Professor Arvind Krishnamurthy

Fall 2016

Middleboxes as a Cloud Service

Abstract

Middleboxes as a Cloud Service

by

Justine Marie Sherry

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Sylvia Ratnasamy, Chair

Today's networks do much more than merely deliver packets. Through the deployment of middleboxes, enterprise networks today provide improved security – e.g., filtering malicious content – and performance capabilities – e.g., caching frequently accessed content. Although middleboxes are deployed widely in enterprises, they bring with them many challenges: they are complicated to manage, expensive, prone to failures, and challenge privacy expectations.

In this thesis, we aim to bring the benefits of cloud computing to networking. We argue that middlebox services can be outsourced to cloud providers in a similar fashion to how mail, compute, and storage are today outsourced. We begin by presenting APLOMB, a system that allows enterprises to outsource middlebox processing to a third party cloud or ISP. For enterprise networks, APLOMB can reduce costs, ease management, and provide resources for scalability and failover. For service providers, APLOMB offers new customers and business opportunities, but also presents new challenges. Middleboxes have tighter performance demands than existing cloud services, and hence supporting APLOMB requires redesigning software at the cloud. We re-consider classical cloud challenges including fault-tolerance and privacy, showing how to implement middlebox software solutions with throughput and latency 2-4 orders of magnitude more efficient than general-purpose cloud approaches.

To my parents.

Contents

List of Figures v

List of Tables vii

Acknowledgments viii

1 Introduction 1
- 1.1 Traditional Middlebox Deployments . . . 2
- 1.2 The Cloud Computing Blueprint . . . 3
- 1.3 Obstacles to Moving Middleboxes the Cloud . . . 4
- 1.4 Summary of Results . . . 4
- 1.5 Dissertation Plan . . . 6

2 Traditional Enterprise Middlebox Deployments 7
- 2.1 Middlebox Deployments . . . 7
- 2.2 Complexity in Management . . . 8
- 2.3 Overload and Failures . . . 10
- 2.4 Discussion . . . 11

3 Middleboxes as Cloud Services 13
- 3.1 Design Space . . . 14
  - 3.1.1 Redirection . . . 16
  - 3.1.2 Low Latency Operation . . . 18
  - 3.1.3 APLOMB+ Gateways . . . 20
  - 3.1.4 Summary . . . 21
- 3.2 APLOMB: Detailed Design . . . 23
  - 3.2.1 Enterprise Configuration . . . 23
  - 3.2.2 Cloud Functionality . . . 24
  - 3.2.3 Control Plane . . . 25
  - 3.2.4 Implementation . . . 26
- 3.3 Evaluation . . . 28
  - 3.3.1 Application Performance . . . 28
  - 3.3.2 Scaling and Failover . . . 29

3.3.3 Case Study 30
3.4 Discussion 32
3.5 Related Work 34
3.6 Conclusion 35

4 Fault-Tolerance For Middleboxes **36**
4.1 Problem Space 37
4.1.1 System Model 37
4.1.2 Failure Model 38
4.1.3 Challenges 38
4.2 Goals and Design Rationale 39
4.2.1 Goals 39
4.2.2 Existing Middleboxes 40
4.2.3 Design Options 41
4.2.4 No-Replay Designs 42
4.2.5 Replay-Based Designs 42
4.3 Design 43
4.3.1 Defining Determinants 45
4.3.2 How to Log Determinants 45
4.3.3 Defining a Packet's Dependencies 46
4.3.4 Output Commit 47
4.4 System Implementation 49
4.4.1 Input Logger 49
4.4.2 Master 49
4.4.3 Output Logger 50
4.4.4 Periodic snapshots 51
4.4.5 Replay 51
4.5 Evaluation 52
4.5.1 Overhead on Failure-free Operation 53
4.5.2 Recovery 57
4.6 Related Work 59
4.7 Discussion 60

5 Privacy Preserving Middleboxes **61**
5.1 Overview 63
5.1.1 Usage Scenarios 64
5.1.2 Security and Threat Model 65
5.1.3 System Architecture 66
5.1.4 Protocols 68
5.2 Protocol I: Basic Detection 68
5.2.1 The DPIEnc Encryption Scheme 69
5.2.2 BlindBox Detect Protocol 70

5.2.3 Rule Preparation 72
5.2.4 Validate Tokens 74
5.2.5 Security Guarantees 74
5.3 Protocol II: Limited IDS 74
5.4 Protocol III: Full IDS with Probable Cause Privacy 75
5.5 Discussion 76
5.5.1 Adoption and Deployment 76
5.5.2 Generating Rules 77
5.6 System Implementation 78
5.7 Evaluation 78
5.7.1 Functionality Evaluation 78
5.7.2 Performance Evaluation 80
5.8 Related work 87
5.8.1 Insecure Proposals 87
5.8.2 Computing on Encrypted Data 87
5.9 Conclusion 88

6 Conclusion, Lessons Learned & Thoughts for the Future 89
6.1 The Rise of Network Functions Virtualization 90
6.2 Lessons Learned and Thoughts for the Future 90

Bibliography 93

List of Figures

2.1 8
2.2 8
2.3 9
2.4 11

3.1 Comparing two redirection architectures. 15
3.2 A pure-IP solution cannot ensure that inbound and outbound traffic traverse the same PoP, breaking bidirectional middlebox services. 17
3.3 17
3.4 18
3.5 20
3.6 20
3.7 Average number of middleboxes remaining in enterprise under different outsourcing options. 22
3.8 Architectural components of APLOMB. 23
3.9 27
3.10 28
3.11 30
3.12 Number of middleboxes in the enterprise with and without APLOMB+. The enterprise has an atypical number of 'internal' firewalls and NIDS. 31
3.13 Ratio of peak traffic volume to average traffic volume, divided by protocol. . . . 32
3.14 95^{th} percentile bandwidth without APLOMB, with APLOMB, and with APLOMB+. 32

4.1 Our model of a middlebox application 37
4.2 44
4.3 Four threads (black lines) process packets A, B, C, D. As time goes (left to right), they access (circles) shared variables X, Y, Z, T generating the PALs in parentheses. The red tree indicates the dependencies for packet B. 46
4.4 48
4.5 53
4.6 53
4.7 54

4.8 55
4.9 55
4.10 56
4.11 57
4.12 57
4.13 58

5.1 System architecture. Shaded boxes indicate algorithms added by BlindBox. . . . 63
5.2 Rule preparation. The endpoint has a key k and the middlebox has a keyword r. 73
5.3 80
5.4 82
5.5 82
5.6 84
5.7 85
5.8 85
5.9 86

List of Tables

2.1 10

3.1 Complexity of design and cloud footprint required to outsource different types of middleboxes. 21
3.2 Cost comparison of different cloud bandwidth pricing models given an enterprise with a monthly transfer volume of 500TB (an overestimate as compared to the very large enterprise in our study); assumes conversion rate of 1Mbps of sustained transfer equals 300GB over the course of a month. 33

4.1 52

5.1 80
5.2 81

Acknowledgments

Contributions to this thesis will be addressed chronologically, starting from the beginning. Events are abridged.

1861 Yale University grants the first doctorates of philosophy in US history.

1868 Founding of UC Berkeley.

Some other, trivial events.

1987 The author is born to Stephanie and Patrick Sherry.[1]

1990 Veronica Sherry, sister of the author, is born. Additional sisters follow in 1993 (Moira) and 1996 (Rosalie).[2]

1997 The author meets Neil Taylor, father of her friend Sara.[3]

2005 Laura Finney convinces the author to enroll in Programming I at the University of Washington.[4]

2008 Barack Obama wins presidential election.[5]

2009 Arvind Krishnamurthy and Tom Anderson don't eat undergraduate bones for breakfast.[6]

2009 Colin Scott joins the Reverse Traceroute team.[7]

[1]I could not be luckier to have two such loving and supportive parents.

[2]My sisters are my confidants, my cheerleaders, and my best friends. They inspire me to be my very best.

[3]My first coding project? An Animorphs fan page with starry background and animated gifs under Neil's tutelage. I fell in love with computers because Neil let me play with them.

[4]Laura has been a dear friend and a tremendous influence on my life. I should also mention her parents, Barbara and Bob, who taught me that one can have an academic career and a happy family at the same time.

[5]The connection to this thesis is as follows. Justine attends an election party where Ethan Katz-Bassett is also attending. Barack Obama wins. Justine asks Ethan if she can join his research project, and Ethan says yes. End.

Ethan mentored me not only in college on the way to graduate school, but continues to provide advice about students, work/life balance, and faculty careers to this day. He's great.

[6]They introduced me to the joy of research and planted the idea of an academic career in my head. Indeed, working with Arvind is so much fun that it's only within the last year that I finally managed to publish a paper *without* Arvind as a co-author.

[7]Colin! Always asking why we're in grad school and to what end, the only one of us to actually deserve this doctorate of *philosophy*. I'm so grateful for the countless late-night discussions in the lab with him.

2010 Scott Shenker repeatedly stops the author from being a wallflower.[8]

2010 The author meets her future advisor, Sylvia Ratnasamy.[9]

2010 Invasion of 1044 Keith Avenue.[10]

2011 Founding of NetSys Lab.[11]

2012 APLOMB is published.

2012 The author spends the Summer in Cambridge, UK.[12]

2014 The thesis is proposed.

Much work, until some papers are accepted to SIGCOMM. The author learns she will graduate.

2016 Assorted junior faculty kindly respond to panicked emails from the author. [13]

2016 This thesis is filed.

Future *???*

[8]One of Scott's many strengths is finding a student floundering and identifying exactly who or what they need to move forward. He did this for me the first time at Berkeley visit day, finding me in a corner and introducing me to all the networking faculty; he did it again when he connected me to Sylvia before she'd even joined Berkeley faculty. This skill of his is something I hope to emulate as a professor in the future.

[9]I suspect I've embarassed her enough over the last few weeks with my sappy comments so I'll keep this short: Sylvia has been my most important teacher, role-model, and career guide. She is my hero. I am so looking forward to many more years of knowing her as a mentor, colleague, and friend.

[10]Shoutouts to the whole (extended) 1044 crew: Greg Durrett, Jonathan Long, Jonathan Kohler, Jono Kummerfeld, Edgar Solomonik, Paul and Judy Pearce, Ellen Stuart, Allie and Ryan Janoch, Pat & Caitlin & Molly Virtue, Shaddi Hasan, Elinor Benami, Michael Cole, Greg Affeldt, Mollie and Itamar Kimchi-Schwartz, Emily Cogsdill, (The) Meghan Kelly, and Dave Moore: Don't forget to keep an eye on the lemon tree, may Roger Federer always watch over you, and let your Friday nights be blessed with many wall-snakes.

[11]To Kay Ousterhout, Aurojit Panda, Shivaram Venkataraman (honorary NetSys member), Colin Scott, Radhika Mittal, Colin Scott, Murphy McCauley, Amin Toontoonchian, Peter Gao, Chang Lan, Sangjin Han (my brother), Ethan Jackson, Akshay Narayan and Shoumik Palkar: it's been a joy to have you as collaborators and friends.

[12]Ruben: És o amor da minha vida e estou muito emocionada estar contigo até o fim das nossas dias. O mais dificil nos ultimos anos, mais do que o trabalho ou a preocupação por publicações, tem sido a distancia entre nós. Por telefone e pelo Skype, apoiaste-me muito durante esta tese e agradeço-te por tudo. No futuro espero continuarmos nos apoiando, juntos – nas triunfas e difucultades. Amo-te para sempre.

[13]Thanks to Peter Bailis, Raluca Popa, Ethan K-B, Simon Peter, and Matei Zaharia for all of the 'how to interview' advice and support!

Chapter 1

Introduction

Modern enterprise networks are quite complex. Originally, networks had one very simple goal: forwarding packets. Today, the task of the network has grown to meet new and sophisticated demands. For example, many networks are required to meet security requirements by detecting and blocking malicious behavior [152, 153, 123]. Others perform performance optimizations such as compressing and caching data [136]. Public and carrier networks track bandwidth consumption to bill users for usage [67]. These and many other capabilities – transcoding, address translation, protocol conversion, to name a few more – are widely supported today, and go well beyond the early requirements for networks which merely forward packets.

All of these features are implemented by *middleboxes*: specialized, on-path systems which inspect, transform, and manipulate traffic en route to its destination. Examples of middleboxes [60] include the following.

- *Intrusion Detection/Prevention Systems (IDS/IPS).* These devices inspect both packet headers and contents for known malicious behaviors; upon detection of an attack the device alerts an administrator and may block the connection.
- *Network Address Translators (NATs).* Facing a depleting supply of public IPv4 addresses, NATs allow multiple end hosts to share a single IP address.
- *Transcoders.* These systems convert file formats as data is transmitted, often downgrading size and quality of images so they load faster on resource-constrained mobile devices [162].

While middleboxes are widely deployed to bring well-recognized security and performance benefits, they also introduce new challenges in network administration. As this thesis presents in Chapter 2, middleboxes make network management more complex and more expensive. Around one out of every three devices in enterprise network is a middlebox, each of which cost tens of thousands of dollars. Because each middlebox serves a different purpose (*e.g.* a transcoder is different from an IDS), cognitive overhead for administrators is high as each device requires unique expertise. Furthermore, as this thesis elaborates in Chapters 4 and 5 respectively, middleboxes introduce new and challenging failure modes in networks, and create privacy concerns – both exacerbating challenges for middlebox administration.

Thesis: *By following the blueprint of outsourcing and cloud computing, middleboxes can be made easier to manage, more cost-effective, and more efficient.*

In this thesis, we advocate for a new architecture in how middleboxes are deployed and operated. Instead of requiring middleboxes to be deployed independently by every edge network, enterprise, or university – where administrators must 'reinvent the wheel' over and over – we argue that middlebox deployment should be taken out of the hands of average administrators entirely. Rather, middleboxes should be deployed by clouds and Internet Service Providers as public services, allowing experts to solve common challenges once and for all. Outsourcing middlebox processing in this way mirrors the trend of cloud outsourcing for other systems, *e.g.* for compute and storage. As we will show, the cloud computing blueprint is feasible for networking workloads and brings well-known benefits of cloud deployments to networking: better manageability, cheaper deployments, and more efficient software infrastructure.

1.1 Traditional Middlebox Deployments

Today, middlebox deployments are instantiated in an uncoordinated, device-by-device manner dependent on custom, fixed-function hardware devices. When a network administrator requires new functionality in her network – *e.g.* a new firewall, or a protocol accelerator, or a cache – she purchases a new device which implements the desired features. She then installs the device at a 'choke-point' in her network where traffic is guaranteed to traverse it; many middleboxes may be co-located at the same choke-point to ensure that traffic receives a series of different inspections and modifications. These middleboxes must be deployed in partial topological order: functionality fails if, *e.g.*, data is encrypted before it is passed through a device which inspects traffic for malware. Networks which deploy many middleboxes are hence characterized by the following challenges:

Management Complexity. Management requires knowledge of many heterogenous devices, each middlebox with different goals and configuration requirements. Administrators must cope with these different requirements in purchasing, installation, configuration, error-handling and debugging, *etc.*. In §2.2 we elaborate further on management challenges, all of which lead to a high rate of error: as much as $\frac{2}{3}$ of administrators cite that misconfiguration is their most common cause of failure.

High Capital and Operating Expenses. Every device costs tens of thousands of dollars; administrators must allocate capacity for peak hours of the day when users can consume on average 2-3$\times$ as much bandwidth as a typical hour of the day. More physical devices in a network entails both additional hardware costs and more administrative staff. We discuss these costs in §2.1

Expensive or Nonexistent Failure Recovery. Each middlebox has a custom implementation from a specific vendor; hence any backup infrastructure requires purchasing duplicate hardware for each and every middlebox (often called 1-to-1 backup provisioning). We find in §2.3 that some administrators forgo deploying such backups because of the cost of duplicate infrastructure which usually goes unused.

Custom Solutions for Common Challenges. Failure recovery is illustrative of how common challenges are solved for each and every middlebox, increasing complexity for administrators, wasting resources, and making things more difficult for middlebox developers. We discuss failure recovery further in Chapter 4, and other challenges such as scaling, provisioning, and monitoring that can and should be implemented generally in Chapter 6.

1.2 The Cloud Computing Blueprint

We argue that the challenges discussed in the previous section can be resolved by a new architecture for middlebox deployments, one based on Cloud the Computing Blueprint [46]. We focus on three core concepts in cloud computing and how they can benefit network processing: outsourcing, the illusion of infinite resources, and utility computing.

Outsourcing. In cloud computing, third party providers implement middleboxes rather than end-users. Outsourcing centralizes where advanced expertise is needed: a few experts at service providers handle common tasks like provisioning, physical configuration, upgrades, *etc.* – solving common challenges for all of their clients at once. Client enterprises are freed of these tasks altogether, reducing administrative complexity. Lower complexity leads to fewer human-hours dealing with middleboxes, and hence lower operational expenses.

Illusion of Infinite Resources. The huge scale of a third party provider can be tapped into by clients, but only as needed. Hence, at peak usage hours, a client may purchase more capacity, but scale down to use fewer resources at average or low usage hours. Overall this cuts down on capital costs for clients, who do not need to purchase infrastructure planning for maximum utilization – they simply scale up and down their usage. Similarly, when a system fails, a client may purchase the capacity of a new device; however, the client does not need to pay for that device in the absence of failure.

Software Utility Computing. applications are independent from physical infrastructure and may be migrated from machine to machine, scaled by adding more generic resources, and integrated with other applications via standardized APIs. Utility computing is a prerequisite to benefit from outsourcing and infinite resources, and also brings other benefits such as the ability to design generic solutions to common challenges (such as failover and scaling), the ability to implement continuous upgrades, and cost benefits of amortizing equipment costs not only among clients but different applications as well. Middleboxes are traditionally sold as atomic units with hardware and software entirely coupled and hence not amenable to utility computing. Shifting middleboxes from the monolithic approach to one based on software is the focus of an industry movement known as Network Functions Virtualization;

the goals of NFV dovetail with those of this thesis and hence we discuss NFV in Chapter 6.

1.3 Obstacles to Moving Middleboxes the Cloud

The benefits of moving to the cloud follow a familiar story of the same arguments that motivated a cloud shift for compute and storage as well. Nonetheless, migrating middleboxes to the cloud present several unique, technical challenges that must be solved in order to achieve cloud computing's promised benefits.

Performance Overheads. Migrating middleboxes to the cloud can introduce performance overheads in two ways. First, as we will discuss in Chapter 3, moving middleboxes to a third party provider necessitates redirecting traffic to a cloud datacenter to receive processing – potentially inflating latencies, introducing jitter, and reducing throughput. Second, middleboxes as deployed within the cloud datacenter, if poorly implemented, may fail to meet throughput requirements of tens of gigabits per second or ultra-low latency requirements per device, typically under 100μs.

Functional Equivalence. Middleboxes are typically deployed local to an enterprise, and directly on-path for traffic. Given performance constraints, implementations in software, and locality requirements, it's unclear that moving middleboxes to the cloud will be able to provide the same functionality as if they were deployed locally. Functional equivalence concerns never existed for web services or batch compute tasks in migrating to the cloud, as their as their correct operation is not sensitive to topology.

Privacy. Redirecting traffic through a service provider's infrastructure reveals all traffic content to this third party – revealing potentially confidential information. Middleboxes already introduce privacy tension in between users and administrators who are known to them; typically in office environments a user has no expectation of privacy on a corporate network. However, the shift to the cloud exposes both user and enterprise-internal traffic to a third, external party. Advances in functional cryptography have shown how to ameliorate this challenge for applications such as databases [126] and webservers [127], but their performance overheads run in to the milliseconds – too high for middleboxes and network traffic.

1.4 Summary of Results

This thesis presents three novel systems which demonstrate the feasibility and highlight some of the benefits of outsourcing middleboxes to the cloud.

APLOMB is a system implementing the overall outsourcing architecture, redirecting traffic from a remote enterprise to a cloud provider's infrastructure where it can receive processing before being sent out to the Internet. APLOMB illustrates the following:

- The feasibility of outsourcing given *wide area performance* properties from real universities and one major enterprise using the APLOMB infrastructure. APLOMB on average *improves* round-trip latencies, penalizes download times by only 5%, and has no noticeable impact on jitter.

- The feasibility of outsourcing to provide *functional equivalence* to existing middlebox deployments. APLOMB serves as an existence proof that almost all middleboxes can be outsourced, with only one class of middleboxes (discussed in §3.1.4) remaining behind. A typical large enterprise (10k-100k hosts) would see a 90% reduction in on-premises middleboxes, and a typical very large (>100k hosts) enterprise would see a 98% reduction.
- The benefit of outsourcing in (a) reducing the number of on-premise middleboxes at enterprises hence reduced management overhead; and (b) providing resources for scalability which can fluctuate to as much as 13×peak demand relative to average hours of the day.

We present APLOMB in Chapter 3.

FTMB is a system that performs stateful failure recovery for middleboxes in software. FTMB demonstrates:

- The benefit of utility-computing in allowing multiple, heterogenous middleboxes to share one backup device. Since software and hardware are decoupled, a backup is merely a generic compute server on standby ready to run any middlebox software as needed. This turns the 1:1 backup ratio to a many:1 ratio.
- The benefit of utility-computing in enabling a generic solution to a common problem – fault tolerance. All middleboxes can adopt the same algorithms and use common interfaces to interact with backup components to achieve fault-tolerance in a uniform mechanism. This saves developers from reinventing new solutions for every device, and administrators from having to understand diverse implementations of the same features.
- The feasibility of implementing generic middlebox extensions in software with acceptably low overheads. FTMB imposes only 30μs of latency overhead and 5-30% throughput reductions, making it suitable for practical use within a cloud datacenter.

We present FTMB in Chapter 4.

BlindBox is a system which allows Deep Packet Inspection (DPI) middleboxes to operate directly over encrypted traffic, without learning the contents of that traffic. BlindBox shows:

- The feasibility of implementing outsourced middleboxes without providing the cloud provider complete access to user data, thus relieving challenges to outsourcing due to privacy.
- The benefit of utility-computing in enabling a generic solution to a common problem – again, all DPI middleboxes (including IDS, parental filters, and exfiltration detectors) can implement common algorithms and invoke the same APIs, as the BlindBox approach implements a privacy solution that can be used *in common across all middleboxes.*

We present BlindBox in Chapter 5.

Overall, these three systems demonstrate the overall feasibility and benefits of the cloud computing approach for middleboxes. Nonetheless, the APLOMB architecture overall requires careful attention to system implementation in all of its components, many beyond the scope of this thesis: network virtualization, scaling, scaling and orchestration, software isolation, I/O performance, and so on. We discuss other systems in active development in research and industry which integrate into this vision in Chapter 6. In particular, we discuss Network Functions Virtualization (NFV), which aims to re-architect middleboxes to best take advantage of software utility computing.

1.5 Dissertation Plan

This thesis proceeds as follows. In Chapter 2 we perform a survey of middlebox deployments as of 2011 to understand traditional middlebox deployments and the challenges they present. In Chapter 3 we present APLOMB, which serves as a feasibility study of the overall outsourcing architecture and its benefits for enterprise networks. In Chapter 4, we discuss FTMB, a system for fault-tolerance in software middleboxes. In Chapter 5, we discuss BlindBox, which allows traffic to be processed without revealing traffic contents to the cloud provider. Finally, in Chapter 6 we discuss NFV and present activity in developing new middleboxes, the future of middleboxes as a cloud service, and conclude.

Chapter 2

Traditional Enterprise Middlebox Deployments

In the previous chapter we discussed that middlebox deployments suffer from high capital and operating expenses, management complexity, limited resources for failure recovery, and a lack of general solutions to common problems. In this chapter we present data substantiating these claims. In 2011, we conducted a survey of 57 enterprise network administrators, including the number of middleboxes deployed, personnel dedicated to them, and challenges faced in administering them. To the best of our knowledge, this is the first large-scale survey of middlebox deployments in the research community. Our dataset includes 19 small (fewer than 1k hosts) networks, 18 medium (1k-10k hosts) networks, 11 large (10k-100k hosts) networks, and 7 very large (more than 100k hosts) networks. Our respondents were drawn primarily from the NANOG network operator's group and university networks; 62.9% described their role as an engineers, 27.7% described their role as technical management, and the rest described their role as 'other.' We augment our analysis with network measurements from a single large enterprise with approximately 600 middleboxes and tens of international sites; we elaborate on this dataset in §3.3.3.

2.1 Middlebox Deployments

Our data illustrates that typical enterprise networks are a complex ecosystem of firewalls, IDSes, web proxies, and other devices. Figure 2.1 shows a box plot of the number of middleboxes deployed in networks of all sizes, as well as the number of routers and switches for comparison. Across all network sizes, the number of middleboxes is on par with the number of routers in a network! The average very large network in our data set hosts 2850 L3 routers, and 1946 total middleboxes; the average small network in our data set hosts 7.3 L3 routers and 10.2 total middleboxes.[1]

[1]Even 7.3 routers and 10.2 middleboxes represents a network of a substantial size. Our data was primarily surveyed from the NANOG network operators group, and thus does not include many of the very smallest

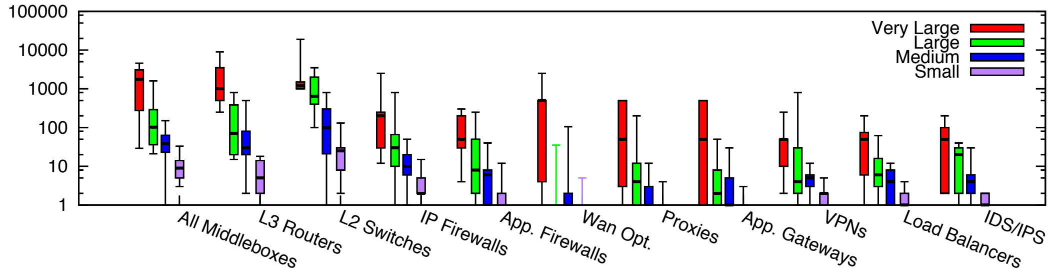


Figure 2.1: Box plot of middlebox deployments for small (fewer than 1k hosts), medium (1k-10k hosts), large (10k-100k hosts), and very large (more than 100k hosts) enterprise networks. Y-axis is in log scale.

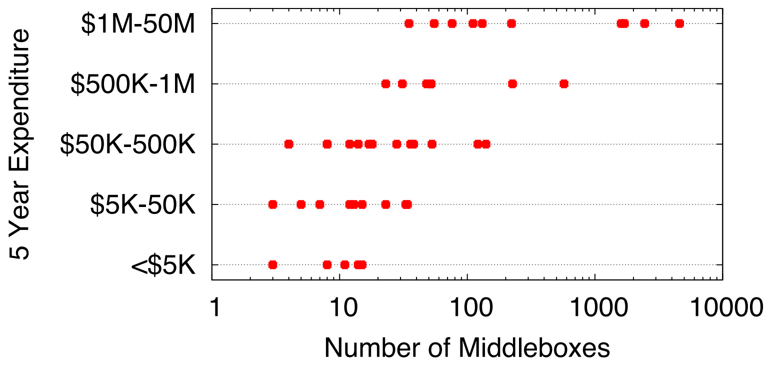


Figure 2.2: Administrator-estimated spending on middlebox hardware per network.

These deployments are not only large, but are also costly, requiring high up-front investment in hardware: thousands to millions of dollars in physical equipment. Figure 2.2 displays five year expenditures on middlebox hardware against the number of actively deployed middleboxes in the network. All of our surveyed very large networks had spent over a million dollars on middlebox hardware in the last five years; the median small network spent between $5,000-50,000 dollars, and the top third of the small networks spent over $50,000.

Paralleling arguments for cloud computing, outsourcing middlebox processing can reduce hardware costs: outsourcing eliminates most of the infrastructure at the enterprise, and a cloud provider can provide the same resources at lower cost due to economies of scale.

2.2 Complexity in Management

Figure 2.1 also shows that middleboxes deployments are diverse. Of the eight middlebox categories we present in Figure 2.1, the median very large network deployed seven categories of middleboxes, and the median small network deployed middleboxes from four. Our cate-

networks (*e.g.* homes and very small businesses with only tens of hosts).

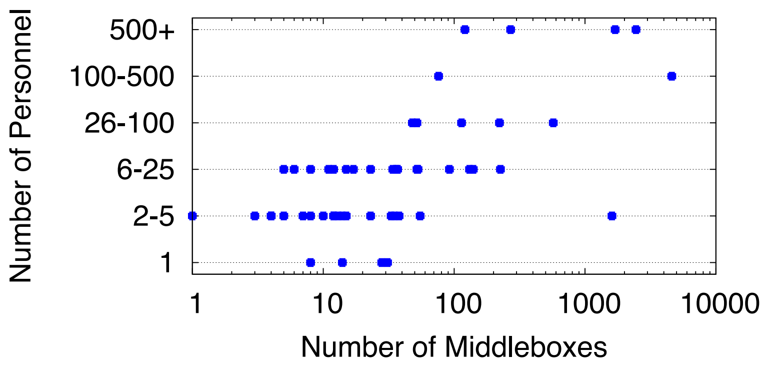


Figure 2.3: Administrator-estimated number of personnel per network.

gories are coarse-grained (*e.g.* Application Gateways include smartphone proxies and VoIP gateways), so these figures represent a *lower bound* on the number of distinct device types in the network.

Managing many heterogeneous devices requires broad expertise and consequently a large management team. Figure 2.3 correlates the number of middleboxes against the number of networking personnel. Even small networks with only tens of middleboxes typically required a management team of 6-25 personnel. Thus, middlebox deployments incur substantial operational expenses in addition to hardware costs.

Understanding the administrative tasks involved further illuminates why large administrative staffs are needed. We break down the management tasks related to middleboxes below.
Upgrades and Vendor Interaction. Deploying new features in the network entails deploying new hardware infrastructure. From our survey, network operators upgrade in the median case every four years. Each time they negotiate a new deployment, they must select between several offerings, weighing the capabilities of devices offered by numerous vendors – an average network in our dataset contracted with 4.9 vendors. This four-year cycle is at the same time both too frequent and too infrequent. Upgrades are too frequent in that every four years, administrators must evaluate, select, purchase, install, and train to maintain new appliances. Upgrades are too infrequent in that administrators are 'locked in' to hardware upgrades to obtain new features. Quoting one administrator:

> Upgradability is very important to me. I do not like it when vendors force me to buy new equipment when a software upgrade could give me additional features.

Cloud computing eliminates the upgrade problem: enterprises sign up for a middlebox *service*; how the cloud provider chooses to upgrade hardware is orthogonal to the service offered.
Monitoring and Diagnostics. To make managing tens or hundreds of devices feasible, enterprises deploy network management tools (e.g., [32, 17]) to aggregate exported monitoring data, *e.g.* SNMP. However, with a cloud solution, the cloud provider monitors utilization and

	Misconfig.	**Overload**	**Physical/Electric**
Firewalls	67.3%	16.3%	16.3%
Proxies	63.2%	15.7%	21.1%
IDS	54.5%	11.4%	34%

Table 2.1: Fraction of network administrators who estimated misconfiguration, overload, or physical/electrical failure as the most common cause of middlebox failure.

failures of specific devices, and only exposes a middlebox *service* to the enterprise administrators, simplifying management at the enterprise.

Configuration. Configuring middleboxes requires two tasks. *Appliance configuration* includes, for example, allocating IP addresses, installing upgrades, and configuring caches. *Policy configuration* is customizing the device to enforce specific enterprise-wide policy goals (*e.g.* a HTTP application filter may block social network sites). Cloud-based deployments obviate the need for enterprise administrators to focus on the low-level mechanisms for appliance configuration and focus only on policy configuration.

Training. New appliances require new training for administrators to manage them. One administrator even stated that existing training and expertise was a key question in purchasing decisions:

> Do we have the expertise necessary to use the product, or would we have to invest significant resources to use it?

Another administrator reports that a lack of training limits the benefits from use of middleboxes:

> They [middleboxes] could provide more benefit if there was better management, and allocation of training and lab resources for network devices.

Training entails not only learning the unique capabilities of each device (*e.g.* setting firewall rules and configuring caching policies) but also learning how to perform the same tasks given different interfaces and implementations. For example, administrators at one very large enterprise shared how devices from different vendors shared data about CPU, memory, and network utilization using multiple different GUIs and data formats. Outsourcing diminishes the training problem by offloading many administrative tasks to the cloud provider, reducing the set of tasks an administrator must be able perform. In summary, for each management task, outsourcing eliminates or greatly simplifies management complexity.

2.3 Overload and Failures

Most administrators who described their role as engineering estimated spending between one and five hours per week dealing with middlebox failures; 9% spent between six and ten

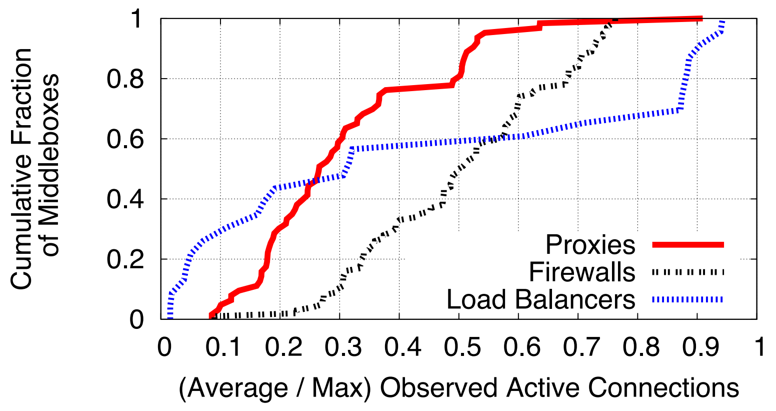


Figure 2.4: Ratio of average to peak active connections for all proxies, firewalls, and load balancers in the very large enterprise dataset.

hours per week. Table 2.1 shows the fraction of network administrators who labeled misconfiguration, overload, and physical/electrical failures as the most common cause of failures in their deployments of three types of middleboxes. Note that this table is *not* the fraction of failures caused by these issues; it is the fraction of administrators who estimate each issue to be the *most common* cause of failure. A majority of administrators stated misconfiguration as the most common cause of failure; in the previous section we highlight management complexity which likely contributes to this figure.

On the other hand, many administrators saw overload and physical/electrical problems as the most common causes of errors. For example, roughly 16% of administrators said that overload was the most common cause of IDS and proxy failure, and 20% said that physical failures were the most common cause for proxies. The cost to recover automatically from such failures is high: recovery often relies on the availability of a standby device. Recovery mechanisms are implemented independently by every vendor, and so for each middlebox that might fail, a 1:1 physical backup purchased from the same vendor is required, effectively doubling capitol costs. The cloud blueprint helps in two ways. First, a generic software utility for middlebox redundancy can standardize fault-tolerance and allow multiple middleboxes to share a single backup. Second, pay-per-use and elastic provisioning enables on-demand scaling and resolves failure with standby devices – without the need for expensive overprovisioning.

2.4 Discussion

To recap, our survey across 57 enterprises illuminates several middlebox-specific challenges that cloud deployments can solve: large deployments with high capital and operating expenses, complex management requirements inflating operation expenses, and failures from physical infrastructure and overload. Cloud outsourcing can cut costs by leveraging economies of scale, simplify management for enterprise administrators. Economies of scale can provide elastic scaling to limit failures. Software utility computing can standardize capabilities like resource monitoring and fault-tolerance, making them easier to reason about and more efficient in resource usage.

Outsourcing to the cloud not only solves challenges in existing deployments, but also presents new opportunities. For example, resource elasticity not only allows usage to scale *up*, but also to scale *down*. Figure 2.4 shows the distribution of average-to-max utilization (in terms of active connections) for three devices across one large enterprise. We see that most devices operate at moderate to low utilization; e.g., 20% of Load Balancers run at $<5\%$ utilization. Today, however, enterprises must invest resources for peak utilization. With a cloud solution, an enterprise can lease a large load balancer only at peak hours and a smaller, cheaper instance otherwise. Furthermore, a pay-per-use model democratizes access to middlebox services and enables even small networks who cannot afford up-front costs to benefit from middlebox processing.

These arguments parallel familiar arguments for the move to cloud computation [47]. This parallel, we believe, only bolsters the case.

Chapter 3

Middleboxes as Cloud Services

We now discuss APLOMB, an architecture that enables outsourcing the processing of their traffic to third-party *middlebox service providers* running in the cloud. In the previous chapter, we discussed shortcomings of the traditional middlebox deployment model. We saw that these challenges mirror the concerns that motivated enterprises to transition their in-house IT infrastructures to managed cloud services. Inspired by this trend, APLOMB illustrates how the promised benefits of cloud computing—reduced expenditure for infrastructure, personnel and management, pay-by-use, the flexibility to try new services without sunk costs, *etc.*—can be brought to middlebox infrastructure. Beyond improving the status quo, cloud-based middlebox services can also make the security and performance benefits of middleboxes available to users such as small businesses and home and mobile users who cannot otherwise afford the associated costs and complexity.

We illustrate that APLOMB is both *feasible* and *beneficial* as a mechanism for enterprise middlebox deployments. To be feasible, APLOMB must meet three challenges:
(1) Functional equivalence. A cloud-based middlebox must offer functionality and semantics equivalent to that of an on-site middlebox – *i.e.*, a firewall must drop packets correctly, an intrusion detection system (IDS) must trigger identical alarms, *etc.* In contrast to traditional endpoint applications, this is challenging because middlebox functionality may be topology dependent. For example, traffic compression must be implemented *before* traffic leaves the enterprise access link, and an IDS that requires stateful processing must see *all* packets in *both* directions of a flow. Today, these requirements are met by deliberately placing middleboxes 'on path' at network choke points within the enterprise – options that are not readily available in a cloud-based architecture. As we shall see, these topological constraints complicate our ability to outsource middlebox processing.
(2) Low complexity at the enterprise. As we shall see, an outsourced middlebox architecture still requires some supporting functionality at the enterprise. We aim for a cloud-based middlebox architecture that minimizes the complexity of this enterprise-side functionality: failing to do so would detract from our motivation for outsourcing in the first place.
(3) Low performance overhead. Middleboxes today are located on the direct path between two communicating endpoints. Under our proposed architecture, traffic is instead sent on a

detour through the cloud leading to a potential increase in packet latency and bandwidth consumption. We aim for system designs that minimize this performance penalty.

We explore points in a design space defined by three dimensions: the redirection options available to enterprises, the footprint of the cloud provider, and the complexity of the outsourcing mechanism. We find that all options have natural tradeoffs across the above requirements and settle on a design that we argue is the sweet spot in this design space, which we term APLOMB, the Appliance for Outsourcing Middleboxes. We implement APLOMB and evaluate our system on EC2 using real end-user traffic and an analysis of traffic traces from a large enterprise network. In our enterprise evaluation, APLOMB imposes an average latency increase of only 1 ms and a median bandwidth inflation of 3.8%. We also show benefits of middlebox outsourcing through a case study of a large enterprise deployment; e.g. showing that enterprises can dynamically invoke additional scaling or new middlebox services in response to new workload requirements with minimal configuration changes or downtime.

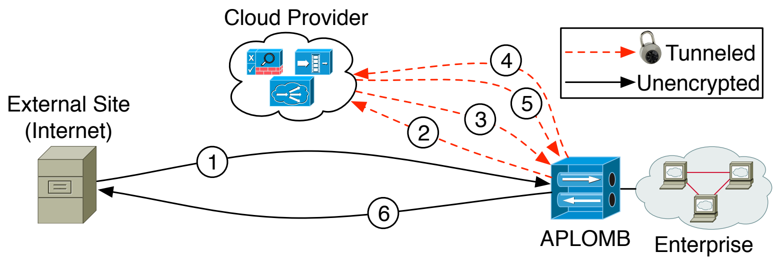
3.1 Design Space

Having established the potential benefits of outsourcing middleboxes to the cloud, we now consider how such outsourcing might be achieved. To start, any solution will require some supporting functionality deployed at the enterprise: at a minimum, we will require some device to *redirect* the enterprise's traffic to the cloud. Hence, we assume that each enterprise deploys a generic appliance which we call an *Appliance for Outsourcing Middleboxes* or *APLOMB*. However, depending on the complexity of the design, the functionality might be integrated with the egress router. We assume that the APLOMB redirects traffic to a Point of Presence (PoP), a datacenter hosting middleboxes which process the enterprise's traffic.

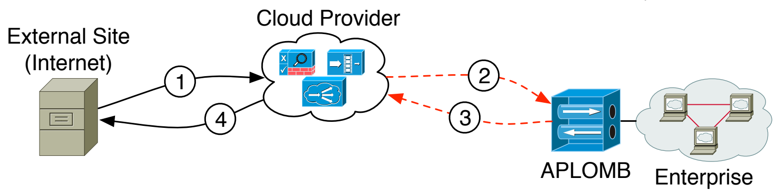
As a baseline, we reflect on the properties of middleboxes as deployed today within the enterprise. Consider a middlebox m that serves traffic between endpoints a and b. Our proposal is to change the ***placement*** of m – moving m from the enterprise to the cloud. Moving m to the cloud eliminates three key properties of its current placement:
(1) on-path: m lies on the direct IP path between a and b
(2) choke point: all paths between a and b traverse m
(3) local: m is located inside the enterprise.

The challenges we face in outsourcing middleboxes all derive from losing the above properties, and our design focuses on compensating for this loss. More specifically, in attempting to regain the benefits of the above properties, we arrive at three design components, as described below.

Redirection: Being on-path makes it trivially easy for a middlebox to obtain the traffic it must process; being at a choke point ensures the middlebox sees both directions of traffic flow between two endpoints (bidirectional visibility is critical since most middleboxes operate at the session level). A middlebox in the cloud loses this natural ability; hence we need a redirection architecture that routes traffic between a and b via the cloud, with both directions



(a) "Bounce" redirection inflates latency.



(b) A direct architecture reduces latency.

Figure 3.1: Comparing two redirection architectures.

of traffic consistently traversing the same cloud PoP.

Latency Strategy: A second consequence of being on-path is that the middlebox introduces no additional latency into the path. In contrast, sending traffic on a detour through the cloud could increase path latency, necessitating a practical strategy for low latency operation.

Further, certain 'extremely local' middleboxes such as proxies and WAN optimizers rely on being local to obtain significant reductions in latency and bandwidth costs. Caching proxies effectively terminate communication from an enterprise host a to an external host b thus reducing communication latency from that of path a-m-b to that of a-m. Likewise, WAN optimizers include a protocol acceleration component that achieves significant latency savings (although using very different mechanisms from a proxy).Thus, the latency optimizations we develop also must serve to minimize the latency increase due to taking extremely local middleboxes out of the enterprise.

APLOMB +: 'Extremely local' middleboxes not only reduce latency, but also reduce bandwidth consumption. Caching proxies, by serving content from a local store, avoid fetching data from the wide area; WAN Optimizers include a redundancy elimination component. To retain the savings in bandwidth consumption, we propose what we term APLOMB + appliances that extend APLOMB to provide comparable bandwidth reduction to extremely local appliances.

We explore solutions for the above design components in §3.1.1 (redirection), §3.1.2 (low latency) and §3.1.3 (APLOMB+). Recall that our design goals are to ensure: (i) functional equivalence, (ii) low performance overhead, and (iii) low enterprise-side complexity. We analyze our design options through the lens of these goals and recap the solution we arrive at in §3.1.4.

3.1.1 Redirection

We consider three natural approaches to redirection and discuss their latency vs. complexity tradeoffs.

Bounce Redirection

In the simplest case, the APLOMB gateway at the enterprise tunnels both ingress and egress traffic to the cloud, as shown in Figure 3.1(a). Incoming traffic is bounced to the cloud PoP (1), processed by middleboxes, then sent back to the enterprise (2,3) and delivered to the appropriate hosts. Outgoing traffic is similarly redirected (4-6).

This scheme has two advantages. First, the APLOMB gateway is the only device that needs to be cloud-aware; no modification is required to existing enterprise network or application infrastructure. Second, the design requires minimal gateway functionality and configuration—a few static rules to redirect traffic to the PoP. The obvious drawback of this architecture is the increase in end-to-end latency due to an extra round trip to the cloud PoP for each packet.[1]

IP-based Redirection

To avoid the extra round-trips in bounce redirection, we might instead route traffic directly to/from the cloud as in Figure 3.1(b). One approach is to redirect traffic at the IP level: for example, the cloud provider could announce IP prefix P on the enterprise's behalf. Hosts communicating with the enterprise direct their traffic to P and thus their enterprise-bound traffic is received by the provider. The cloud provider, after processing the traffic, then tunnels the traffic to the enterprise gateways, who announce an additional prefix P'. [2]

In practice, enterprises would like to leverage the multi-PoP footprint of a provider for improved latency, load distribution and fault tolerance. For this, the cloud provider might advertise P from multiple PoPs so that client traffic is effectively 'anycasted' to the closest PoP. Unfortunately, IP-based redirection breaks down in a multi-PoP scenario since we cannot ensure that traffic from a client a to enterprise b will be routed to the same cloud PoP as that from b to a, thus breaking stateful middleboxes. This is shown in Figure 3.2 where the Cloud-West PoP is closest (in terms of BGP hops) to the enterprise while Cloud-East is closest to the external site. Likewise, if the underlying BGP paths change during a session then different PoPs might be traversed, once again disrupting stateful processing. Finally, because traffic is redirected at the network layer based on BGP path selection criteria (*e.g.*, AS hops), the enterprise or the cloud provider has little control over which PoP is selected and cannot (for

[1] We could eliminate a hop for outgoing traffic by routing return traffic directly from the cloud to the external target. However, this would require the cloud provider to spoof the enterprise's IP addresses, and such messages may be filtered by intermediate ISPs.

[2] The prefix P would in fact have to be owned by the cloud provider. If the cloud provider simply advertises a prefix assigned to the enterprise, then ISPs might filter the BGP announcements as they would fail the origin authorization checks.

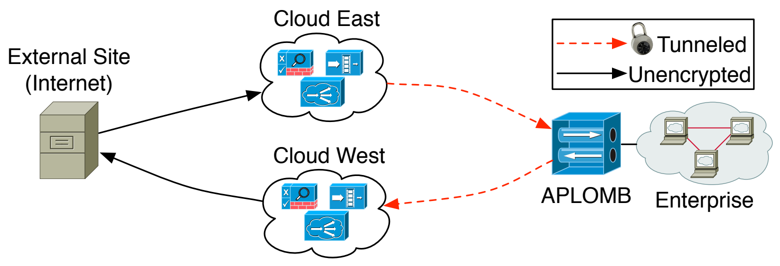


Figure 3.2: A pure-IP solution cannot ensure that inbound and outbound traffic traverse the same PoP, breaking bidirectional middlebox services.

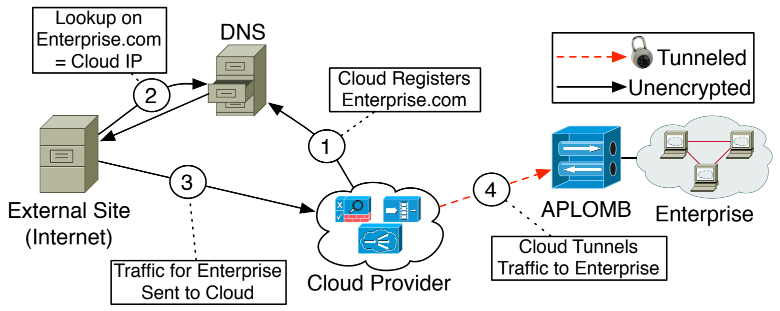

Lookup on Enterprise.com = Cloud IP
DNS
Cloud Registers Enterprise.com
Tunneled
Unencrypted
2
1
3
4
External Site (Internet)
APLOMB
Enterprise
Cloud Tunnels Traffic to Enterprise
Traffic for Enterprise Sent to Cloud
Cloud Provider

Figure 3.3: DNS redirection step by step.

example) pick PoPs to optimize end-to-end latency. Because of these limitations, we reject IP-based redirection as an option.

DNS-based Redirection

DNS-based redirection avoids the problems of IP-based redirection. Here the cloud provider runs the DNS resolution on the enterprise's behalf [4]. We explain this using the example in Figure 3.3. After an enterprise client provides its cloud provider with a manifest of their externally accessible services, the provider registers DNS names on behalf of the client's external services (step 1); *e.g.*, the provider registers '`MyEnterprise.com`'. When a user performs a DNS lookup on `MyEnterprise.com` (step 2), the DNS record directs it to the cloud PoP. The user then directs his traffic to the cloud PoP (step 3), where the traffic undergoes NAT to translate from the public IP address mapped to the cloud PoP to a private IP address internal to the enterprise client's network. The traffic is then processed by any relevant middleboxes and tunneled (step 4) to the enterprise.

This scheme addresses the bidirectionality concerns even in a multi-PoP setting as the intermediate PoP remains the same even if the network-level routing changes. Outbound traffic from the enterprise is relatively easy to control; the gateway device looks up a redirection map to find the PoP to which it must send return traffic. This ensures the symmetric traversal of middleboxes. Finally, Internet traffic initiated by enterprise hosts undergo NAT at the

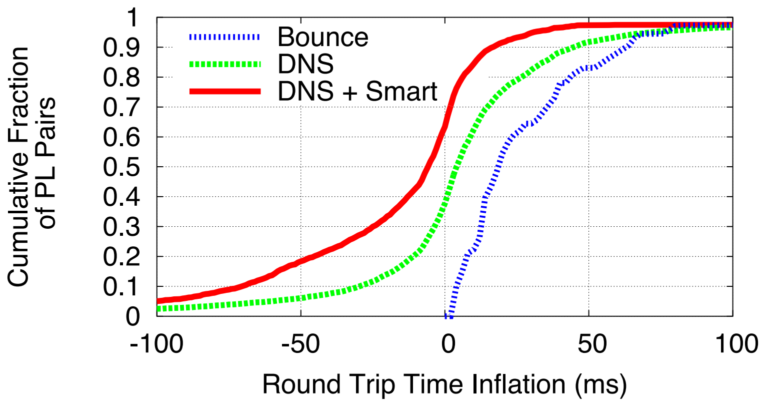


Figure 3.4: Round Trip Time (RTT) inflation when redirecting traffic between US PlanetLab nodes through Amazon PoPs.

cloud provider. Thus, return traffic is forced to traverse the same PoP based on the public IP the provider assigned this connection.[3]

Redirection Tradeoffs

To compare the latency inflation from bounce redirection *vs.* DNS-based redirection, we use measurements from over 300 PlanetLab nodes and twenty Amazon CloudFront locations. We consider an enterprise "site" located at one of fifty US-based PlanetLab sites while the other PlanetLab nodes emulate "clients". For each site e, we pick the closest Amazon CloudFront PoP $P_e^* = \arg\min_P Latency(P, e)$ and measure the impact of tunneling traffic to/from this PoP.

Figure 3.4 shows that the simplest bounce redirection can increase the end-to-end RTT by more than 50ms for 20% of inter-PlanetLab paths. The basic DNS-based redirection reduces the 80^{th} percentile of latency inflation $2\times$ compared to bounce redirection. In fact, for more than 30% of the pairwise measurements, the latency is actually lower than the direct IP path. This is because of well-known triangle inequality violations in inter-domain routing and the fact that cloud providers are very well connected to tier-1/2 ISPs [94]. Hence because the additional enterprise-side complexity required for DNS-based redirection is minimal and yet it achieves significantly lower latencies than Bounce redirection, we choose the DNS-based design.

3.1.2 Low Latency Operation

We now consider additional latency-sensitive PoP selection algorithms and analyze the scale of deployment a cloud provider requires to achieve low latency operation.

[3]Many enterprises already use NATs to external services for other reasons (*e.g.*, flexibility and security); we introduce no new constraints.

Smarter Redirection

So far, we considered a simple PoP selection algorithm where an enterprise site e picks its closest PoP. Figure 3.4 shows that with this simple redirection, 10% of end-to-end scenarios still suffer more than 50ms inflation. To reduce this latency further, we will try to utilize multiple PoPs from the cloud provider's footprint to optimize the *end-to-end* latency as opposed to just the enterprise-to-cloud latency. That is, instead of using a single fixed PoP P_e^* for each enterprise site e, we choose the optimal PoP for each c, e combination. Formally, for each client c and enterprise site e, we identify:

$$P_{c,e}^* : arg \min_P Latency(P, c) + Latency(P, e)$$

We quantify the inflation using smart redirection and the same experimental setup as before, with Amazon CloudFront sites as potential PoPs and PlanetLab nodes as enterprise sites. Figure 3.4 shows that with this "Smart Redirection", more than 70% of the cases have zero or negative inflation and 90% of all traffic has less than 10ms inflation.

Smart redirection requires that the APLOMB appliance direct traffic to different PoPs based on the client's IP and maintain persistent tunnels to multiple PoPs instead of just one tunnel to its closest PoP. This requirement is modest: mappings for PoP selection can be computed at the cloud provider and pushed to APLOMB appliances, and today's commodity gateways can already support hundreds of persistent tunneled connections.

Finally, we note that if communication includes extremely local appliances such as proxies and WAN optimizers, then the bulk of communication is between the enterprise and the middlebox and hence the optimal strategy (which we follow) for such cases is still to simply pick the closest PoP.

Provider Footprint

We now analyze how the middlebox provider's choice of geographic footprint may impact latency. Today's clouds have a few tens of global PoPs and expand as new demand arises [5]. For greater coverage, we could envision an extreme point with a middlebox provider with a footprint comparable to CDNs such as Akamai with thousands of vantage points [154]. While it is clear that a larger footprint provides lower latency, what is not obvious is how large a footprint is required in the context of outsourcing middleboxes.

To understand the implications of the provider's footprint, we extend our measurements to consider a cloud provider with an Akamai-like footprint using IP addresses of over 20,000 Akamai hosts [62]. First, we repeat the the end-to-end latency analysis for paths between US PlanetLab nodes and see that a larger, edge-concentrated Akamai footprint reduces tail latency, but the overall changes are marginal compared to a smaller but well connected Amazon-like footprint. End-to-end latency is the metric of interest when outsourcing most middleboxes – all except for 'extremely local' appliances. Because roughly 70% of inter-PlanetLab node paths actually experience *improved* latency, these results suggest that a middlebox provider can service most customers with most types of middleboxes (*e.g.*, NIDS, firewalls) with an Amazon-like footprint of a few tens of PoPs.

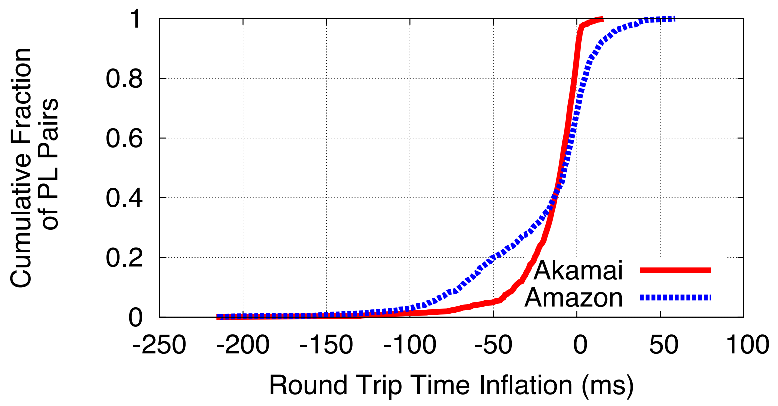


Figure 3.5: PlanetLab-to-PlanetLab RTTs with APLOMB redirection through Amazon and Akamai.

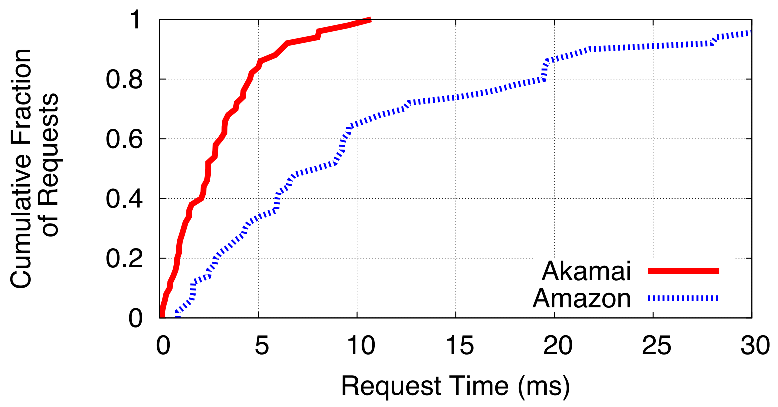


Figure 3.6: Direct RTTs from PlanetLab to nearest Akamai or Amazon redirection node.

To evaluate whether we can outsource even extremely local middleboxes without a high latency penalty (we discuss bandwidth penalties in §3.1.3), we look at the RTT between each Planetlab node and its closest Akamai node in Figure 3.6. In this case, we see a more dramatic impact of Akamai's footprint as it provides sub-millisecond latencies to 20% of sites, and less than 5 ms latencies to almost 90% of sites. An Amazon-like footprint provides only 30% of sites with an RTT <5 ms. Hence our results suggest that an Amazon-like footprint can serve latency acceleration benefits in only a limited portion of the US; to serve a nation-wide set of sites, an Akamai-like footprint is necessary.

3.1.3 APLOMB+ Gateways

As mentioned earlier, extremely local appliances optimize both latency and bandwidth consumption. Our results above suggest that, with an appropriate provider footprint, these appliances can be outsourced and still offer significant latency savings. We now consider the question of the bandwidth savings they enable. Unfortunately, this is a harder problem since bandwidth optimizations must fundamentally be implemented before the enterprise access link in order to be useful. We thus see three options, described below.

The first is to simply not outsource these appliances. From the enterprises we surveyed and Figure 2.1, we see that WAN optimizers and proxies are currently only deployed in large enterprises and that APLOMB is of significant value even if it doesn't cover proxies and WAN optimizers. Nevertheless, we'd like to do better and hence ask whether a full-fledged middlebox is really needed or whether we could achieve much of their benefit with a more minimal design.

Thus the second option we consider is to embed some *general-purpose* traffic compression capabilities into the APLOMB appliance—we term such an augmented appliance an APLOMB+. In §3.3.3, we evaluate APLOMB+ against traditional WAN optimizers using measurements from a large enterprise and show that protocol-agnostic compression [44] can provide similar bandwidth savings (Figure 3.14). While our measurements suggest that in the specific case of WAN optimization a minimalist APLOMB+ suffices, we do not claim that such a minimal capability exists for every conceivable middlebox (*e.g.*, consider an appliance that encodes outgoing traffic for loss protection), nor that APLOMB+ can fully replicate the behavior of dedicated appliances.

Our third option considers more general support for extremely local appliances at the APLOMB gateway. For this, we envision a more "active" appliance architecture that can run specialized software modules (*e.g.*, a FEC encoder). A minimal set of such modules can be dynamically installed either by the cloud provider or the enterprise administrator. Although more general, this option increases both device and configuration complexity for the enterprise. For this reason, and because APLOMB+ suffices to outsource the extremely local appliances we find in today's networks, we choose to implement APLOMB+ in our design.

Type of Middlebox	Enterprise Device	Cloud Footprint
IP Firewalls	Basic APLOMB	Multi-PoP
Application Firewalls	Basic APLOMB	Multi-PoP
VPN Gateways	Basic APLOMB	Multi-PoP
Load Balancers	Basic APLOMB	Multi-PoP
IDS/IPS	Basic APLOMB	Multi-PoP
WAN optimizers	APLOMB+	CDN
Proxies	APLOMB+	CDN

Table 3.1: Complexity of design and cloud footprint required to outsource different types of middleboxes.

3.1.4 Summary

We briefly recap our design and its performance and complexity tradeoffs. At the enterprise end, the functionality we require is embedded in an APLOMB appliance. The basic

APLOMB tunnels traffic to multiple cloud PoPs and stores a redirection map based on which it forwards traffic to the cloud. The cloud provider uses DNS redirection to redirect traffic from the enterprise's external contacts to a cloud PoP before forwarding it to the enterprise. APLOMB+ augments this basic functionality with general compression for bandwidth savings.

In addition to middlebox processing, a cloud-based middlebox provider must support DNS translation for its customers, NAT, and tunneling. The key design choice to a provider is the scale of its deployment footprint. We saw that an Amazon-like footprint often *decreases* latency relative to the direct IP path. However, for performance optimization devices, we saw that a larger Akamai-like footprint is necessary to provide extremely local services with nation-wide availability.

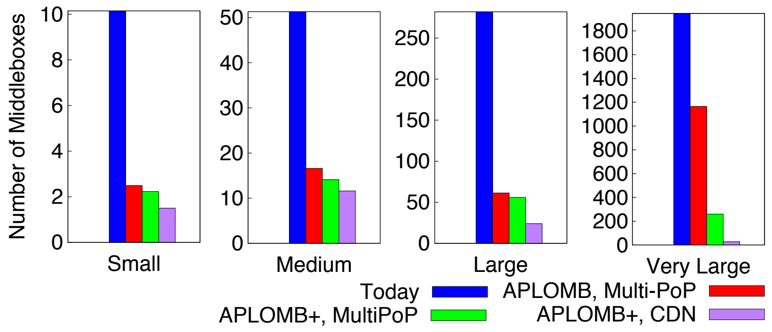


Figure 3.7: Average number of middleboxes remaining in enterprise under different outsourcing options.

Table 3.1 identifies the design option (and hence its associated complexity) that is needed to retain the functional equivalence of the different middleboxes observed in our survey, *e.g.*, outsourcing an IP firewall requires only a basic APLOMB at the enterprise and an Amazon-scale footprint.[4]

Based on this analysis, Figure 3.7 shows the number of middleboxes that remain in an average small, medium, and large enterprise under different outsourcing deployment options. This suggests that small and medium enterprises can achieve almost all outsourcing benefits with a basic APLOMB architecture using today's cloud providers (we discuss the remaining middleboxes, 'internal firewalls', in §3.3.3). The same basic architecture can outsource close to 50% of the appliances in very large enterprise networks; using APLOMB+ increases the percentage of outsourced appliances to close to 90%.

[4]We note that even load balancers can be outsourced since APLOMB retains stateful semantics. One subtle issue is whether load balancers really need to be physically close to backend servers; *e.g.*, for identifying load imbalances at the sub-millisecond granularity. Our conversations with administrators suggest that this is not a typical requirement.

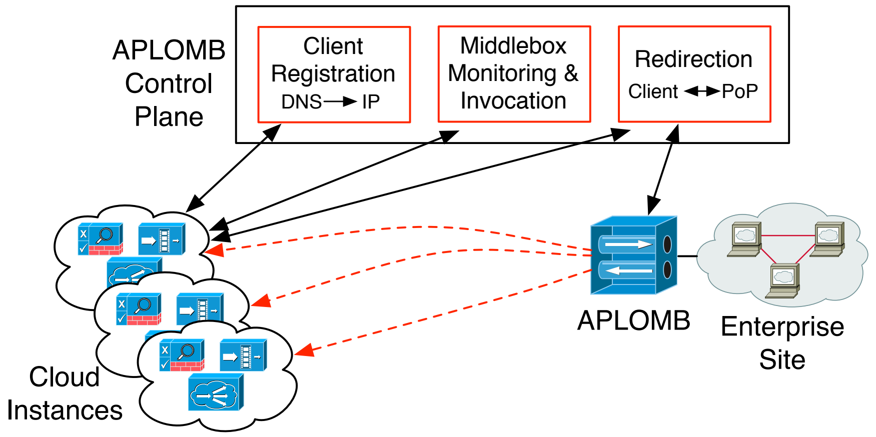


Figure 3.8: Architectural components of APLOMB.

3.2 APLOMB: Detailed Design

In describing the detailed design of the APLOMB architecture, we focus on three key components as shown in Figure 3.8: (1) a APLOMB gateway to redirect enterprise traffic, (2) the corresponding functions and middlebox capabilities at the cloud provider, and (3) a control plane which is responsible for managing and configuring these components.

3.2.1 Enterprise Configuration

Redirecting traffic from the enterprise client to the cloud middlebox provider is simple: an APLOMB gateway is co-located with the enterprise's gateway router, and enterprise administrators supply the cloud provider with a manifest of their address allocations. APLOMB changes neither routing nor switching, and end hosts require no new configuration.

Registration

APLOMB involves an initial registration step in which administrators provide the cloud provider with an *address manifest*. These manifests list the enterprise network's address blocks in its *private* address space and associates each address or prefix with one of three types of address records:

Protected services: Most private IP addresses are registered as protected services. These address records contain an IP address or prefix and the public IP address of the APLOMB device at the gateway to the registered address(es). This registration allows inter-site enterprise traffic to traverse the cloud infrastructure (*e.g.* a host at site A with address 10.2.3.4 can communicate with a host at site B with address 10.4.5.6, and the cloud provider knows that the internal address 10.4.5.6 maps to the APLOMB gateway at site B). The cloud provider allocates no permanent public IP address for hosts with 'protected services' addresses; Internet-destined connections instead undergo traditional NAPT.

DNS services: For hosts which accept incoming traffic, such as web servers, a publicly routeable address must direct incoming traffic to the appropriate cloud PoP. For these IP addresses, the administrator requests DNS service in the address manifest, listing the private IP address of the service, the relevant APLOMB gateway, and a DNS name. The cloud provider then manages the DNS records for this address on the enterprise client's behalf. When a DNS request for this service arrives, the cloud provider (dynamically) assigns a public IP from its own pool of IP addresses and directs this request to the appropriate cloud PoP and subsequent APLOMB gateway.

Legacy IP services: While DNS-based services are the common case, enterprise may require legacy services that require fixed IP addresses. For these services, the enterprise registers the internal IP address and corresponding APLOMB gateway, and the cloud provider allocates a static public IP address at a single PoP for the IP service. For this type of service, we fall back to the single-PoP Cloud-IP solution rather than DNS redirection discussed in §3.1.

APLOMB gateway

The APLOMB gateway is logically co-located with the enterprise's gateway router and has two key functions: (1) maintaining persistent tunnels to multiple cloud PoPs and (2) steering the outgoing traffic to the appropriate cloud PoP. The gateway registers itself with the cloud controller (§3.2.3), which supplies it with a list of cloud tunnel endpoints in each PoP and forwarding rules (5-tuple $\rightarrow$ cloud PoP Identifier) for redirection. (The gateway router blocks all IP traffic into the network that is not tunneled to a APLOMB gateway.) For security reasons, we use encrypted tunnels (e.g., using OpenVPN) and for reducing bandwidth costs, we enable protocol-agnostic redundancy elimination [44]. Note that the functionality required of the APLOMB gateway is simple enough to be bundled with the egress router itself or built using commodity hardware.

For scalability and fault tolerance, we rely on traditional load balancing techniques. For example, to load balance traffic across multiple APLOMB gateways, the enterprise's private address space can be split to direct traffic to, *e.g.* 10.1.0.0/17 to one gateway, and 10.1.128.0/17 to another. To handle gateway failures, we envision APLOMB hardware with fail-open NICs configured to direct the packets to a APLOMB replica under failure. Since each APLOMB box keeps almost no per-flow state, the replica receiving traffic from the failed device can start forwarding the new traffic without interruption to existing flows.

3.2.2 Cloud Functionality

To provide basic outsourcing functionality, the cloud provider has three main tasks: (1) map publicly addressable IP addresses to the appropriate enterprise customer and internal private address, (2) apply middlebox processing services to the customers' traffic according to their *policies* (§3.2.3), and (3) tunnel traffic to and from the appropriate APLOMB gateways at enterprise sites. Thus, the core components – and the enabling technologies to implement them – at the cloud PoP are:

- *Tunnel Endpoints* to encapsulate/decapsulate traffic from the enterprise (and to encrypt/decrypt and compress/decompress if enabled). Tunnel endpoints are implemented using any VPN software [84, 86, 19].
- *Middlebox Instances* to process the customers' traffic. Middleboxes may be implemented in hardware or software.
- *NAT Devices* to translate between publicly visible IP addresses and the clients' internal addresses. NAT devices manage statically configured IP to IP mappings for DNS and Legacy IP services, and generate IP and port mappings for Protected Services (§3.2.1).
- *Policy switching* logic to steer packets between the above components. Policy switching relies on virtual networking to 'steer' traffic between the appropriate middleboxes [102, 129, 125].

Specific outsourcing solutions may differ along two key dimensions depending on whether the middlebox services are: (1) provided by the cloud infrastructure provider (e.g., Amazon) or by third-party cloud service providers running within these infrastructure providers (e.g., [38]), and (2) realized using hardware- (e.g., [30, 21]) or software-based middleboxes (e.g., [136, 40, 27, 142]. Our architecture is agnostic to these choices and accommodates a broad range of deployment scenarios as long as there is some feasible path to implement the four components described above. The specific implementation of APLOMB runs as a third-party service using software-based middleboxes over an existing infrastructure provider.

APLOMB implements basic services relying entirely on existing technologies. Nonetheless, software utility computing – should a cloud provider focus primarily on software middleboxes – opens up opportunities for more robust, efficient, and rich services. These new opportunities often rely on technologies that are new or as of yet unexplored. For example, APLOMB provides the *resources* for automatic fault-tolerance, but implementing correct, generic fault-tolerance will require new algorithms (as we discuss in Chapter 4). For resource efficiency, one might wish to have a scheduler that is aware of network-intensive workloads [119]; for high throughput one might wish for new fast I/O mechanisms [115, 99]; or one might wish to be able to verify the correctness of middlebox software and pipelines of composed middleboxes [75, 120]. We discuss two such opportunities – fault-tolerance (Chapter 4) and privacy (Chapter 5) – and others from related work in Chapter 6.

3.2.3 Control Plane

A driving design principle for APLOMB is to keep the new components introduced by our architecture that are on the critical path – *i.e.*, the APLOMB gateway device and the cloud terminal endpoint – as simple and as stateless as possible. This not only reduces the enterprise's administrative overhead but also enables seamless transition in the presence of hardware and network failures. To this end, the APLOMB Control Plane manages the relevant network state representing APLOMB gateways, cloud PoPs, middlebox instances, and tunnel endpoints. It is responsible for determining optimal redirection strategies between communicating parties, managing and pushing middlebox policy configurations, and dynamically scaling cloud middlebox capacity to meet demands.

In practice, the control plane is realized in a *cloud controller*, which manages every APLOMB gateway, middlebox, tunneling end point, and the internals of the cloud switching policy.[5] Each entity (APLOMB device, middlebox, *etc.*) registers itself with the controller. The controller sends periodic 'heartbeat' health checks to each device to verify its continued activity. In addition, the controller gathers RTTs from each PoP to every prefix on the Internet (for PoP selection) and utilization statistics from each middlebox (for adaptive scaling). Below we discuss the redirection optimization, policy management, and middlebox scaling performed by the cloud controller.

Redirection Optimization. Using measurement data from the cloud PoPs, the cloud controller pushes the current best (as discussed in §3.1.2) tunnel selection strategies to the APLOMB gateways at the enterprise and mappings in the DNS. To deal with transient routing issues or performance instability, the cloud controller periodically updates these tunneling configurations based on the newest measurements from each cloud PoP.

Policy Configuration. The cloud controller is also responsible for implementing enterprise- and middlebox-specific *policies*. Thus, the cloud provider provides a rich policy configuration interface that exports the available types of middlebox processing to enterprise administrators and also implements a programmatic interface to specify the types of middlebox processing required [101]. Enterprise administrators can specify different *policy chains* of middlebox processing for each class of traffic specified using the traditional 5-tuple categorization of flows (i.e., source and destination IPs, port values and the protocol). For example, an enterprise could require all egress traffic to go through a firewall $\rightarrow$ exfiltration engine $\rightarrow$ proxy. and require that all ingress traffic traverse a firewall $\rightarrow$ IDS, and all traffic to internal web services further go through an application-level firewall. If appropriate, the provider may also export certain device-specific configuration parameters that the enterprise administrator can tune.

Middlebox Scaling. APLOMB providers have a great deal of flexibility in how they actually implement the desired middlebox processing. In particular, as utilization increases on a particular middlebox, the APLOMB provider simply increases the number of instances of that middlebox being utilized for a client's traffic. Using data from heartbeat health checks on all middleboxes, the cloud controller detects changes in utilization. When utilization is high, the cloud controller launches new middleboxes and updates the policy switching framework; when utilization is low, the cloud controller deactivates excess instances. Detailed mechanisms for software middlebox scaling are explored in [119, 161, 88, 133].

3.2.4 Implementation

We built a prototype system for cloud middlebox processing using middlebox processing services running on EC2 and APLOMB endpoints in our lab and at the authors' homes. We consciously choose to use off-the-shelf components that run on existing cloud providers and end host systems. This makes our system easy to deploy and use and demonstrates that the barriers to adoption are minimal. Our APLOMB endpoint software can be deployed on a

[5]While the cloud controller may be in reality a replicated or federated set of controllers, for simplicity this discussion refers to a single logically centralized controller.

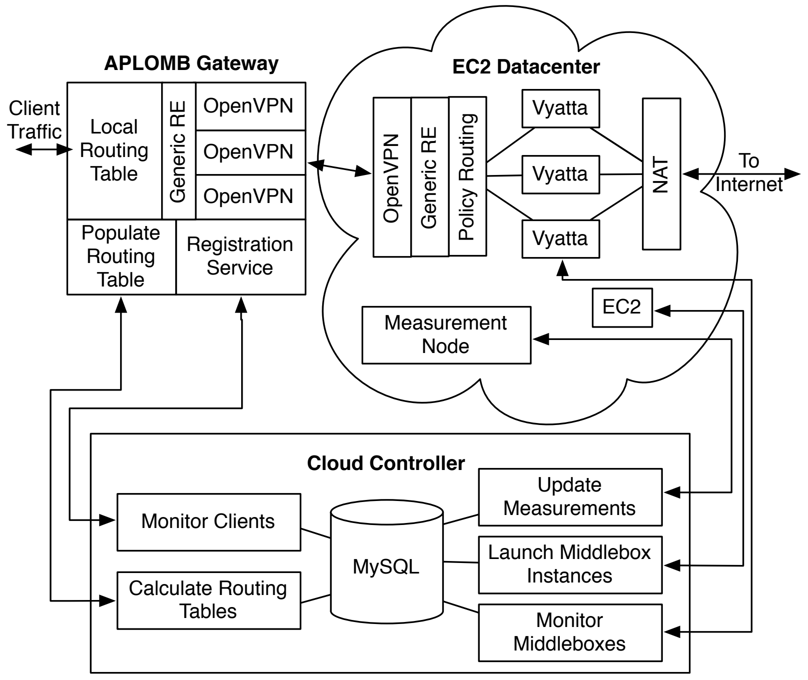


Figure 3.9: Software architecture of APLOMB.

stand-alone software router or as a tunneling layer on an end host; installing and running the end host software is as simple as connecting to a VPN.

Figure 3.9 is a software architecture diagram of our implementation. We implement a cloud controller on a server in our lab and use geographically distributed EC2 datacenters as cloud PoPs. Our cloud controller employs a MySQL database to store data on middlebox nodes, RTTs to and from cloud PoPs, and registered clients. The cloud controller monitors APLOMB devices, calculates and pushes routing tables to the APLOMB devices, requests measurements from the cloud PoPs, monitors middlebox instances, and scales middlebox instances up or down as demand varies.

At the enterprise or the end host, the APLOMB gateway maintains several concurrent VPN tunnels, one to a remote APLOMB at each cloud PoP. On startup, the APLOMB software contacts the cloud controller and registers itself, fetches remote tunnel endpoints for each cloud PoP, and requests a set of initial tunnel redirection mappings. A simple tunnel selection layer, populated by the cloud controller, directs traffic to the appropriate endpoint tunnel, and a redundancy elimination encoding module compresses all outgoing traffic. When run on a software router, ingress traffic comes from an attached hosts for whom the router serves as their default gateway. Running on a laptop or end host, static routes in the kernel direct application traffic to the appropriate egress VPN tunnel.

EC2 datacenters host tunnel endpoints, redundancy elimination decoders, middlebox routers, and NATs, each with an inter-device switching layer and controller registration and monitoring service. For tunneling, we use OpenVPN [19], a widely-deployed VPN solution with packages for all major operating systems. We use a Click [107] implementation of the redundancy elimination technique described by Anand et al [44]. For middlebox process-

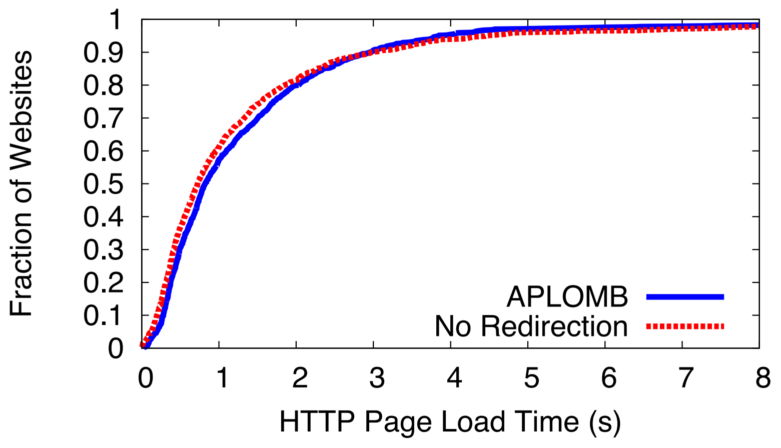


Figure 3.10: CDF of HTTP Page Load times for Alexa top 1,000 sites with and without APLOMB.

ing, we use Vyatta [36], a customizable software middlebox. Our default Vyatta installation performs firewalling, intrusion detection, caching, and application-layer web filtering. Policy configurations (§3.2.3) are translated into Vyatta configurations such that each client can have a unique Vyatta configuration dependent on their needs. Finally, each cloud PoP also hosts one 'measurement node', which periodically issues ping measurements for RTT estimation to assist in PoP selection.

3.3 Evaluation

We now evaluate APLOMB. First, we present performance benchmarks for three common applications running over our implementation (§4.5.1).We then demonstrate APLOMB's dynamic scaling capability and its resilience to failure (§3.3.2). Having shown that APLOMB is practical, we return to our goal of outsourcing all middlebox functionality in an enterprise with a trace-driven evaluation of middlebox outsourcing using APLOMB, applied to data from a middlebox deployment in a large enterprise (§3.3.3).

3.3.1 Application Performance

We first demonstrate that APLOMB's architecture is practical for enterprise use with performance benchmarks for common applications using our APLOMB implementation.

HTTP Page Loads: In Figure 3.10, we plot page load times (fetching the front page and all embedded content) from a university network for the Alexa top 1,000 most popular web pages with and without APLOMB processing. We performed this experiment with a vacant cache. For pages at the 50^{th} percentile, page loads without APLOMB took 0.72 seconds, while page loads with took 0.82 seconds. For pages at the 95^{th} percentile, using APLOMB results in shorter page load times: 3.85 seconds versus 4.53 seconds.

BitTorrent: While we don't expect BitTorrent to be a major component of enterprise traffic, we chose to experiment with Bit Torrent because it allowed us to observe a bulk transfer over a long period of time, to observe many connections over our infrastructure simultaneously, and to establish connections to non-commercial endpoints. We downloaded a 698MB public domain film over BitTorrent with and without APLOMB from both a university network and from a residential network, five times repeatedly. The average residential download took 294 seconds without APLOMB, with APLOMB the download speed increased 2.8% to 302 seconds. The average university download took 156 seconds without APLOMB, with APLOMB the average download took 165 seconds, a 5.5% increase.

Voice over IP: Voice over IP (VoIP) is a common enterprise application, but unlike the previously explored applications, VoIP performance depends not only on low latency and high bandwidth, but on low *jitter*, or variance in latency. APLOMB easily accommodates this third demand: we ran VoIP calls over APLOMB and for each call logged the jitter estimator, a running estimate of packet interarrival variance developed for RTP. Industry experts cite 30ms of one-way jitter as a target for maximum acceptable jitter [8]. In the first call, to a residential network, median inbound/outbound jitter with APLOMB was 2.49 ms/2.46 ms and without was 2.3 ms/1.03 ms. In the second, to a public WiFi hotspot, the median inbound/outbound jitter with APLOMB was 13.21 ms/14.49 ms and without was 4.41 ms/4.04 ms.

In summary, these three common applications suffer little or no penalty when their traffic is redirected through APLOMB.

3.3.2 Scaling and Failover

To evaluate APLOMB's dynamic scaling, we measured traffic from a single client to the APLOMB cloud. Figure 3.11 shows capacity adapting to increased network load over a 10-minute period. The client workload involved simultaneously streaming a video, repeatedly requesting large files over HTTP, and downloading several large files via BitTorrent. The resulting network load varied significantly over the course of the experiment, providing an opportunity for capacity scaling. The controller tracks CPU utilization of each middlebox instance and adds additional capacity when existing instances exceed a utilization threshold for one minute.

While middlebox capacity lags changes in demand, this is primarily an artifact of the low sampling resolution of the monitoring infrastructure provided by our cloud provider. Once a new middlebox instance has been allocated and initialized, actual switchover time to begin routing traffic through it is less than 100ms. To handle failed middlebox instances, the cloud controller checks for reachability between itself and individual middlebox instances every second; when an instance becomes unreachable, APLOMB ceases routing traffic through it within 100ms. Using the same mechanism, the enterprise APLOMB can cope with failure of a remote APLOMB, re-routing traffic to another remote APLOMB in the same or even different cloud PoP, providing fault-tolerance against loss of an entire datacenter.

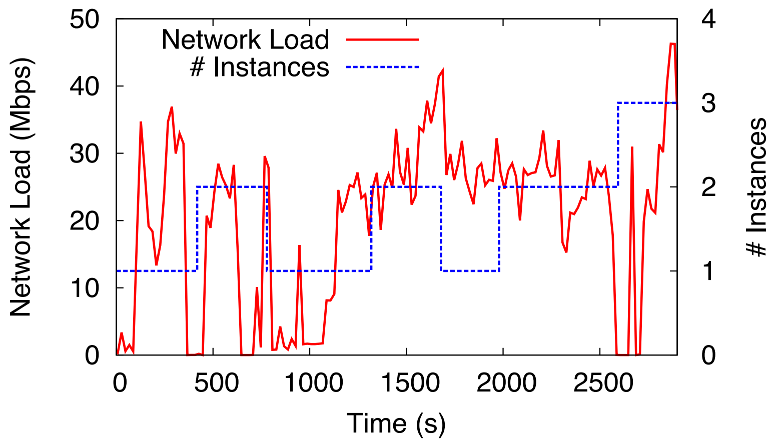


Figure 3.11: Network load (Y_1) and number of software middlebox instances (Y_2) under load. Experiment used low-capacity instances to highlight scaling dynamics.

3.3.3 Case Study

We set out with the goal of outsourcing as many middleboxes as possible and reducing enterprise costs, all the while without increasing bandwidth utilization or latency. We revisit this using the data from the very large enterprise to determine:

- How many middleboxes can the enterprise outsource?
- What are the gains from elastic scaling?
- What latency penalty will inter-site traffic suffer?
- How much does the enterprise's bandwidth costs increase?

Middleboxes Outsourced: Figure 3.12 shows that the large enterprise can outsource close to 60% of the middleboxes under a CDN footprint with APLOMB+.

This high fraction of outsourceability comes despite an atypically high deployment of "internal" firewalls and NIDS at this enterprise. Internal firewalls protect a host or subnetwork not only from Internet-originated traffic, but from traffic originated within the enterprise; the most common reason we found for these deployments was PCI compliance for managing credit card data. While the average enterprise of this size deploys 27.7 unoutsourceable internal firewalls, this enterprise deploys over 100 internal firewalls. From discussions with the network's administrators, we learned these were installed in the past to protect internal servers against worms that preferentially scanned internal prefixes, *e.g.* CodeRed and Nimda. As more IT infrastructure moves to the cloud (see §4.7), many internal firewalls will be able to move to the cloud as well.

Cost Reduction: To evaluate benefits from elastic scaling, in Figure 3.13 we focus on each site of the enterprise and show the ratio of peak-to-average volumes for total inter-site traffic. We use sites across three continents: North America (NA-x), Asia (AS-x), and Europe (EU-x). The peak represents a conservative estimate of the traffic volume the enterprise has provisioned at the site, while the average is the typical utilization; we see that most sites are

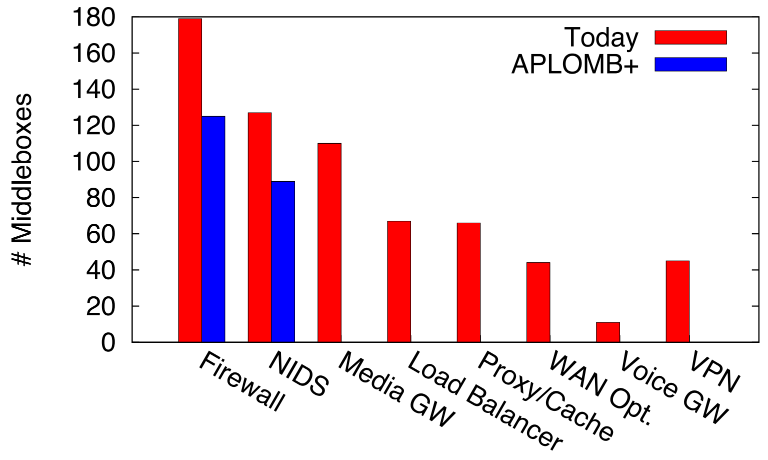


Figure 3.12: Number of middleboxes in the enterprise with and without APLOMB+. The enterprise has an atypical number of 'internal' firewalls and NIDS.

provisioned over 2× their typical load, and some of the smaller sites as much as 12×! In addition, we show peak-to-average values for the top four protocols in use. The per-protocol numbers are indicative of elasticity savings per middlebox, as different protocols are likely to traverse different middleboxes.

Latency: We measured redirection latency for inter-site traffic between the top eleven sites of the enterprise through the APLOMB infrastructure by pinging hosts at each site from within EC2. We found that for more than 60% of inter-site pairs, the latency with redirection is almost identical to the direct RTT. We found that most sites with inflated latency were in Asia, where EC2 does not have a wide footprint.

We also calculated a weighted inflation value, weighted by traffic volume and found that in expectation a typical redirected packet experiences only 1.13 ms of inflation. This results from the fact that the inter-site pairs with high traffic volume actually have negative inflation, by virtue of one or both endpoints being in the US or Europe, where EC2's footprint and connectivity is high.

Bandwidth: Last, we evaluate bandwidth inflation. We ran a traffic trace with full packet payloads collected at a different small enterprise [15] through our APLOMB prototype with and without generic redundancy elimination. Without Generic RE, the bandwidth utilization increased by 6.2% due to encryption and encapsulation overhead. With Generic RE, the bandwidth utilization reduced by 28%, giving APLOMB+ a 32% improvement over basic APLOMB.

Many larger enterprises already compress their inter-site traffic using WAN optimizers. To evaluate the impact of switching compression for inter-site traffic from a traditional WAN optimizer solution to APLOMB+, we compared our observed benefits to those provided by WAN optimizers at eight of the large enterprise sites. In Figure 3.14, we measure the bandwidth cost of a given site in terms of the 95th percentile of the total traffic volume with

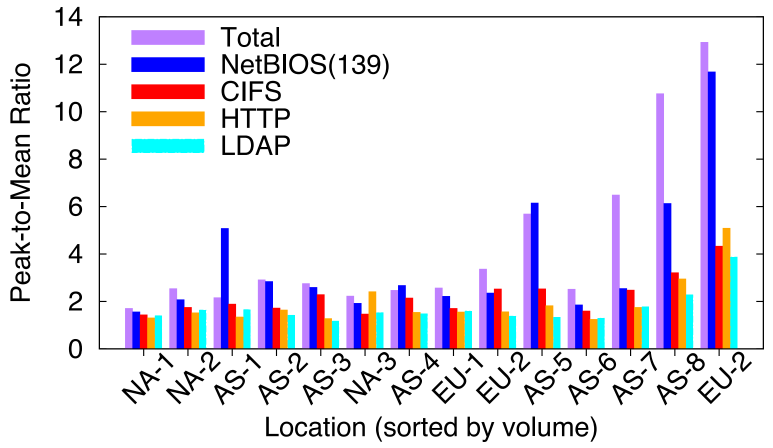


Figure 3.13: Ratio of peak traffic volume to average traffic volume, divided by protocol.

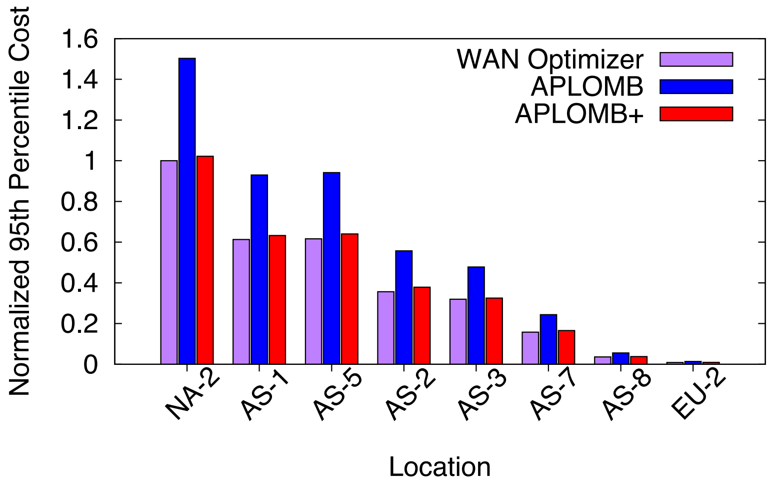


Figure 3.14: 95^{th} percentile bandwidth without APLOMB, with APLOMB, and with APLOMB+.

a WAN Optimizer, with APLOMB, and with APLOMB+. With APLOMB, the worst case inflation is 52% in the median case and at most 58%; APLOMB+ improves this to a median case of 3.8% inflation and a worst case of 8.1%.

3.4 Discussion

Before concluding, we mention some final thoughts on the future of "hybrid" enterprise/cloud architectures, potential cost models for bandwidth, and security challenges that continue to face APLOMB and cloud computing.

IT Outsourcing and Hybrid Clouds: APLOMB complements the ongoing move by enterprises from locally-hosted and managed infrastructure to outsourced cloud infrastructure. A

Pricing Model	Total Cost	$/GB	$/Mbps
Standard EC2	30003.20	0.0586	17.58
Amazon DirectConnect	11882.50	0.0232	6.96
Wholesale Bandwidth	6826.70	0.0133	4.00

Table 3.2: Cost comparison of different cloud bandwidth pricing models given an enterprise with a monthly transfer volume of 500TB (an overestimate as compared to the very large enterprise in our study); assumes conversion rate of 1Mbps of sustained transfer equals 300GB over the course of a month.

network administrator at one large enterprise we surveyed reported their company's management had issued a broad mandate to moving a significant portion of their IT infrastructure to the cloud. Federal government agencies are also rapidly moving their IT infrastructure to the cloud, in compliance with a mandate to adopt a "cloud first" policy for new services and to reduce the number of existing federal datacenters by 800 before 2015 [109]. As these services move to the cloud, the middleboxes protecting them (including internal firewalls, which APLOMB itself cannot outsource) will move to the cloud as well.

Nevertheless, many enterprises plan to keep at least some local infrastructure, citing security and performance concerns for applications currently deployed locally [41]. Further, user-facing devices such as laptops, desktops, smartphones, and printers will always remain within the enterprise – and the majority of middlebox services benefit these devices rather than servers. With some end hosts moving to the cloud, and the majority remaining behind in the enterprise, multiple vendors now offer services for integrating public cloud services with enterprises' existing infrastructure [3, 34], facilitating so-called "hybrid clouds" [96]. APLOMB allows administrators to evade the middlebox-related complexity in this hybrid model by consolidating middleboxes in only one deployment setting.

Bandwidth Costs: APLOMB reduces the cost of middlebox infrastructure, but it may increase bandwidth costs due to current cloud business models. Today, tunneling traffic to a cloud provider necessitates paying for bandwidth twice – once for the enterprise network's access link, and again at the cloud provider. Nevertheless, this does not mean that APLOMB will double bandwidth costs for an enterprise. We observed earlier that redundancy elimination and compression can reduce bandwidth demands at the enterprise access link by roughly 30%. This optimization is not possible without redirection through a cloud PoP, and could allow a lower capacity, less expensive access link to satisfy an enterprise's needs.

The largest factor in the cost of APLOMB for an enterprise is the bandwidth cost model used by a cloud provider. Today, cloud providers price bandwidth purely by volume; for example, Amazon EC2 charges between $0.05-$0.12 per GB of outgoing traffic, decreasing as volume increases (all incoming traffic is free). On the other hand, a dedicated APLOMB service provider would be able to take advantage of wholesale bandwidth, which is priced by transfer rate. We convert between the two pricing strategies (per-GB and per-Mbps) with the rough conversion factor of 1Mbps sustained monthly throughput equaling 300GB per month. This is in comparison with "wholesale" bandwidth prices of $3-$5 per Mbps for high-volume customers. As a result, though current pricing strategies are not well-suited for APLOMB, a dedicated APLOMB provider could offer substantially lower prices. Indeed, Amazon offers a bulk-priced bandwidth service, "DirectConnect", which offers substantially lower per-GB costs for high-volume customers [3]. Table 3.2 provides a comparison of the bandwidth costs for a hypothetical enterprise which transfers 500TB of traffic per month to and from a cloud service provider under each of these models. These charges a minimal compared to expected savings in hardware, personnel, and other management costs.

Security Challenges: Adopting APLOMB brings with it the same security questions as have challenged cloud computing. These challenges have not stopped widespread adoption of cloud computing services, nor the willingness of security certification standards to certify cloud services (for example, services on Amazon EC2 can achieve PCI-1 compliance, the highest level of certification for storing credit card data). However, these challenges remain concerns for APLOMB and cloud computing in general. Just as cloud storage services have raised questions about providing a cloud provider unencrypted access to data, cloud middlebox services give the cloud provider unencrypted access to traffic flows. We believe this is potentially a major obstacle to many enterprises making use of middlebox processing services. While some cloud services such as those used for storage can use end-to-end cryptography to shield data from third party providers, middlebox processing cannot use standard cryptography techniques since the service requires allowing middleboxes access to unencrypted data. Hence, we discuss a functional cryptography-based technique to address privacy concerns in Chapter 5.

3.5 Related Work

Our work contributes to and draws inspiration from a rich corpus of work in cloud computing, redirection services, and network management.

Cloud Computing: The motivation for APLOMB parallels traditional arguments in favor of cloud computing, many of which are discussed by Armbrust et al. [47]. APLOMB also adapts techniques from traditional cloud solutions, *e.g.* utilization monitoring and dynamic scaling [25], and DNS-based redirection to datacenters with optimal performance for the customer [150].

Middlebox Management: Others have tackled middlebox management challenges within the enterprise [101, 102, 51, 73, 142]. Their solutions offer insights we can apply for managing middleboxes within the cloud – *e.g.*, the policy-routing switch of Joseph et al. [102], the management plane of Ballani et al. [51], and the consolidated appliance of Sekar et al. [142]. None of these proposals consider moving middlebox management out of the enterprise entirely, as we do. Like us, ETTM [73] proposes removing middleboxes from the enterprise network but, where we advocate moving them to the cloud, ETTM proposes the opposite: pushing middlebox processing to enterprise end hosts. As such, ETTM still retains the problem of middlebox management in the enterprise. Sekar et al [142] report on the middlebox deployment of a single large enterprise; our survey is broader in scope (covering a range of management and failure concerns) and covers 57 networks of various scales. They also propose a consolidated middlebox architecture that aims to ameliorate some of the administrative burden associated with middlebox management, but they do not go so far as to propose removing middleboxes from the enterprise network entirely.

Redirection Services: Traffic redirection infrastructures have been explored in prior work [45, 148, 159] but in the context of improving Internet or overlay routing architectures as opposed to APLOMB's goal of enabling middlebox processing in the cloud. RON showed how routing via an intermediary might improve latency; we report similar findings using cloud PoPs as intermediaries. Walfish et al. [159] propose a clean-slate architecture, DOA, by which end hosts explicitly address middleboxes. Gibb et al. [92] develop a service model for middleboxes that focuses on service-aware routers that redirect traffic to middleboxes that can be in the local network or Internet.

Cloud Networking: Using virtual middlebox appliances [36] reduces the physical hardware cost of middlebox ownership, but cannot match the performance of hardware solutions and does little to improve configuration complexity. Some startups and security companies have cloud-based offerings for specific middlebox services: Aryaka [6] offers protocol acceleration; ZScalar [38] performs intrusion detection; and Barracuda Flex [7] offers web security. To some extent, our work can be viewed as an extreme extrapolation of their services and we provide a comprehensive exploration and evaluation of such a trend. CloudNaaS [54] and startup Embrane [11] aim at providing complete middlebox solutions for enterprise services that are *already* in the cloud.

3.6 Conclusion

Outsourcing middlebox processing to the cloud relieves enterprises of major problems caused by today's enterprise middlebox infrastructure: cost, management complexity, capacity rigidity, and others. APLOMB succeeds in outsourcing the vast majority of middleboxes from a typical enterprise network without impacting performance, making scalable, affordable middlebox processing accessible to enterprise networks of every size.

In this chapter, we illustrated that outsourcing was feasible and beneficial for enterprises. In the following two chapters, we show (a) how software utility computing and new algorithms can provide strong correctness guarantees for middlebox applications with low performance overheads (Chapter 4), and (b) how to use functional cryptography to ameliorate privacy concerns surrounding cloud services (Chapter 5).

Chapter 4

Fault-Tolerance For Middleboxes

We saw in Chapter 2 that failures are a common source of problems for network administrators, many of whom lack resources or mechanisms to implement automatic recovery. In the previous chapter we mentioned that outsourcing can provide the illusion of infinite resources for clients, providing redundancy for failover as needed. In this chapter, we discuss how to implement failover. Importantly, we aim to avoid relying on custom, per-middlebox solutions (*e.g.* one approach for IDSes and another for WAN optimizers), aiming instead for a generic fault-tolerance approach that is suitable to arbitrary packet processers.

In traditional deployments, the common approach to fault tolerance in middleboxes is a combination of careful engineering to avoid faults, and deploying a backup appliance to rapidly restart when faults occur. Unfortunately, neither of these approaches – alone or in combination – are ideal. With traditional middleboxes, each "box" is developed by a single vendor and dedicated to a single application. This allows vendors greater control in limiting the introduction of faults by, for example, running on hardware designed and tested for reliability (ECC, proper cooling, redundant power supply, *etc.*). This approach will not apply to to software middlebox deployments in a cloud provider, where developers have little control over the environment in which their applications run and vendor diversity in hardware and applications will explode the test space.

The second part to how operators handle middlebox failure is also imperfect. With current middleboxes, operators often maintain a dedicated per-appliance backup. This is inefficient (requiring 1:1 redundancy) and offers only a weak form of recovery for the many middlebox applications that are stateful – *e.g.*, Network Address Translators (NATs), WAN Optimizers, and Intrusion Prevention Systems all maintain dynamic state about flows, users, and network conditions. With no mechanism to recover state, the backup may be unable to correctly process packets after failure, leading to service disruption. (We discuss this further in §4.2.2 and quantify disruption in §5.7.)

In this chapter, we aim to design middleboxes that guarantee correct recovery from failures. This solution must be low-latency (*e.g.*, the additional per-packet latency under failure-free conditions must be well under 1ms) and recovery must be fast (*e.g.*, in less than typical transport timeout values). To the best of our knowledge, no existing middlebox design satisfies these goals. In addition, we would prefer a solution that is general (*i.e.*, can be applied across applications rather than having to be designed on a case-by-case basis for each individual middlebox) and passive (*i.e.*, does not require one dedicated backup per middlebox).

Our solution – FTMB– introduces new algorithms and techniques that tailor the classic approach of rollback recovery to the middlebox domain and achieves correct recovery in a general and passive manner. Our prototype implementation introduces low additional latency on failure-free operation (adding only 30μs to median per-packet latencies, an improvement of 2-3 orders of magnitude over existing fault tolerance mechanisms) and achieves rapid recovery (reconstructing lost state in between 40-275ms for practical system configurations).

4.1 Problem Space

We present our system and failure model (§4.1.1 and §4.1.2) and the challenges in building fault-tolerant middleboxes (§4.1.3).

4.1.1 System Model

Parallel implementations: We assume middlebox applications are multi-threaded and run on a multicore CPU (Figure 4.1). The middlebox runs with a fixed number of threads. We assume 'multi-queue' NICs that offer multiple transmit and receive queues that are partitioned across threads. Each thread reads from its own receive queue(s) and writes to its own transmit queue(s). The NIC partitions packets across threads by hashing a packet's flow identifier (*i.e.*, 5-tuple including source and destination port and address) to a queue; hence all packets from a flow are processed by the same thread and a packet is processed entirely by one thread. The above are standard approaches to parallelizing traffic processing in multicore systems [74, 97, 141, 124].
Shared state: By shared state we mean state that is accessed across threads. In our parallelization approach, all packets from a flow are processed by a single thread so per-flow state

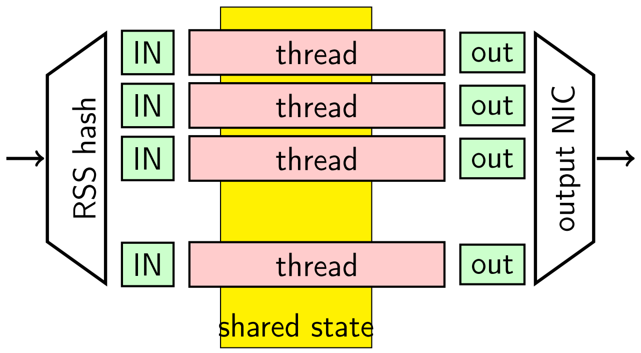


Figure 4.1: Our model of a middlebox application

is local to a single thread and is not shared state. However, other state may be relevant to multiple flows, and accesses to such state may incur cross-thread synchronization overheads. Common forms of shared state include aggregate counters, IDS state machines, rate limiters, packet caches for WAN optimizers, *etc.*

Virtualization: Finally, we assume the middlebox code is running in a virtualized mode. The virtualization need not be a VM per se; we could use containers [14], lightweight VMs [114], or some other form of compartmentalization that provides isolation and supports low-overhead snapshots of its content.

4.1.2 Failure Model

We focus on recovery from "fail-stop" (rather than Byzantine) errors, where under failure 'the component changes to a state that permits other components to detect that a failure has occurred and then stops' [140]. This is the standard failure model assumed by virtual machine fault tolerance approaches like Remus [69], Colo [76], and vSphere [35].

Our current implementation targets failures at the virtualization layer and below, down to the hardware.[1] Our solutions – and many of the systems we compare against – thus cope with failures in the system hardware, drivers, or host operating system. According to a recent study (see Figure 13 in [128]), hardware failures are quite common (80% of firewall failures, 66% of IDS failures, 74% of Load Balancer failures, and 16% of VPN failures required some form of hardware replacement), so this failure model is quite relevant to operational systems.

4.1.3 Challenges

Middlebox applications exhibit three characteristics that, in combination, make fault-tolerance a challenge: statefulness, very frequent non-determinism, and low packet-processing latencies.

As mentioned earlier, many middlebox applications are *stateful* and the loss of this state can degrade performance and disrupt service. Thus, we want a failover mechanism that correctly restores state such that future packets are processed as if this state were never lost (we define correctness rigorously in §4.2.1). One might think that this could be achieved via 'active:active' operation, in which a 'master' and a 'replica' execute on all inputs but only the master's output is released to users. However, this approach fails when system execution is *non-deterministic*, because the master and replica might diverge in their internal state and produce an incorrect recovery.[2]

Non-determinism is a common problem in parallel programs when threads 'race' to access shared state: the order in which these accesses occur depends on hard-to-control effects (such

[1]In §4.7, we discuss how emerging 'container' technologies would allow us to extend our failure model to recover from failures in the guest OS. With such extensions in place, the only errors that we would be unable to recover from are those within the middlebox application software itself.

[2]Similarly, such non-determinism prevents replicated state machine techniques from providing recovery in this context.

as the scheduling order of threads, their rate of progress, *etc.*) and are thus hard to predict. Unfortunately, as mentioned earlier, shared state is common in middlebox applications, and shared state such as counters, caches or address pools may be accessed on a per-packet or per-flow basis leading to frequent nondeterminism.[3] In addition, non-determinism can also arise because of access to hardware devices, including clocks and random number generators, whose return values cannot be predicted. FTMB must cope with all of these sources of nondeterminism.

As we elaborate on shortly, the common approach to accommodating non-determinism is to intercept and/or record the outcome of all potentially non-deterministic operations. However, such interception slows down normal operation and is thus at odds with the other two characteristics of middlebox applications, namely *very* frequent accesses to shared state and low packet processing latencies. Specifically, a piece of shared state may be accessed 100k-1M times per second (the rate of packet arrivals), and the latency through the middlebox should be in 10-100s of microseconds. Hence mechanisms for fault-tolerance must support high access rates and introduce extra latencies of a similar magnitude.

4.2 Goals and Design Rationale

Building on the previous discussion, we now describe our goals for FTMB (§4.2.1), some context (§4.2.2), and the rationale for the design approach we adopt (§4.2.3)

4.2.1 Goals

A fault-tolerant middlebox design must meet the three requirements that follow.
(1) Correctness. The classic definition of correct recovery comes from Strom and Yemeni [149]: "A system recovers correctly if its internal state after a failure is consistent with the observable behavior of the system before the failure." It is important to note that reconstructed state need not be identical to that before failure. Instead, it is sufficient that the reconstructed state be one that *could* have generated the interactions that the system has already had with the external world. This definition leads to a necessary condition for correctness called "**output commit**", which is stated as follows: no output can be released to the external world until all the information necessary to recreate internal state consistent with that output has been committed to stable storage.

As we discuss shortly, the nature of this necessary information varies widely across different designs for fault-tolerance as does the manner in which the output commit property is enforced. In the context of middleboxes, the output in question is a packet and hence to meet the output commit property we must ensure that, before the middlebox transmits a packet p, it has successfully logged to stable storage all the information needed to recreate internal state consistent with an execution that would have generated p.

[3] We evaluate the effects of such non-determinism in §5.7.

(2) Low overhead on failure-free operation. We aim for mechanisms that introduce no more than 10-100s of microseconds of added delay to packet latencies.
(3) Fast Recovery. Finally, recovery from failures must be fast to prevent degradation in the end-to-end protocols and applications. We aim for recovery times that avoid endpoint protocols like TCP entering timeout or reset modes.

In addition, we seek solutions that obey the following two supplemental requirements:
(4) Generality. We prefer an approach that does not require complete rewriting of middlebox applications nor needs to be tailored to each middlebox application. Instead, we propose a single recovery mechanism and assume access to the source code. Our solution requires some annotations and and automated modifications to this code. Thus, we differ from some recent work [132, 133] in not introducing an entirely new programming model, but we cannot use completely untouched legacy code. Given that many middlebox vendors are moving their code from their current hardware to software implementations, small code modifications of the sort we require may be a reasonable middle ground.
(5) Passive Operation. We do not want to require dedicated replicas for each middlebox application, so instead we seek solutions that only need a passive replica that can be shared across active master instances.

4.2.2 Existing Middleboxes

To our knowledge, no middlebox design in research or deployment simultaneously meets the above goals.[4]

In research, Pico[132] was the first to address fault-tolerance for middleboxes. Pico guarantees correct recovery but does so at the cost of introducing non-trivial latency under failure-free operation – adding on the order of 8-9ms of delay per packet. We describe Pico and compare against it experimentally in §5.7.

There is little public information about what commercial middleboxes do and therefore we engaged in discussions with two different middlebox vendors. From our discussions, it seems that vendors do rely heavily on simply engineering the boxes to not fail (which is also the only approach one can take without asking customers to purchase a separate backup box). For example, one vendor uses only a single line of network interface cards and dedicates an entire engineering team to testing new NIC driver releases.

Both vendors confirmed that shared state commonly occurs in their systems. One vendor estimated that with their IDS implementation, a packet touches 10s of shared variables per packet, and that even their simplest devices incur at least one shared variable access per packet.

Somewhat to our surprise, both vendors strongly rejected the idea of simply resetting all active connections after failure, citing concerns over the potential for user-visible disruption to applications (we evaluate cases of such disruption in §5.7). Both vendors do attempt stateful recovery but their mechanisms for this are ad-hoc and complex, and offer no correctness

[4]Traditional approaches to reliability for routers and switches do little to address statefulness as there is no need to do so, and thus we do not discuss such solutions here.

guarantee. For example, one vendor partially addresses statefulness by checkpointing select data structures to stable storage; since checkpoints may be both stale and incomplete (*i.e.*, not all state is checkpointed) they cannot guarantee correct recovery. After recovery, if an incoming packet is found to have no associated flow state, the packet is dropped and the corresponding connection reset; they reported using a variety of application-specific optimizations to lower the likelihood of such resets. Another vendor offers an 'active:active' deployment option but they do not address non-determinism and offer no correctness guarantees; to avoid resetting connections their IDS system 'fails open' – *i.e.*, flows that were active when the IDS failed bypass some security inspections after failure.

Both vendors expressed great interest in general mechanisms that guarantee correctness, saying this would both improve the quality of their products and reduce the time their developers spend reasoning through the possible outcomes of new packets interacting with incorrectly restored state.

However, both vendors were emphatic that correctness could not come at the cost of added latency under failure-free operation and independently cited 1ms as an upper bound on the latency overhead under failure-free operation.[5] One vendor related an incident where a trial product that added 1-2ms of delay per-packet triggered almost 100 alarms and complaints within the hour of its deployment.

Finally, both vendors emphasized avoiding the need for 1:1 redundancy due to cost. One vendor estimated a price of \$250K for one of their higher-grade appliances; the authors of [141] report that a large enterprise they surveyed deployed 166 firewalls and over 600 middleboxes in total, which would lead to multi million dollar overheads if the dedicated backup approach were applied broadly.

4.2.3 Design Options

Our goal is to provide stateful recovery that is correct in the face of nondeterminism, yet introduces low delay under both failure-free and post-failure operation. While less explored in networking contexts, stateful recovery has been extensively explored in the general systems literature. It is thus natural to ask what we might borrow from this literature. In this section, we discuss this prior work in broad terms, focusing on general approaches rather than specific solutions, and explain how these lead us to the approach we pursued with FTMB. We discuss specific solutions and experimentally compare against them in §5.7.

At the highest level approaches to stateful recovery can be classified based on whether lost state is reconstructed by *replaying* execution on past inputs. As the name suggests, solutions based on 'replay' maintain a log of inputs to the system and, in the event of a failure, they recreate lost state by replaying the inputs from the log; in contrast, 'no-replay' solutions do not log inputs and never replay past execution.

As we will discuss in this section, we reject no-replay solutions because they introduce high latencies on per-packet forwarding – on the order of many milliseconds. However, replay-

[5]This is also consistent with carrier requirements from the Broadband Forum which cite 1ms as the upper bound on forwarding delay (through BGN appliances) for VoIP and other latency-sensitive traffic[39].

based approaches have their own challenges in sustaining high throughput given the output frequency of middleboxes. FTMB follows the blueprint of rollback-recovery, but introduces new algorithms for logging and output commit that can sustain high throughput.

4.2.4 No-Replay Designs

No-replay approaches are based on the use of system checkpoints: processes take periodic "snapshots" of the necessary system state and, upon a failure, a replica loads the most recent snapshot. However, just restoring state to the last snapshot does not provide correct recovery since all execution beyond the last snapshot is lost – *i.e.*, the output commit property would be violated for all output generated after the last snapshot. Hence, to enforce the output commit property, such systems buffer all output for the duration between two consecutive snapshots[69]. In our context, this means packets leaving the middlebox are buffered and not released to the external world until a checkpoint of the system up to the creation of the last buffered packet has been logged to stable storage.

Checkpoint-based solutions are simple but delay outputs even under failure-free operation; the extent of this delay depends on the overhead of (and hence frequency between) snapshots. Several efforts aim to improve the efficiency of snapshots – *e.g.*, by reducing their memory footprint[132], or avoiding snapshots unless necessary for correctness[76]. Despite these optimizations, the latency overhead that these systems add – in the order of many milliseconds – remains problematically high for networking contexts. We thus reject no-replay solutions.

4.2.5 Replay-Based Designs

In replay-based designs, the inputs to the system are logged along with any additional information (called 'determinants') needed for correct replay in the face of non-determinism. On failure, the system simply replays execution from the log. To reduce replay time and storage requirements these solutions also use periodic snapshots as an optimization: on failure, replay begins from the last snapshot rather than from the beginning of time. Log-based replay systems can release output without waiting for the next checkpoint so long as all the inputs and events on which that output depends have been successfully logged to stable storage. This reduces the latency sensitive impact on failure-free operation making replay-based solutions better suited for FTMB.

Replay-based approaches to system *recovery* should not be confused with replay-based approaches to *debugging*. The latter has been widely explored in recent work for debugging multicore systems [156, 43, 110]. However, debugging systems do not provide mechanisms for output commit, the central property needed for correct recovery – they do not need to, since their aim is not to resume operation after failure. Consequently, these systems cannot be used to implement high availability. [6]

[6]A second question is whether or not we can adopt logging and instrumentation techniques from these systems to detect determinants. However, as we discuss experimentally in §5.7, most debugging approaches

Instead, the most relevant work to our goals comes from the classic distributed systems literature from the 80s and 90s, targeting rollback-recovery for multi-process distributed systems (see [80] for an excellent survey). Unfortunately, because of our new context (a single multi-threaded server, rather than independent processes over a shared network) and performance constraints (output is released every few microseconds or nanoseconds rather than seconds or milliseconds), existing algorithms from this literature for logging and output commit cannot sustain high throughput.

With all recovery approaches, the system must check that all determinants – often recorded in the form of vector clocks [111] or dependency trees [79] – needed for a given message to be replayed have been logged before the message may be released. This check enforces the output commit property. In systems which follow an *optimistic logging* approach, this output commit 'check' requires *coordination* between all active process/threads every time output is released. This coordination limits parallelism when output needs to be released frequently. For example, in §5.6 we discuss a design we implemented following the optimistic approach which could sustain a maximum throughput of only 600Mbps (where many middleboxes process traffic on the order of Gbps) due to frequent cross-core coordination. Other systems, which follow a *causal logging* approach, achieve coordination-free output commit and better parallelism, but do so by permitting heavy redundancy in what they log: following the approach of one such causal system [79], we estimated that the amount of logged determinants would reach between 500Gbps-300Tbps just for a 10Gbps of packets processed on the dataplane. Under such loads, the system would have to devote far more resources to recording the logs themselves than processing traffic on the dataplane, once again limiting throughput.

Hence, instead of following a standard approach, we instead designed a new logging and output commit approach called *ordered logging with parallel release*. In the following section, we describe how our system works and why ordered logging with parallel release overcome the issues presented by previous approaches.

4.3 Design

FTMB is a new solution for rollback recovery, tailored to the middlebox problem domain through two new solutions:

1. 'ordered logging': an efficient mechanism for representing and logging the information required for correct replay; ordered logging represents information in such a way that it is easy to verify the output commit property.

2. 'parallel release': an output commit algorithm that is simple and efficient to implement on multicore machines.

rely on heavyweight instrumentation (*e.g.*, using memory protection to intercept access to shared data) and often logging data that is unnecessary for our use cases (*e.g.*, all calls to malloc) – this leads to unnecessarily high overheads.

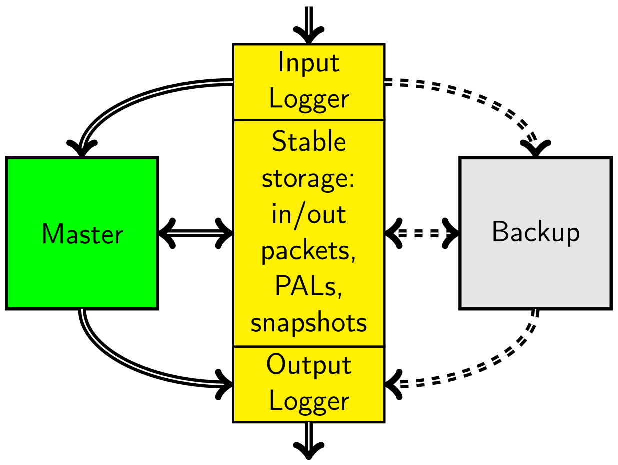


Figure 4.2: Architecture for FTMB.

The architecture of FTMB is shown in Figure 4.2. A *master* VM runs the middlebox application(s), with two *loggers* that record its input and output traffic. Periodic system snapshots are sent from the master to stable storage, and used to start a *backup* in case the master crashes. In our prototype, the master and backup are two identical servers; the input and output loggers are software switches upstream and downstream from the master node; and the stable storage is volatile memory at the downstream switch – the storage is 'stable' in that it will survive a failure at the master, even though it would not survive failure of the switch it resides on. [7]

As explained in earlier sections, the crux of ensuring correct recovery is enforcing the output commit property which, for our context, can be stated as: *do not release a packet until all information needed to replay the packet's transmission has been logged to stable storage.* Enforcing this property entails answering the following questions:

- What information must we log to resolve potential nondeterminism during replay? In the language of rollback recovery protocols this defines what the literature calls *determinants.*
- How do we log this information efficiently? This specifies how we *log* determinants.
- What subset of the information that we log is a given packet dependent on for replay? This defines an output's *dependencies.*
- How do we efficiently check when an individual packet's dependencies have been logged to stable storage? This specifies how we check whether the *output commit* requirements for an output have been met.

[7]There is some flexibility on the physical placement of the functions; our system can withstand the failure of either the middlebox (Master/Backup) or the node holding the saved state but not both simultaneously. We envisage the use of "bypass" NICs that fail open on detecting failure, to survive failures at the loggers[13].

We now address each question in turn and present the architecture and implementation of the resultant system in §5.6.

4.3.1 Defining Determinants

Determinants are the information we must record in order to correctly replay operations that are vulnerable to nondeterminism. As discussed previously, nondeterminism in our system stems from two root causes: races between threads accessing shared variables, and access to hardware whose return values cannot be predicted, such as clocks and random number generators. We discuss each of them below.

Shared State Variables. Shared variables introduce the possibility of nondeterministic execution because we cannot control the order in which threads access them.[8] We thus simply record the order in which shared variables are accessed, and by whom.

Each shared variable v_j is associated with its own lock and counter. The lock protects accesses to the variable, and the counter indicates the order of access. When a thread processing packet p_i accesses a shared variable v_j, it creates a tuple called Packet Access Log (PAL) that contains $(p_i, n_{ij}, v_j, s_{ij})$ where n_{ij} is the number of shared variables accessed so far when processing p_i, and s_{ij} is the number of accesses received so far by v_j.

As an example, figure 4.3 shows the PALs generated by the four threads (horizontal lines) processing packets A, B, C, D. For packet B, the thread first accesses variable X (which has previously been accessed by the thread processing packet A), and then variable Y (which has previously been accessed by the thread processing packet C).

Note that PALs are created independently by each thread, while holding the variable's lock, and using information (the counters) that is either private to the thread or protected by the lock itself.

Shared pseudorandom number generators are treated in the same way as shared variables, since their behavior is deterministic based on the function's seed (which is initialized in the same way during a replay) and the access order recorded in the PALs.

Clocks and other hardware. Special treatment is needed for hardware resources whose return values cannot be predicted, such as `gettimeofday()` and `/dev/random`. For these, we use the same PAL approach, but replacing the variable name and access order with the hardware accessed and the value returned. Producing these PALs does not require any additional locking because they only use information local to the thread. Upon replay, the PALs allow us to return the exact value as during the original access.

4.3.2 How to Log Determinants

The key requirement for logging is that PALs need to be on stable storage (on the Output Logger) before we release the packets that depend on them. While there are many options

[8] Recent research[68, 71] has explored ways to reduce the performance impact of enforcing deterministic execution but their overheads remain impractically high for applications with frequent nondeterminism.

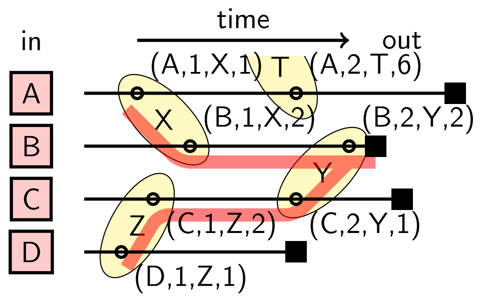


Figure 4.3: Four threads (black lines) process packets A, B, C, D. As time goes (left to right), they access (circles) shared variables X, Y, Z, T generating the PALs in parentheses. The red tree indicates the dependencies for packet B.

for doing so, we pursue a design that allows for fine-grained and correct handling of dependencies.

We make two important design decisions for how logging is implemented. The first is that PALs are *decoupled* from their associated data packet and communicated separately to the output logger. This is essential to avoid introducing unnecessary dependencies between packets. As an example, packet B in the figure depends on PAL $(A, 1, X, 1)$, but it need not be delayed until the completion of packet A, (which occurs much later) – it should only be delayed until $(A, 1, X, 1)$ has been logged.

The second decision has to do with *when* PALs are placed in their outgoing PAL queue. **We require that PALs be placed in the output queue before releasing the lock associated to the shared variable they refer to**. This gives two guarantees: i) when p_i is queued, all of its PALs are already queued; and ii) when a PAL for v_j is queued, all previous PALs for the same variable are already in the output queues for this or other threads. We explain the significance of these properties when we present the output commit algorithm in §4.3.4.

4.3.3 Defining a Packet's Dependencies

During the replay, the replica must evolve in the same way as the master. For a shared variable v_j accessed while processing p_i, this can happen only if i) the variable has gone through the same sequence of accesses, and ii) the thread has the same internal state. These conditions can be expressed recursively in terms of the PALs: each PAL (p_i, n, v_j, m) in turn has up to two dependencies: one **per-packet** $(p_i, n-1, v_k, s_{ik})$, ***i.e.***, on its predecessor PAL for p_i, and one **per-variable** $(p_{i'}, n', v_j, m-1)$, ***i.e.***, on its predecessor PAL for v_j, generated by packet $p_{i'}$. A packet depends on its last PAL, and from that we can generate the tree of dependencies; as an example, the red path in the figure represents the dependencies for packet B.

We should note that the recursive dependency is essential for correctness. If, for instance, packet B in the figure were released without waiting for the PAL $(D, 1, Z, 1)$, and the thread generating that PAL crashed, during the replay we could not adequately reconstruct the state of the shared variables used while processing packet B.

4.3.4 Output Commit

We now develop an algorithm that ensures we do not release p_i until all PALs corresponding to p_i's dependencies have arrived at the output logger. This output commit decision is implemented at the output logger. The challenge in this arises from the parallel nature of our system. Like the master, our output logger is multi-threaded and each thread has an independent queue. As a result, the PALs corresponding to p_i's dependencies may be distributed across multiple per-thread queues. We must thus be careful to minimize cache misses and avoid the use of additional synchronization operations.

Rejected Design: Fine-grained Tracking

The straightforward approach would be to explicitly track individual packet and PAL arrivals at the output logger and then release a packet p_i after all of its PAL dependencies have been logged. Our first attempt implemented a 'scoreboard' algorithm that did exactly this at the output logger. We used two matrices to record PAL arrivals: (i) $\text{SEQ}[i, j]$ which stores the sequence number of p_i at v_j and (ii) $\text{PKT}[j, k]$, the identifier of the packet that accessed v_j at sequence number s_k. These data structures contain all the information needed to check whether a packet can be released. We designed a lock-free multi-threaded algorithm that provably released data packets immediately as their dependencies arrived at the middlebox; however, the overhead of cache contention in reading and updating the scoreboard resulted in poor throughput. Given the two matrices described above, we can expect $\text{O}(nc)$ cache misses per packet release, where n is the number of shared variables and c the number of cores (we omit details due to space considerations). Despite optimizations, we find that explicitly tracking dependencies in the above fashion will result in the scoreboard becoming the bottleneck for simple applications.

Parallel release of PALs

We now present a solution that is slightly more coarse-grained, but is amenable to a parallel implementation with very limited overhead. Our key observation here is that the rules chosen to queue PALs and packets *guarantee* that both the **per-packet** and **per-variable** dependencies for a given packet are already queued for release on some thread before the packet arrives at the output queue on its own thread. This follows from the fact that the PAL for a given lock access is always queued *before* the lock is released. Hence, we only need to transfer PALs and packets to the output logger in a way that preserves the ordering between PALs and data packets.

This is achieved with a simple algorithm run between the Master and the Output Logger, illustrated in Fig. 4.4. Each thread on the Master maps 'one to one' to an ingress queue on

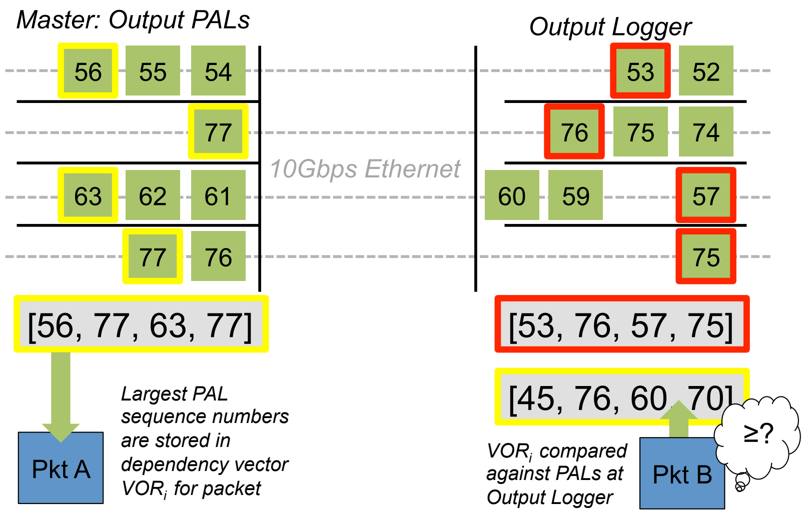


Figure 4.4: Parallel release. Each PAL is assigned a sequence number identifying *when it was generated* within that thread; a packet is released from the output logger if all PALs that were queued before it (on any thread) have been logged.

the Output Logger. PALs in each queue are transferred as a sequential stream (similar to TCP), with each PAL associated to an per-queue sequence number. This replaces the second entry in the PAL, which then does not need to be stored. Each thread at the Master keeps track of MAX, the maximum sequence number that has been assigned to any PAL it has generated.

On the Master: Before sending a data packet from its queue to the output logger, each thread on the master *reads* the current MAX value at all other threads and creates a vector clock VOR which is associated with the packet. It then reliably transfers the pending PALs in its queue, followed by the data packets and associated vector clocks.

On the Output Logger: Each thread continuously receives PALs and data packets, requesting retransmissions in the case of dropped PALs. When it receives a PAL, a thread updates the value MAX representing the highest sequence number such that it has received all PALs prior to MAX. On receiving a data packet, each thread *reads* the value MAX over all other threads, comparing each with the vector clock VOR. Once all values $\text{MAX}_i \geq \text{VOR}_i$, the packet can be released.

Performance

Our parallel release algorithm is efficient because i) threads on the master and the output logger can run in parallel; ii) there are no write-write conflicts on the access to other queues, so memory performance does not suffer much; iii) the check to release a packet requires a very small constant time operation; iv) when batching is enabled, all packets released by the master in the same batch can use the same vector clock, resulting in very small overhead on the link between the master and the output logger and amortizing the cost of the 'check' operation.

4.4 System Implementation

We present key aspects of our implementation of FTMB. For each, we highlight the performance implications of adding FTMB to a regular middlebox through qualitative discussion and approximate back-of-the-envelope estimates; we present experimental results with our prototype in §5.7.

The logical components of the architecture are shown in Figure 4.2. Packets flow from the Input Logger (IL), to the Master (M), to the Output Logger (OL). FTMB also needs a Stable Storage (SS) subsystem with enough capacity to store the state of the entire VM, plus the packets and PALs accumulated in the IL and OL between two snapshots. In our implementation the IL, OL and SS are on the same physical machine, which is expected to survive when M crashes.

To estimate the amount of storage needed we can assume a snapshot interval in the 50–200 ms range (§5.7), and input and output traffic limited by the link's speed (10–40 Gbit/s). We expect to cope with a large, but not overwhelming PAL generation rate; *e.g.*, in the order of 5 M PALs/s (assuming an input rate of 1.25M packets/second and 5 shared state accesses per packet).

4.4.1 Input Logger

The main role of the IL is to record input traffic since the previous snapshot, so that packets can be presented in the same order to the replica in case of a replay.

The input NIC on the IL can use standard mechanisms (such as 5-tuple hashing on multiqueue NICs) to split traffic onto multiple queues, and threads can run the IL tasks independently on each queue. Specifically, on each input queue, the IL receives incoming packets, assigns them sequence numbers, saves them into stable storage, and then passes them reliably to the Master.

Performance implications: The IL is not especially CPU intensive, and the bandwidth to communicate with the master or the storage is practically equal to the input bandwidth: the small overhead for reliably transferring packets to the Master is easily offset by aggregating small frames into MTU-sized segments.

It follows that the only effect of the IL on performance is the additional (one way) latency for the extra hop the traffic takes, which we can expect to be in the 5–10μs range[85].

4.4.2 Master

The master runs a version of the Middlebox code with the following modifications:

- the input must read packets from the reliable stream coming from the IL instead of individual packets coming from a NIC;
- the output must transfer packets to the output queue instead of a NIC.
- access to shared variables is protected by locks, and includes calls to generate and queue PALs;
- access to special hardware functions (timers, etc.) also generates PALs as above.

A shim layer takes care of the first two modifications; for a middlebox written using Click, this is as simple as replacing the `FromDevice` and `ToDevice` elements. We require that developers annotate shared variables at the point of their declaration. Given these annotations, we automate the insertion of the code required to generate PALs using a custom tool inspired by generic systems for data race detection [139].

Our tool uses LLVM's [112] analysis framework (also used in several static analysis tools including the Clang Static Analyzer [9] and KLEE [59]) to generate the call graph for the middlebox. We use this call graph to record the set of locks held while accessing each shared variable in the middlebox. If all accesses to the shared variable are protected by a common lock, we know that there are no contended accesses to the variable and we just insert code to record and update the PAL. Otherwise we generate a "protecting" lock and insert code that acquires the lock before any accesses, in addition to the code for updating the PALs. Note that because the new locks never wrap another lock (either another new lock or a lock in the original source code), it is not possible for this instrumentation to introduce deadlocks [48, 66]. Since we rely on static analysis, our tool is conservative, *i.e.* it might insert a protecting lock even when none is required.

FTMB is often compatible with lock-free optimizations. For example, we implemented FTMB to support seqlocks [37], which are used in multi-reader/single-writer contexts. seqlocks use a counter to track what 'version' of a variable a reader accessed; this version number replaces s_{ij} in the PAL.

Performance implications: the main effect of FTMB on the performance of the Master is the cost of PAL generation, which is normally negligible unless we are forced to introduce additional locking in the middlebox.

4.4.3 Output Logger

The Output Logger cooperates with the Master to transfer PALs and data packets and to enforce output commit. The algorithm is described in §4.3.4. Each thread at M transports packets with a unique header such that NIC hashing at OL maintains the same affinity, enforcing a one-to-one mapping between an eggress queue on M to an ingress queue on OL.

The traffic between M and OL includes data packets, plus additional information for PALs and vector clocks. As a very coarse estimate, even for a busy middlebox with a total of 5 M PALs and vector clocks per second, assuming 16 bytes per PAL, 16 bytes per vector clock, the total bandwidth overhead is about 10% of the link's capacity for a 10 Gbit/s link.

Performance implications: once again the impact of FTMB on the OL is more on latency than on throughput. The minimum latency to inform the OL that PALs are in stable storage is the one-way latency for the communication. On top of this, there is an additional latency component because our output commit check requires *all queued PALs* to reach the OL before the OL releases a packet. In the worst case a packet may find a full PAL queue when computing its vector clock, and so its release may be delayed by the amount of time required to transmit a full queue of PALs. Fortunately, the PAL queue can be kept short *e.g.*, 128 slots each, without any adverse effect on the system (PALs can be sent to the OL right away; the only reason to queue them is to exploit batching). For 16-byte PALs, it takes less than 2μs of link time to drain one full queue, so the total latency introduced by the OL and the output commit check is in the 10-30μs range.

4.4.4 Periodic snapshots

FTMB takes periodic snapshots of the state of the Master, to be used as a starting point during replay, and avoid unbounded growth of the replay time and input and output logs size. Checkpointing algorithms normally freeze the VM completely while taking a snapshot of its state.
Performance implications: The duration of the freeze, hence the impact on latency, has a component proportional to the number of memory pages modified between snapshots, and inversely proportional to bandwidth to the storage server. This amounts to about 5μs for each 4 Kbyte page. on a 10 Gbit/s link, and quickly dominates the fixed cost (1-2ms) for taking the snapshot. However, a worst case analysis is hard as values depend on the (wildly variable) number of pages modified between snapshots. Hence it is more meaningful to gauge the additional latency from the experimental values in §5.7 and the literature in general[69].

4.4.5 Replay

Finally, we describe our implementation of replay, when a Replica VM starts from the last available snapshot to take over a failed Master. The Replica is started in "replay mode", meaning that the input is fed (by the IL) from the saved trace, and threads use the PALs to drive nondeterministic choices.

On input, the threads on the Replica start processing packets, discarding possible duplicates at the beginning of the stream. When acquiring the lock that protects a shared variable, the thread uses the recorded PALs to check whether it can access the lock, or it has to block waiting for some other thread that came earlier in the original execution. The information in the PALs is also used to replay hardware related non deterministic calls (clocks, etc.). Of course, PALs are not generated during the replay.

On output, packets are passed to the OL, which discards them if a previous instance had been already released, or pass it out otherwise (*e.g.*, copies of packets still in the Master when it crashed, even though all of their dependencies had made it to the OL). A thread exits replay mode when it finds that there are no more PALs for a given shared variable. When this happens, it starts behaving as the master, i.e. generate PALs, compute output dependencies, *etc.*

Middlebox	LOC	SVs	Elts	Source
Mazu-NAT	5728	3	46	Mazu Networks [26]
WAN Opt.	5052	2	40	Aggarwal et al. [42]
BW Monitor	4623	251	41	Custom
SimpleNAT	4964	2	42	Custom
Adaptive LB	5058	1	42	Custom
QoS Priority	5462	3	56	Custom
BlindFwding	1914	0	24	Custom

Table 4.1: Click configurations used in our experiments, including Lines of Code (LOC), Shared Variables (SVs), number of Elements (Elts), and the author/origin of the configuration.

Performance implications: other than having to re-run the Middlebox since the last snapshot, operation speed in replay mode is comparable to that in the original execution. §4.5.2 presents some experimental results. Of course, the duration of service unavailability after a failure also depends on the latency of the failure detector, whose discussion is beyond the scope of this paper.

4.5 Evaluation

We added FTMB support into 7 middlebox applications implemented in Click: one configuration comes from industry, five are research prototypes, and one is a simple 'blind forwarding' configuration which performs no middlebox processing; we list these examples in Table 4.1.

Our experimental setup is as follows. FTMB uses Xen 4.2 at the master middlebox with Click running in an OpenSUSE VM, chosen for its support of fast VM snapshotting [24]. We use the standard Xen bridged networking backend; this backend is known to have low throughput and substantial recent work aims to improve throughput and latency to virtual machines, *e.g.*, through netmap+xennet [135, 115] or dpdk+virtio [99, 138]. However, neither of these latter systems yet supports seamless VM migration. We thus built two prototypes: one based on the Xen bridged networking backend which runs at lower rates (100Mbps) but is complete with support for fast VM snapshots and migration, and a second prototype that uses netmap+xennet and scales to high rates (10Gbps) but lacks snapshotting and replay. We primarily report results from our complete prototype; results for relevant experiments with the high speed prototype were qualitatively similar.

We ran our tests on a local network of servers with 16-core Intel Xeon EB-2650 processors at 2.6Ghz, 20MB cache size, and 128GB memory divided across two NUMA nodes. For all experiments shown, we used a standard enterprise trace as our input packet stream [72]; results are representative of tests we ran on other traces.

We first evaluate the FTMB's latency and bandwidth overheads under failure-free operation (§4.5.1). We then evaluate recovery from failure (§4.5.2).

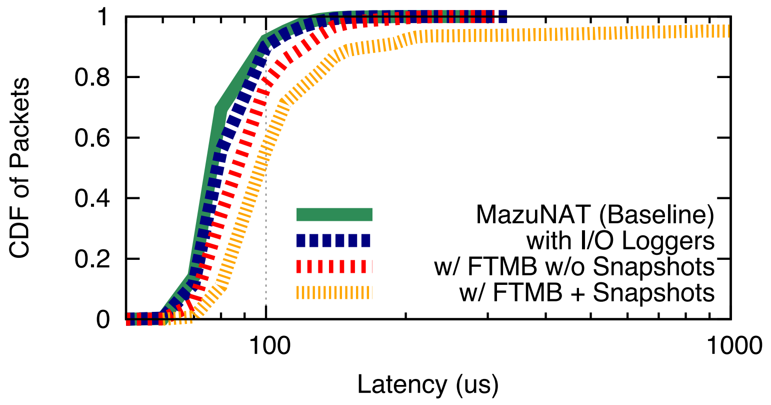


Figure 4.5: Local RTT with and without components of FTMB enabled.

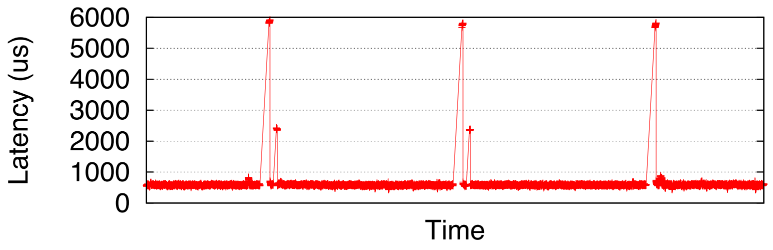


Figure 4.6: Testbed RTT over time.

4.5.1 Overhead on Failure-free Operation

How does FTMB impact packet latency under failure-free operation? In Figure 4.5, we present the per-packet latency through a middlebox over the local network. A packet source sends traffic (over a logging switch) to a VM running a MazuNAT (a combination firewall-NAT released by Mazu Networks [26]), which loops the traffic back to the packet generator. We measure this RTT. To test FTMB, we first show the base latency with (a) just the MazuNAT, (b) the MazuNAT with I/O logging performed at the upstream/downstream switch, (c) the MazuNAT with logging, PAL-instrumented locks, parallel release for the output commit condition and (d) running the MazuNAT with all our fault tolerance mechanisms, including VM checkpointing every 200ms. Adding PAL instrumentation to the middlebox locks in the MazuNAT has a negligible impact on latency, increasing 30μs over the baseline at the median, leading to a 50th percentile latency of 100μs.[9] However, adding VM checkpointing does increase latency, especially at the tail: the 95th %-ile is 810μs, and the 99th %-ile s 18ms.

To understand the cause of this tail latency, we measured latency against time using the Blind Forwarding configuration. Figure 4.6 shows the results of this experiment: we see that the latency spikes are periodic with the checkpoint interval. Every time we take a VM

[9]In similar experiments with our netmap-based prototype we observe a median latency increase of 25μs and 40μs over the baseline at forwarding rates of 1Gbps and 5Gbps respectively, both over 4 cores.

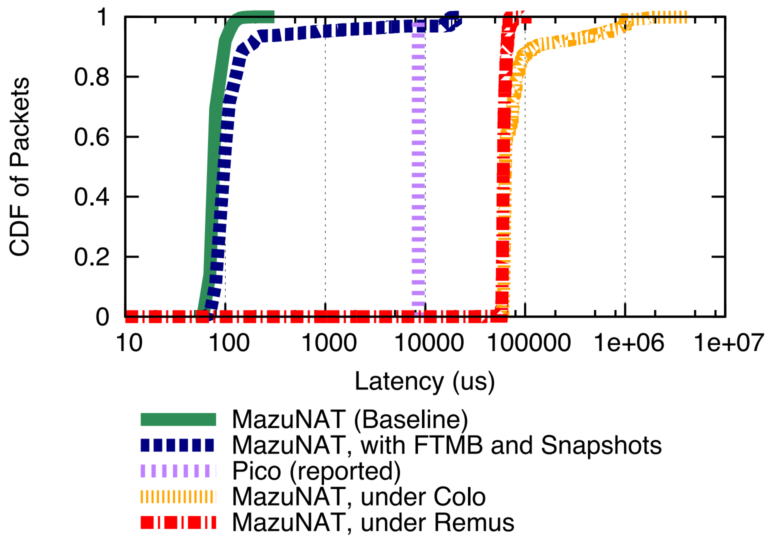


Figure 4.7: Local RTT with FTMB and other FT systems.

snapshot, the virtual machine suspends temporarily, leading to a brief interval where packets are buffered as they cannot be processed. As new hardware-assisted virtualization techniques improve [1, 63] we expect this penalty to decrease with time; we discuss these opportunities further in §4.7.

How does the latency introduced by FTMB compare to existing fault-tolerance solutions? In Figure 4.7, we compare FTMB against three proposals from the research community: Pico [132], Colo [76], and Xen Remus [69]. Remus and Colo are general no-replay solutions which can provide fault tolerance for any VM-based system running a standard operating system under x86. Remus operates by checkpointing and buffering output until the next checkpoint completes; this results in a median latency increase for the MazuNAT by over 50ms. for general applications Colo can offer much lower latency overhead than Remus: Colo allows two copies of a virtual machine to run side-by-side in "lock step". If their output remains the same, the two virtual machines are considered identical; if the two outputs differ, the system forces a checkpoint like Remus. Because multi-threaded middleboxes introduce substantial nondeterminism, though, Colo cannot offer us any benefits over Remus: when we ran the MazuNAT under Colo, it checkpointed just as frequently as Remus would have, leading to an equal median latency penalty.

Pico is a no-replay system similar to Remus but tailored to the middlebox domain by offering a custom library for flow state which checkpoints packet processing state only, but *not* operating system, memory state, *etc.*, allowing for much lighter-weight and therefore faster checkpoint. The authors of Pico report a latency penalty of 8-9ms in their work which is a substantial improvement over Colo and Remus, but still a noticeable penalty due to the reliance on packet buffering until checkpoint completion.

How does inserting PALs increase latency? To measure the impact of PALs over per-

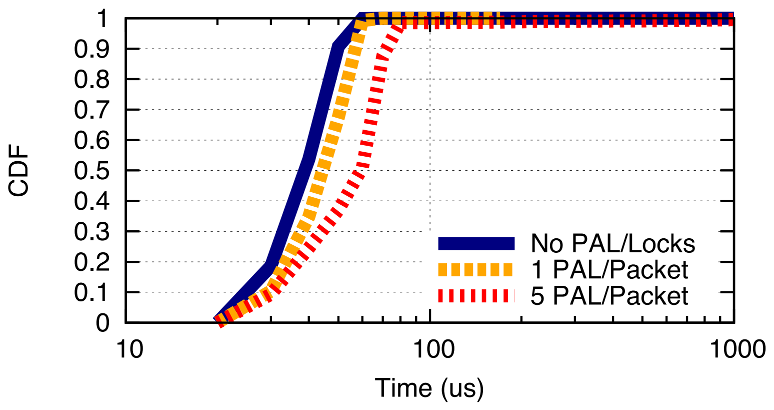


Figure 4.8: Testbed RTT with increasing PALs/packet.

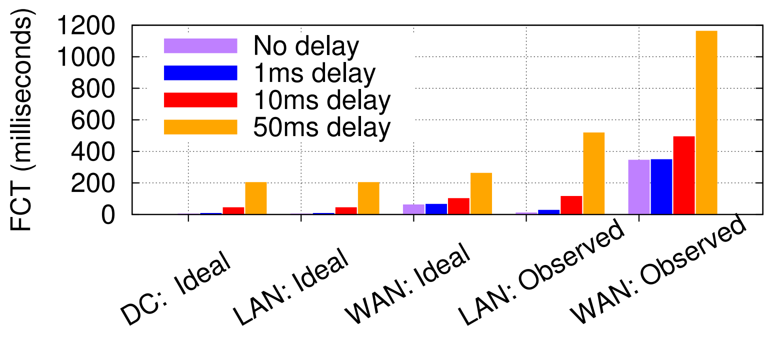


Figure 4.9: Ideal [116] and observed page load times when latency is artificially introduced in the network.

packet latency, we used a toy middlebox with a simple pipeline of 0, 1, or 5 locks and ran measurements with 500-byte packets at 1Gbps with four threads dedicated to processing in our DPDK testbed. Figure 4.8 shows the latency distributions for our experiments, relative to a baseline of the same pipeline with no locks. At 5 PALS/Locks per packet, latency increases to 60μs with 5 PALS/Locks per packet, relative to a median latency under 40μs in the baseline – an increase of on average 4μs per PAL/Lock per packet. Note that this latency figure includes both the cost of PAL creation and lock insertion; the worst case overhead for FTMB is when locks are not already present in the base implementation.

How much does latency matter to application performance? We measured the impact of inflated latency on Flow Completion Times (FCTs) with both measurements and modeling. In Figure 4.9, we show flow completion times for a 2MB flow (representative of web page load sizes) given the flow completion time model by Mittal et al. [116] marked as ‘Ideal’. Marked as ‘Observed’, we downloaded the Alexa top-1000 [2] web pages over a LAN and over a WAN and used `tc` to inflate the latency by the same amounts. In both the datacenter and LAN cases, adding 10ms of latency on the forward and reverse path increases flow completion

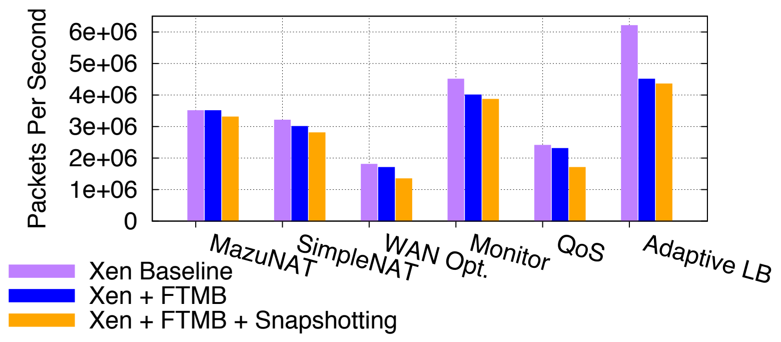


Figure 4.10: Impact of FTMB on forwarding plane throughput.

times to 20× the original in the simulated case; in the experimental LAN case it increased FCT to 10×. In the WAN case, page load times increased to 1.5× by adding 10ms of latency from a median of 343ms to 492ms. An experiment by Amazon shows that every 100ms of additional page load time their customers experienced costs them 1% in sales [106].

Given these numbers in context, we can return to Figure 4.7 and see that solutions based on Colo, Pico, or Remus would *noticeably* harm network users' quality of experience, while FTMB, with introduced latency typically well under 1ms, would have a much weaker impact.

How much does FTMB impact throughput under failure-free operation? Figure 4.10 shows forwarding plane throughput in a VM, in a VM with PAL instrumentation, and running complete FTMB mode with both PAL instrumentations and periodic VM snapshotting. To emphasize the extra load caused by FTMB, we ran the experiment with locally sourced traffic and dropping the output. Even so, the impact is modest, as expected (see §4.4.2). For most configurations, the primary throughput penalty comes from snapshotting rather than from PAL insertion. The MazuNAT and SimpleNat saw a total throughput reduction of 5.6% and 12.5% respectively. However, for the Monitor and the Adaptive Load Balancer, PAL insertion was the primary overhead, causing a 22% and 30% drop in throughput respectively. These two experience a heavier penalty since typically they have no contention for access to shared state variables: the tens of nanoseconds required to generate a PAL for these middleboxes is a proportionally higher penalty than it is for middleboxes which spend more time per-packet accessing complex and contended state.

We ran similar experiments with Remus and Colo, where throughput peaked in the low hundreds of *Kpps*. We also ran experiments with Scribe [110], a publicly-available system for record and replay of general applications, which aims to *automatically* detect and record data races using page protection. This costs about $400us$ per lock access due to the overhead of page faults.[10] Using Scribe, a simple two-threaded Click configuration with a single piece of shared state stalled to a forwarding rate of only 500 packets/second.

[10]Measured using the Scribe demo image in VirtualBox.

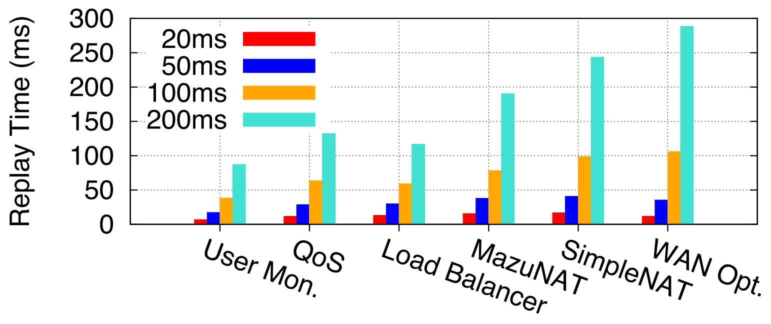


Figure 4.11: Time to perform replay with varying checkpoint intervals and middlebox configurations.

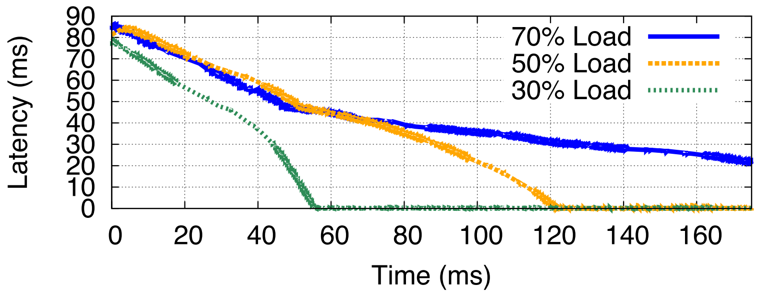


Figure 4.12: Packet latencies post-replay.

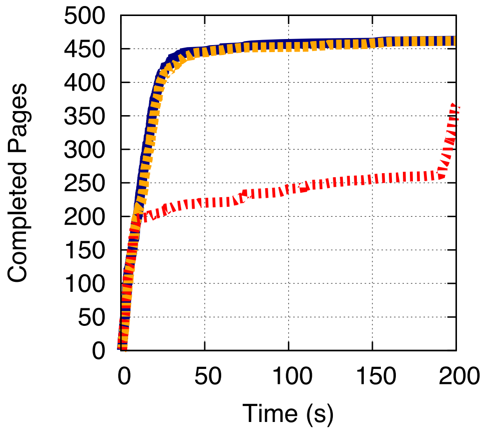
4.5.2 Recovery

How long does FTMB take to perform replay and how does replay impact packet latencies? Unlike no-replay systems, FTMB adds the cost of replay. We measure the amount of time required for replay in Fig. 4.11. We ran these experiments at 80% load (about 3.3 Mpps) with periodic checkpoints of 20, 50, 100, and 200ms.

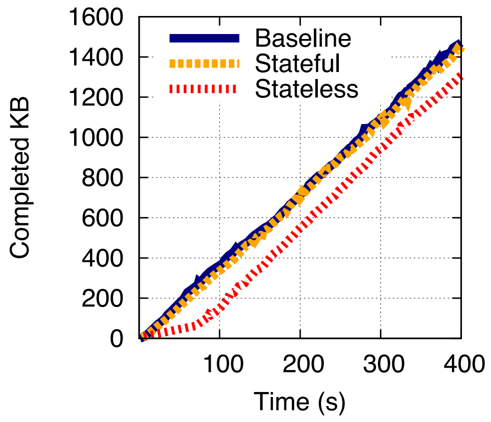
For lower checkpoint rates, we see two effects leading to a replay time that is actually *less* than the original checkpoint interval. First, the logger begins transmitting packets to the replica as soon as replay begins – while the VM is loading. This means that most packets are read pre-loaded to local memory, rather than directly from the NIC. Second, the transmission arrives at almost 100% of the link rate, rather than 80% load as during the checkpoint interval.

However, at 200ms, we see a different trend: some middleboxes that make frequent accesses to shared variable have a *longer* replay time than the original checkpoint interval because of the overhead of replaying lock accesses. Recall that when a thread attempts to access a shared-state variable during replay, it will spin waiting for its 'turn' to access the variable and this leads to slowed execution.

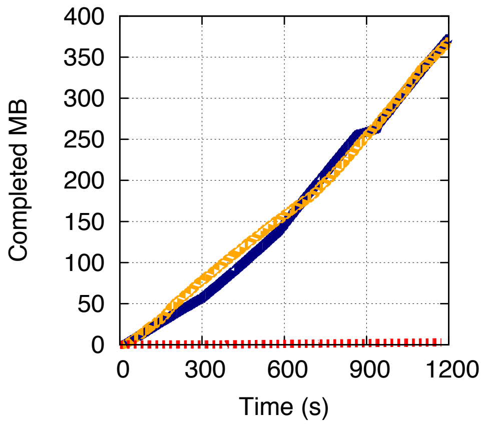
During replay, new packets that arrive must be buffered, leading to a period of increased queueing delays after execution has resumed. In Figure 4.12, we show per-packet latencies for packets that arrive post-failure for MazuNAT at different load levels and replay times between 80-90ms. At 30%-load, packet latencies return to their normal sub-millisecond values within



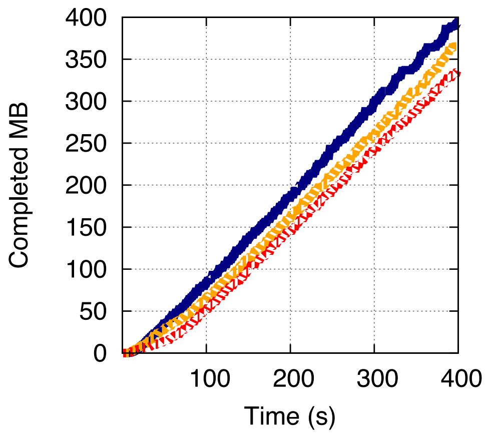
(a) HTTP Page Loads



(b) FTP Download



(c) Torrent (Evolve)



(d) Torrent(Ubuntu)

Figure 4.13: Application performance with and without state restoration after recovery. Key (top right) is same for all figures.

60ms of resumed execution. As expected recovery takes longer at higher loads: at 70% load per-packet latency remains over 10ms even at 175ms, and the latencies do not decrease to under a millisecond until past 300ms after execution has resumed.

Is stateful failover valuable to applications? Perhaps the simplest approach to recovering from failure is simply to bring up a backup from 'cold start', effectively wiping out all connection state after failure: i.e., recovery is *stateless*. To see the impact of stateless recovery on real applications, we tested several applications over the wide area with a NAT which either (a) did not fail (our baseline), (b) went absent for 300ms,[11] during which time traffic was buffered (this represents stateful recovery), or (c) flushed all state on failure (representing stateless recovery). Figure 4.13 shows the time to download 500 pages in a 128-thread loop from the Alexa-top US sites, percentage file completion over time for a large FTP download, and percentage file completion for two separate BitTorrent downloads. In all three configurations, stateful recovery performs close to the performance of the baseline. For stateless recovery over the HTTP connections, we see a sharp knee corresponding to the connection reset time: 180 seconds[12]. The only application with little impact under stateless recovery is one of the BitTorrent downloads – however, the other BitTorrent download failed almost entirely and the client had to be restarted! The torrent which failed had only 10 available peers and, when the connections were reset, the client assumed that the peers had gone offline. The other torrent had a large pool of available peers and hence could immediately reconnect to new peers.

Our point in these experiments is not to suggest that applications are fundamentally incapable of rapid recovery in scenarios of stateless recovery, but simply that many existing applications do not.

4.6 Related Work

We briefly discuss the three lines of work relevant to FTMB, reflecting the taxonomy of related work introduced in §4.1.

First are no-replay schemes. In §5.7 we described in detail three recent systems – Remus, Pico and Colo – that adopt this approach and compared FTMB to them experimentally.

The second are solutions for rollback recovery from the distributed systems literature. The literature includes a wide range of protocol proposals (we refer the reader to Elnozahy et al. [80] for an excellent survey); however, to our knowledge, there is no available system implementation that we can put to the test for our application context.[13] More generally, as mentioned earlier, the focus on distributed systems (as opposed to a parallel program on a single machine) changes the nature of the problem in many dimensions, such as: the failure model (partial vs. complete failure), the nature of non-determinism (primarily the arrival and sending order of messages at a given process vs. threads that 'race' to access the same variable), the frequency of output (for us, outputs are generated at a very high rate) and the

[11] We picked 300ms as a conservative estimate of recovery time; our results are not sensitive to the precise value.

[12] Firefox, Chrome, and Opera have reset times of 300 seconds, 50 seconds, and 115 seconds respectively.

[13] In their survey paper, Elnozahy et al. state that, in practice, log-based rollback-recovery has seen little adoption due to the complexity of its algorithms.

frequency of nondeterminism (per-packet for us), and where the performance bottlenecks lie (for us, in the logging and output commit decision). These differences led us to design new solutions that are simpler and more lightweight than those found in the literature.

The final class of solutions are the multicore record-and-replay systems used for debugging. These do not implement output commit. We discussed these solutions in broad terms in §4.1 and evaluated one such system (Scribe) in §5.7.

In the remainder of this section we briefly review a few additional systems.

Hypervisor-based Fault Tolerance [58] was an early, pioneering system in the 90s to implement fault-tolerance over arbitrary virtual machines; their approach did not address multicore systems, and required synchronization between the master and replica for every nondeterministic operation.

SMP Revirt [77] performs record-and-replay over Xen VMs; unlike FTMB SMPRevirt is hence fully general. As in Scribe, SMP ReVirt uses page protection to track memory accesses. For applications with limited contention, the authors report a 1.2-8x slowdown, but for so-called "racy" applications (like ours) with tens or hundreds of thousands of faults per second we expect results similar to those of Scribe.

Eidetic Systems [70] allow a user to replay any event in the system's history – on the scale of even years. They achieve very low overheads for their target environment: end user desktops. However, the authors explicitly note that their solutions do not scale to racy and high-output systems.

R2 [95] logs a cut in an application's call graph and introduces detailed logging of information flowing across the cut using an R2 runtime to intercept syscalls and underlying libraries; the overhead of their interception makes them poorly suited to our application with frequent nondeterminism.

ODR [43] is a general record-and-replay system that provides output determinism: to reduce runtime overhead ODR foregoes logging all forms of nondeterminism and instead searches the space of possible executions during replay. This can result in replay times that are several orders of magnitude higher than the original execution (in fact, the search space is NP hard). This long replay time is not acceptable for applications looking to recover from a failure (as opposed to debugging post-failure).

4.7 Discussion

In this chapter, we presented FTMB, a system for rollback recovery which uses ordered logging and parallel release for low overhead middlebox fault-tolerance. We showed that FTMB imposes only 30μs of latency for median packets through an industry-designed middlebox. FTMB has modest throughput overheads, and can perform replay recovery in 1-2 wide area RTTs. In outsourced environments, FTMB can implement correct recovery from failure, even when middleboxes are implemented in software and on shared infrastructure.

Chapter 5

Privacy Preserving Middleboxes

Many network middleboxes perform *deep packet inspection* (DPI) to provide a wide range of services which can benefit both end users and network operators. For example, Network Intrusion Detection/Prevention (IDS/IPS) systems (e.g., Snort [136] or Bro [122]) detect if packets from a compromised sender contain an attack. Exfiltration prevention devices block accidental leakage of private data in enterprises by searching for document confidentiality watermarks in the data transferred out of an enterprise network [146]. Parental filtering devices prevent children from accessing adult material in schools, libraries, and homes [33]. These devices and many others [21, 23, 22] all share the common feature that they inspect packet payloads; the market for such DPI devices is expected to grow to over $2B by 2018 [145].

Implementing DPI services entails that the middleboxes operate over unencrypted traffic. The need for unencrypted traffic at times introduces tension between *user privacy* and the *need for security trafic inspection*. For example, in public networks such as cafes and universities, users may desire that their data be kept secret from observers including frin network administrators; today these users can *either* have privacy *or* the network administrators can decrypt the traffic for inspection. In traditional enterprise deployments, however, there is usually no such concern: network users are employees of the enterprise carrying out company business over the network. Any transmissions over the company network are likely to be logged, recorded, and monitored for security, auditing, and company record-keeping. However, in proposing cloud outsourcing, we have now introduced into enterprise networks the same tension between privacy and DPI that troubles public networks because the enterprise and its users may wish to keep their data secret from network service providers at the cloud. Privacy concerns regarding cloud providers are exercerbated by the documented data breaches by cloud employees or hackers [65, 157]. These privacy concerns can act as an obstacle to outsourcing network middleboxes to the cloud.

In this chapter, we demonstrate that it is possible to build a system that provides privacy of the plaintext traffic, while still allowing a third party middlebox provider to implement DPI services. We present *BlindBox*, the first system that provides both the benefits of encryption and functionality at a DPI middlebox. The name “BlindBox” denotes that the middlebox cannot see the private content of traffic. BlindBox keeps data private from any middlebox provider and is applicable both to the 'public network' use case (where users in cafes and universities want privacy from middleboxes in their local network) as well as in the outsourcing scenario (where users and their enterprise administrators wish to keep data secret from the cloud provider).

Our approach is to perform *the inspection directly on the encrypted payload*, without decrypting the payload at the middlebox. Building a practical such system is challenging: networks operate at very high rates requiring cryptographic operations on the critical path to run in micro or even nano seconds; further, some middleboxes require support for rich operations, such as matching regular expressions. A potential candidate is expressive cryptographic schemes such as fully homomorphic or functional encryption [90, 87, 93], but these are prohibitively slow, decreasing network rates by many orders of magnitude.

To overcome these challenges, BlindBox explores and specializes on the network setting. BlindBox enables two classes of DPI computation each having its own privacy guarantees: *exact match privacy* and *probable cause privacy*. Both of BlindBox's privacy models are much stronger than the state-of-the-art "man in the middle" approach deployed today, where traffic is decrypted to enable any processing at all. In both of these models, BlindBox protects the data with strong randomized encryption schemes providing similar security guarantees to the well-studied notion of searchable encryption [147, 103]. Depending on the class of computation, BlindBox allows the middlebox to learn a small amount of information about the traffic to detect rules efficiently.

The first class of computation consists of DPI applications that rely only on exact string matching, such as watermarking, parental filtering, and a limited IDS. Under the associated privacy model, exact match privacy, the middlebox learns at which positions in a flow attack keywords occur; for substrings of the flow that do not match an attack keyword, the middlebox learns virtually nothing.

The second class of computation can support all DPI applications, including those which perform regular expressions or scripting. The privacy model here, probable cause privacy, is a new network privacy model: the middlebox gains the ability to see a (decrypted) individual packet or flow *only if the flow is suspicious*; namely, the flow contains a string that matches a known attack keyword. If the stream is not suspicious, the middlebox cannot see the (decrypted) stream. Hence, privacy is affected *only with a cause.*

BlindBox allows users to select which privacy model they are most comfortable with.

To implement these two models, we developed the following techniques:

- **DPIEnc** and **BlindBox Detect** are a new searchable encryption scheme [147] and an associated fast detection protocol, which can be used to inspect encrypted traffic for certain keywords efficiently. As we explain in §5.2, existing searchable encryption schemes [147, 103, 52] are either deterministic (which can enable fast protocols, but provide weak security) or randomized (which have stronger security, but are slow in our setting). DPIEnc with BlindBox Detect achieve both the speed of deterministic encryption and the security of randomized encryption; detection on encrypted traffic runs as fast as on unencrypted traffic.

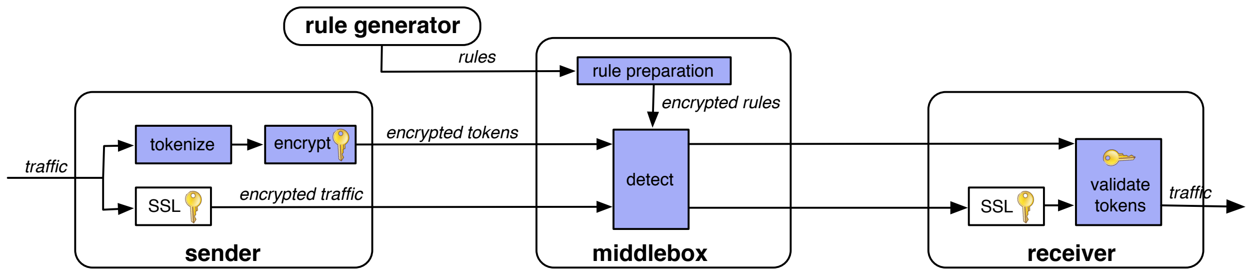


Figure 5.1: System architecture. Shaded boxes indicate algorithms added by BlindBox.

- **Obfuscated Rule Encryption** is a technique to allow the middlebox to obtain encrypted rules based on the rules from the middlebox and the private key of the endpoints, without the endpoints learning the rules or the middlebox learning the private key. This technique builds on Yao's garbled circuits [164] and oblivious transfer [117, 131, 83].
- **Probable Cause Decryption** is a mechanism to allow flow decryption when a suspicious keyword is observed in the flow; this is the mechanism that allows us to implement our probable cause privacy model.

We implemented BlindBox as well as a new secure transport protocol for HTTP, which we call BlindBox HTTPS. We show that BlindBox's performance is practical for many settings. For example, the rate at which the middlebox can inspect packets is as high as 186Mbps per core in our experiments. Given that standard IDS implementations, such as Snort [136], peak at under 100Mbps, this performance is competitive with existing deployments. We achieve this performance due to DPIEnc and BlindBox Detect. When compared to two strawmen consisting of a popular searchable encryption scheme [147] and a functional encryption scheme [104], DPIEnc with BlindBox Detect are 3-6 orders of magnitude faster.

Nevertheless, a component of BlindBox is not yet as fast as desirable: the setup of an HTTPS connection. This setup performs obfuscated rule encryption and it takes time proportional to the number of attack rules. For rulesets with tens of keywords, this setup completes in under a second; however, for large IDS installations with thousands of rules, the setup can take up to 1.5 minutes to complete. Hence, BlindBox is most fit for settings using long or persistent connections through SPDY-like protocols, and not yet practical for short, independent flows with many rules.

5.1 Overview

Fig. 5.1 presents the system architecture. There are four parties: sender (S), receiver (R), middlebox (MB), and rule generator (RG) – these reflect standard middlebox deployments today. RG generates attack rules (also called signatures) to be used by MB in detecting attacks. Each rule attempts to describe an attack and it contains fields such as: one or more keywords to be matched in the traffic, offset information for each keyword, and sometimes regular expressions. The RG role is performed today by organizations like Emerging

Threats [12], McAfee [16], or Symantec [29]. S and R send traffic through MB. MB allows S and R to communicate unless MB observes an attack rule in their traffic.

In today's deployments, MB can read any traffic sent between S and R. With BlindBox, MB should be able to detect if attack rules generated by RG match the traffic between R and S, but should not learn the contents of the traffic that does not match RG's attack rules.

5.1.1 Usage Scenarios

Before formalizing our threat model, we illustrate our usage scenario with three examples. For each individual in these examples, we indicate the party in our model (R, S, MB, or RG) that they correspond to.

Example #1: **University Network**: Alice (R or S) is a student at the University of SIGCOMM and brings her own laptop to her dorm room. However, university policy requires that all student traffic be monitored for botnet signatures and illegal activity by a middlebox (MB) running an IDS. Alice is worried about her computer being infected with botnet software, so she also desires this policy applied to her traffic. McAfee (RG) is the service that provides attack rules to the middlebox and Alice trusts it. However, she is uncomfortable with the idea of someone she doesn't know (who has access to the middlebox) potentially being able to read her private Facebook messages and emails. Alice installs BlindBox HTTPS with McAfee's public key, allowing the IDS to scan her traffic for McAfee's signatures, but not read her private messages.

Example #2: **Enterprise Service**: Bob is an administrator of a small company with many middleboxes. He wants to outsource his middlebox processing to a third-party middlebox service provider as in APLOMB [143], but he doesn't want company secrets revealed to agents at the cloud provider. Within the enterprise, clients always use HTTPS to keep sensitive data encrypted; middleboxes within the enterprise know how to decrypt these streams to scan for malicious or restricted content. Bob wants to outsource these middleboxes, but he doesn't want them to be able to decrypt the content. Bob pushes an update to all company servers, laptops, phones, *etc.*, installing BlindBox HTTPS and Symantec's public key. These *encrypted* HTTP streams are then tunneled to the cloud provider, who searches for Symantec's rules within the encrypted data.

In the above examples, Alice and Bob want to have a middlebox check *for the attack rules the corresponding trusted parties permit*, but the middlebox should not learn anything else about the content of the traffic. A key requirement is that there exists an RG which Alice, Bob and the MB trust with rule generation; if this is not the case, the parties cannot use BlindBox.

Anti-Example #1: **Political Dissident**: Charlie (R or S) is a political dissident who frequently browses sensitive websites, and is concerned about government monitoring. If the government coerces one of MB or RG, Charlie remains protected. However, BlindBox should not be used in a setting in which *both* MB and RG can be controlled by an attacker: in this case, RG can produce signatures for sensitive terms and MB will use these to match the traffic. Hence, if the government can coerce both MB and RG together, Charlie should not use

BlindBox. Similarly, if the government can coerce root certificate generators, Charlie should not use vanilla HTTPS either because it may allow man-in-the-middle attacks on his traffic.

5.1.2 Security and Threat Model

The goal of our work is to protect the privacy of user traffic from MB. Any solution must satisfy a set of systems requirements we discuss in §5.1.2. We then discuss the threat model in §5.1.2 and the privacy guarantees BlindBox provides in §5.1.2.

System Requirements

BlindBox retains key system goals of traditional IDS deployments today: (1) BlindBox must maintain MB's ability to enforce its policies (i.e., detect rules and drop/alert accordingly), and (2) endpoints must not gain access to the IDS rules. The rationale behind the second requirement is twofold. First, in order to make IDS evasion more difficult for an attacker at the user, the rules should be hidden from the endpoints [122]. Second, most vendors (e.g., Lastline and McAfee Stonesoft) rely on the secrecy of their rulesets in their business model, as their value added against competitors often includes more comprehensive, more efficient, or harder to evade rules.

BlindBox maintains these two requirements, and adds an additional one: (3) that the middlebox cannot read the user's traffic, except the portions of the traffic which are considered *suspicious* based on the attack rules.

Threat Model

There are two types of attackers in our setup.

The original attacker considered by IDS: This is the same attacker that traditional (unencrypted) IDS consider and we do not change the threat model here. Our goal is to detect such an attacker over *encrypted* traffic. As in traditional IDS, one endpoint can behave maliciously, but at least one endpoint must be honest. This is a fundamental requirement of any IDS [122] because otherwise two malicious endpoints can agree on a secret key and encrypt their traffic under that key with a strong encryption scheme, making prevention impossible by the security properties of the encryption scheme. Similarly, the assumption that one endpoint is honest is also the default for exfiltration detection and parental filtering today. Parental filters can assume one endpoint is innocent under the expectation that 8-year-olds are unlikely replace their network protocol stack or install tunneling software. Commercial exfiltration detection devices primarily target *accidental* exfiltration (e.g., where an otherwise innocent employee attaches the wrong file to an email), recognizing that deliberate exfiltration requires control of the end host.

The attacker at the middlebox: This is the new attacker in our setting. This attacker tries to subvert our scheme by attempting to extract private data from the encrypted traffic passing through the middlebox. We assume that the middlebox MB performs the detection honestly, but that it tries to learn private data from the traffic and violate the privacy of the endpoints. In particular, we assume that an attacker at MB reads *all* the data accessible to the middlebox, including traffic logs and other state. Given this threat model, BlindBox's goal is to hide the content of the traffic from MB, while allowing MB to do DPI. We do not seek to hide the attack rules from the MB itself; many times these rules are hardcoded in the MB.

Privacy Models

We now describe our privacy models.

Exact Match Privacy gives the following guarantee: the middlebox will be able to discover only those substrings of the traffic that are exact matches for known attack keywords. For example, if there exists a rule for the word "ATTACK", the middlebox will learn at which offset in the flow the word "ATTACK" appears (if it appears), but does not learn what the other parts of the traffic are. Traffic which does not match a suspicious keyword remains unreadable to the middlebox.

Probable Cause Privacy gives a different guarantee: that the middlebox will be able to *decrypt a flow* only if a substring of the flow is an exact match for a known attack keyword. Probable cause privacy is useful for IDS tasks which require regular expressions or scripting to complete their analysis. This model is inspired from two ideas. First, it is inspired from the notion of probable cause from United States' criminal law: one should give up privacy *only* if there is a reason for suspicion. Second, most rules in Snort that contain regular expressions first attempt to find a suspicious keyword in the packet – this keyword is selective so only a small fraction of the traffic matches this string and is passed through the regexp. Indeed, the Snort user manual [151] urges the presence of such selective keywords because otherwise, detection would be too slow. Since rules are structured this way, it becomes easier to implement our probable cause privacy model by decrypting the stream if there is a match to the suspicious keyword.

Exact match privacy provides security guarantees as in searchable encryption [147], which are well-studied. Probable cause privacy is a new privacy model, and we believe it may be useful in other network domains beyond middleboxes (e.g. network forensics or search warrants), although we leave such investigation to future work. We formalize and prove the security guarantees of BlindBox using standard indistinguishability-based definitions in our extended paper [144]. Both models are stronger than the "man in the middle" approach in deployment today, where all traffic is decrypted regardless of suspicion. A user who prefers exact match privacy over probable cause privacy can indicate so within BlindBox HTTPS.

5.1.3 System Architecture

We now return to Fig. 5.1 to explain each module and how BlindBox functions from a high level; we delve into the protocol and implementation details in the following sections.

Prior to any connection, RG generates a set of *rules* which contain a list of suspicious keywords known to formulate parts of attacks; RG signs these rules with its private key and shares them with MB, its customer. S and R, who trust RG, install a BlindBox HTTPS configuration which includes RG's public key. Beyond this initial setup, RG is never directly involved in the protocol. We now discuss the interactions between R, S, and MB when R and S open a connection in a network monitored by MB.

Connection setup: First, the sender and receiver run the regular SSL handshake which permits them to agree on a key k_0. The sender and receiver use k_0 to derive three keys (e.g., using a pseudorandom generator):

- k_{SSL}: the regular SSL key, used to encrypt the traffic as in the SSL protocol,
- k: used in our detection protocol, and
- k_{rand}: used as a seed for randomness. Since both endpoints have the same seed, they will generate the same randomness.

At the same time, MB performs its own connection setup to be able to perform detection over S and R's traffic. In an exchange with S and R, MB obtains each rule from RG deterministically encrypted with key k – this will later enable MB to perform the detection. However, this exchange occurs in such a way that MB *does not learn the value of* k and in such a way that R and S *do not learn what the rules are.* We call this exchange *obfuscated rule encryption* and we describe how it is implemented in the following section.

Unlike the above handshake between S and R, which bootstraps off the existing SSL handshake, obfuscated rule encryption is a new exchange. In existing deployments, clients typically do not communicate directly with DPI middleboxes (although for other kinds of middleboxes, such as explicit proxies [55] or NAT hole-punching [61], they may do so). Even though this step removes the complete "transparency" of the DPI appliance, it is an incremental change that we consider an acceptable tradeoff for the benefits of BlindBox.

Sending traffic: To transmit, the sender: (1) encrypts the traffic with SSL as in a non-BlindBox system; (2) tokenizes the traffic by splitting it in substrings taken from various offsets (as discussed in §5.2); and (3) encrypts the resulting tokens using our DPIEnc encryption scheme.

Detection: The middlebox receives the SSL-encrypted traffic and the encrypted tokens. The detect module will search for matchings between the encrypted rules and the encrypted tokens using BlindBox Detect (Sec. 5.2.2). If there is a match, one can choose the same actions as in a regular (unencrypted IDS) such as drop the packet, stop the connection, or notify an administrator. After completing detection, MB forwards the SSL traffic and the encrypted tokens to the sender.

Receiving traffic: Two actions happen at the receiver. First, the receiver decrypts and authenticates the traffic using regular SSL. Second, the receiver checks that the encrypted tokens were encrypted properly by the sender. Recall that, in our threat model, one endpoint may be malicious – this endpoint could try to cheat by not encrypting the tokens correctly or by encrypting only a subset of the tokens to eschew detection at the middlebox. Since we assume that at least one endpoint is honest, such verification will prevent this attack.

Because BlindBox only supports attack rules at the HTTP application layer, this check is sufficient to prevent evasion. Almost all the rules in our datasets were in this category. Nonetheless, it is worth noting that, if an IDS were to support rules that detected attacks on the client driver or NIC – before verification –, an attacker *could* evade detection by not tokenizing.

5.1.4 Protocols

BlindBox provides three protocols. In Protocol I, a rule consists of one keyword. MB must be able to detect if the keyword appears at any offset in the traffic based on equality match. This protocol suffices for document watermarking [146] and parental filtering [33] applications, but can support only a few IDS rules. In Protocol II, a rule consists of multiple keywords as well as position information of these keywords. This protocol supports a wider class of IDS rules than Protocol I, as we elaborate in §5.7. Protocol I and II provide Exact Match Privacy, as discussed in §5.1.2. Protocol III additionally supports regular expressions and scripts, thus enabling a full IDS. Protocol III provides Probable Cause Privacy, as discussed in §5.1.2.

5.2 Protocol I: Basic Detection

Protocol I enables matching a suspicious keyword against the encrypted traffic. An attack rule in this protocol consists of one keyword. Even though this protocol is the simplest of our protocols, it introduces the majority of our techniques. The other protocols extend Protocol I.

To detect a keyword match on encrypted text, one naturally considers searchable encryption [147, 103]. However, existing searchable encryption schemes do not fit our setting for two reasons. First, the setup of searchable encryption requires the entity who has the secret key to encrypt the rules; this implies, in our setting, that the endpoints see the rules (which is not allowed as discussed in §5.1.2). Our obfuscated rule encryption addresses this problem.

Second, none of the existing schemes meet both of our security and network performance requirements. There are at least two kinds of searchable encryption schemes: deterministic and randomized. Deterministic schemes [52] leak whether two words in the traffic are equal to each other (even if they do not match a rule). This provides weak privacy because it allows an attacker to perform frequency analysis. At the same time, these schemes are fast because they enable MB to build fast indexes that can process each token (e.g. word) in a packet in time logarithmic in the number of rules. On the other hand, randomized schemes [147, 103] provide stronger security guarantees because they prevent frequency analysis by salting ciphertexts. However, the usage of the salt in these schemes requires combining each token with each rule, resulting in a processing time linear in the number of rules for each token; as we show in §5.7, this is too slow for packet processing. In comparison, our encryption scheme DPIEnc and detection protocol BlindBox Detect achieve the best of both worlds: the detection speed of deterministic encryption and the security of randomized encryption.

Let us now describe how each BlindBox module in Fig. 5.1 works in turn. Recall that S and R are the sender and receiver, MB the middlebox and RG the rule generator.

Tokenization: The first step in the protocol is to tokenize the input traffic. We start with a basic tokenization scheme, which we refer to as "window-based" tokenization because it follows a simple sliding window algorithm. For every offset in the bytestream, the sender creates a token of a fixed length: we used 8 bytes per token in our implementation. For example, if the packet stream is "alice apple", the sender generates the tokens "alice ap", "lice app", "ice appl", and so on. Using this tokenization scheme, MB will be able to detect rule keywords of length 8 bytes or greater. For a keyword longer than 8 bytes, MB splits it in substrings of 8 bytes, some of which may overlap. For example, if a keyword is "maliciously", MB can search for "maliciou" and "iciously". Since each encrypted token is 5 bytes long and the endpoint generates one encrypted token per byte of traffic, the bandwidth overhead of this approach is of $5\times$.

We can reduce this bandwidth overhead by introducing some optimizations. First, for an HTTP-only IDS (which does not analyze arbitrary binaries), we can have senders ignore tokenization for images and videos which the IDS does not need to analyze. Second, we can tailor our tokenization further to the HTTP realm by observing how the keywords from attack rules for these protocols are structured. The keywords matched in rules start and end before or after a delimiter. Delimiters are punctuation, spacing, and special symbols. For example, for the payload "login.php?user=alice", possible keywords in rules are typically "login", "login.php", "?user=", "user=alice", but not "logi" or "logi.ph". Hence, the sender needs to generate only those tokens that could match keywords that start and end on delimiter-based offsets; this allows us to ignore redundant tokens in the window. We refer to this tokenization as "delimiter-based" tokenization. In §5.7, we compare the overheads and coverage of these two tokenization protocols.

5.2.1 The DPIEnc Encryption Scheme

In this subsection, we present our new DPIEnc encryption scheme, which is used by the Encrypt module in Fig. 5.1. The sender encrypts each token t obtained from the tokenization with our encryption scheme. The encryption of a token t in DPIEnc is:

$$\boxed{\mathsf{salt},\ \mathsf{AES}_{\mathsf{AES}_k(t)}(\mathsf{salt}) \bmod \mathsf{RS}, \qquad (5.1)}$$

where salt is a randomly-chosen value and RS is explained below.

Let us explain the rationale behind DPIEnc. For this purpose, assume that MB is being handed, for each rule r, the pair (r, $\mathsf{AES}_k(r)$), but not the key k. We explain in §5.2.3 how MB actually obtains $\mathsf{AES}_k(r)$.

Let's start by considering a simple deterministic encryption scheme instead of DPIEnc: the encryption of t is $\mathsf{AES}_k(t)$. Then, to check if t equals a keyword r, MB can simply check if $\mathsf{AES}_k(t) \overset{?}{=} \mathsf{AES}_k(r)$. Unfortunately, the resulting security is weak because every occurrence of t will have the same ciphertext. To address this problem, we need to randomize the encryption.

Hence, we use a "random function" H together with a random salt, and the ciphertext becomes: $\mathsf{salt}, H(\mathsf{salt}, \mathsf{AES}_k(t))$. Intuitively, H must be pseudorandom and not invertible. To perform a match, MB can then compute $H(\mathsf{salt}, \mathsf{AES}_k(r))$ based on $\mathsf{AES}_k(r)$ and salt, and again perform an equality check. The typical instantiation of H is SHA-1, but SHA-1 is not as fast as AES (because AES is implemented in hardware on modern processors) and can reduce BlindBox's network throughput. Instead, we implement H with AES, but this must be done carefully because these primitives have different security properties. To achieve the properties of H, AES must be keyed with a value that MB does not know when there is no match to an attack rule – hence, this value is $\mathsf{AES}_k(t)$. Our algorithm is now entirely implemented in AES, which makes it fast.

Finally, RS simply reduces the size of the ciphertext to reduce the bandwidth overhead, but it does not affect security. In our implementation, RS is 2^{40}, yielding a ciphertext length of 5 bytes. As a result, the ciphertext is no longer decryptable; this is not a problem because BlindBox always decrypts the traffic from the primary SSL stream.

Now, to detect a match between a keyword r and an encryption of a token t, MB computes $\mathsf{AES}_{\mathsf{AES}_k(r)}(\mathsf{salt}) \bmod \mathsf{RS}$ using salt and its knowledge of $\mathsf{AES}_k(r)$, and then tests for equality with $\mathsf{AES}_{\mathsf{AES}_k(t)}(\mathsf{salt}) \bmod \mathsf{RS}$.

Hence, naïvely, MB performs a match test for every token t and rule r, which results in a performance per token linear in the number of rules; this is too slow. To address this slowdown, our detection algorithm below makes this cost logarithmic in the number of rules, the same as for vanilla inspection of unencrypted traffic. This results in a significant performance improvement: for example, for a ruleset with 10000 keywords to match, a logarithmic lookup is four orders of magnitude faster than a linear scan.

5.2.2 BlindBox Detect Protocol

We now discuss how our detection algorithm achieves logarithmic lookup times, resolving the tension between security and performance. For simplicity of notation, denote $\mathsf{Enc}_k(\mathsf{salt}, t) = \mathsf{AES}_{\mathsf{AES}_k(t)}(\mathsf{salt})$.

The first idea is to precompute the values $\mathsf{Enc}_k(\mathsf{salt}, r)$ for every rule r and for every possible salt. Recall that MB can compute $\mathsf{Enc}_k(\mathsf{salt}, r)$ based only on salt and its knowledge of $\mathsf{AES}_k(r)$, and MB does not need to know k. Then, MB can arrange these values in a search tree. Next, for each encrypted token t in the traffic stream, MB simply looks up $\mathsf{Enc}_k(\mathsf{salt}, t)$ in the tree and checks if an equal value exists. However, the problem is that enumerating all possible salts for each keyword r is infeasible. Hence, it would be desirable to use only a few salts, but this strategy affects security: an attacker at MB can see which token in the traffic equals which other token in the traffic whenever the salt is reused for the same token. To maintain the desired security, every encryption of a token t must contain a different salt (although the salts can repeat across different tokens).

To use only a few salts and maintain security at the same time, the idea is for the sender to *generate salts based on the token value* and *no longer send the salt in the clear along with every encrypted token*. Concretely, the sender keeps a *counter table* mapping each token encrypted so far to how many times it appeared in the stream so far. Before sending encrypted tokens, the sender sends one initial salt, salt_0, and MB records it. Then, the sender no longer sends salts; concretely, for each token t, the sender sends $\mathsf{Enc}_k(\mathsf{salt}, t)$ but not salt. When encrypting a token t, the sender checks the number of times it was encrypted so far in the counter table, say ct_t, which could be zero. It then encrypts this token with the salt $(\mathsf{salt}_0 + \mathsf{ct}_t)$ by computing $\mathsf{Enc}_k(\mathsf{salt}_0 + \mathsf{ct}_t, t)$. Note that this provides the desired security because no two equal tokens will have the same salt.

For example, consider the sender needs to encrypt the tokens A, B, A. The sender computes and transmits: salt_0, $\mathsf{Enc}_k(\mathsf{salt}_0, A)$, $\mathsf{Enc}_k(\mathsf{salt}_0, B)$, and $\mathsf{Enc}_k(\mathsf{salt}_0 + 1, A)$. Not sending a salt for each ciphertext both reduces bandwidth and is required for security: if the sender had sent salts, MB could tell that the first and second tokens have the same salt, hence they are not equal.

To prevent the counter table from growing too large, the sender resets it every P bytes sent. When the sender resets this table, the sender sets $\mathsf{salt}_0 \leftarrow \mathsf{salt}_0 + \max_t \mathsf{ct}_t + 1$ and announces the new salt_0 to MB.

For detection, MB creates a table mapping each keyword r to a counter ct^*_r indicating the number of times this keyword r appeared so far in the traffic stream. MB also creates a search tree containing the encryption of each rule r with a salt computed from ct^*_r: $\mathsf{Enc}_k(\mathsf{salt}_0 + \mathsf{ct}^*_r, r)$. Whenever there is a match to r, MB increments ct^*_r, computes and inserts the new encryption $\mathsf{Enc}_k(\mathsf{salt}_0 + \mathsf{ct}^*_r, r)$ into the tree, and deletes the old value. We now summarize the detection algorithm.

BlindBox Detect: The state at MB consists of the counters ct^*_r for each rule r and a fast search tree made of $\mathsf{Enc}_k(\mathsf{salt}_0 + \mathsf{ct}^*_r, r)$ for each rule r.

```
1: For each encrypted token $\mathsf{Enc}_k(\mathsf{salt}, t)$ in a packet:
   1.1: If $\mathsf{Enc}_k(\mathsf{salt}, t)$ is in the search tree:
        1.1.1: There is a match, so take the corresponding action for this match.
        1.1.2: Delete the node in tree corresponding to $r$ and insert $\mathsf{Enc}_k(\mathsf{salt}_0 + \mathsf{ct}^*_r + 1, t)$
        1.1.3: Set $\mathsf{ct}^*_r \leftarrow \mathsf{ct}^*_r + 1$
```

With this strategy, for every token t, MB performs a simple tree lookup, which is logarithmic in the number of rules. Other tree operations, such as deletion and insertion, happen rarely: when a malicious keyword matches in the traffic. These operations are also logarithmic in the number of rules.

5.2.3 Rule Preparation

The detection protocol above assumes that MB obtains $\mathsf{AES}_k(r)$ for every keyword r, every time a new connection (having a new key k) is setup. But how can MB obtain these values? The challenge here is that no party, MB or S/R, seems fit to compute $\mathsf{AES}_k(r)$: MB knows r, but it is not allowed to learn k; S and R know k, but are not allowed to learn the rule r (as discussed in §5.1.2).

Intuition: We provide a technique, called *obfuscated rule encryption*, to address this problem. The idea is that the sender provides to the middlebox an "obfuscation" of the function AES with the key k hardcoded in it. This obfuscation hides the key k. The middlebox runs this obfuscation on the rule r and obtains $\mathsf{AES}_k(r)$, without learning k. We denote this obfuscated function by ObfAES_k.

Since practical obfuscation does not exist, we implement it with Yao garbled circuits [164, 113], on which we elaborate below. With garbled circuits, MB cannot directly plug in r as input to $\mathsf{ObfAES}_k()$; instead, it must obtain from the endpoints an encoding of r that works with ObfAES_k. For this task, the sender uses a protocol called oblivious transfer [117, 49], which does not reveal r to the endpoints. Moreover, MB needs to obtain a fresh, re-encrypted garbled circuit $\mathsf{ObfAES}_k()$ for every keyword r; the reason is that the security of garbled circuits does not hold if MB receives more than one encoding for the same garbled circuit.

A problem is that MB might attempt to run the obfuscated encryption function on rules of its choice, as opposed to rules from RG. To prevent this attack, rules from RG must be signed by RG and the obfuscated (garbled) function must check that there is a valid signature on the input rule before encrypting it. If the signature is not valid, it outputs null.

Let us now present the building blocks and our protocol in more detail.

Yao garbling scheme [164, 113]. At a high level, a garbled circuit scheme, first introduced by Yao, consists of two algorithms Garble and Eval. Garble takes as input a function F with n bits of input and outputs a garbled function ObfF and n pairs of labels $(L_1^0, L_1^1), \ldots, (L_n^0, L_n^1)$, one pair for every input bit of F. Consider any input x of n bits with x_i being its i-th bit. ObfF has the property that $\mathsf{ObfF}(L_1^{x_1}, \ldots, L_n^{x_n}) = F(x)$. Basically, ObfF produces the same output as F if given the labels corresponding to each bit of x. Regarding security, ObfF and $L_1^{x_1}, \ldots, L_n^{x_n}$ do not leak anything about F and x beyond $F(x)$, as long as an adversary receives labels for only one input x.

1-out-of-2 oblivious transfer (OT) [117, 49]. Consider that a party A has two values, L^0 and L^1, and party B has a bit b. Consider that B wants to obtain the b-th label from A, L^b, but B does not want to tell b to A. Also, A does not want B to learn the other label L^{1-b}. Hence, B cannot send b to A and A cannot send both labels to B. Oblivious transfer (OT) enables exactly this: B can obtain L^b without learning L^{1-b} and A does not learn b.

Rule preparation: Fig. 5.2 illustrates the rule preparation process for one keyword r. One endpoint could be malicious and attempt to perform garbling incorrectly to eschew detection. To prevent such an attack, both endpoints have to prepare the garbled circuit and send it to MB to check that they produced the same result. If the garbled circuits and labels match,

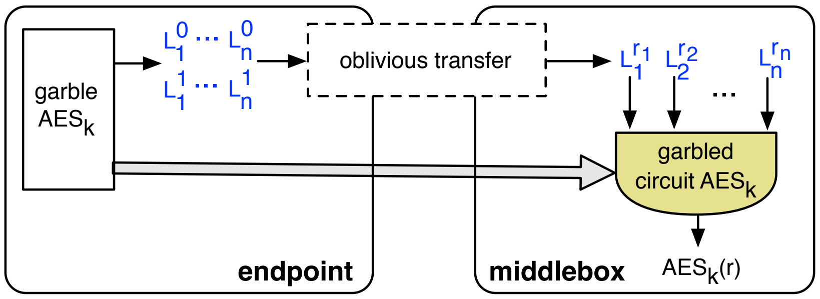


Figure 5.2: Rule preparation. The endpoint has a key k and the middlebox has a keyword r.

MB is assured that they are correct because at least one endpoint is honest (as discussed in Sec. 5.1.2). To enable this check, the endpoints must use the same randomness obtained from a pseudorandom generator seeded with k_{rand} (discussed in Sec. 5.1.3).

Rule preparation:

1: MB tells S and R the number of rules N it has.

2: For each rule $1, \ldots, N$, do:

2.1: S and R: Garble the following function F.

F on input $[x, \mathsf{sig}(x)]$ checks if $\mathsf{sig}(x)$ is a valid signature on x using RG's public key. If yes, it encrypts x with AES_k and outputs $\mathsf{AES}_k(x)$; else, it outputs $\perp$.

In the garbling process, use randomness based on k_{rand}. Send the resulting garbled circuit and labels to MB.

2.2: MB: Verify that the garbled circuits from S and R are the same, and let ObfAES_k be this garbled circuit. Let r and $\mathsf{sig}(r)$ be the current rule and its signature. Run oblivious transfer with each of S and R to obtain the labels for r and $\mathsf{sig}(r)$. Verify that the labels from S and R are the same, and denote them $L_1^{r_1}, \ldots, L_n^{r_n}$.

2.3: MB: Evaluate ObfAES_k on the labels $L_1^{r_1}, \ldots, L_n^{r_n}$ to obtain $\mathsf{AES}_k(r)$.

Rule preparation is the main performance overhead of BlindBox HTTPS. This overhead comes from the oblivious transfer and from the generation, transmission and evaluation of the garbled circuit, all of which are executed once for every rule. We evaluate this overhead in §5.7.

We additionally use a performance optimization that, instead of garbling the verification of sig, it garbles a hash computation while achieving the same security level.

5.2.4 Validate Tokens

As shown in Fig. 5.1, the validate tokens procedure runs at the receiver. This procedure takes the decrypted traffic from SSL and runs the same tokenize and encrypt modules as the sender executes on the traffic. The result is a set of encrypted tokens and it checks that these are the same as the encrypted tokens forwarded by MB. If not, there is a chance that the other endpoint is malicious and flags the misbehavior.

5.2.5 Security Guarantees

We proved our protocol secure with respect to our exact match privacy model; the proofs can be found in our extended paper [144]. We formalized the property that DPIEnc hides the traffic content from MB using an indistinguishability-based security definition. Informally, MB is given encryptions of a sequence of tokens $t'_1, \ldots, t'_n$ and keywords $r_1, \ldots, r_m$. Then, MB can choose two tokens t_0 and t_1 which do not match any of the keywords. Next, MB is given a ciphertext $c = \mathsf{Enc}_k(\mathsf{salt}, t_b)$ for some bit b and salt generated according to the BlindBox Detect protocol. The security property says that no polynomial-time attacker at MB can guess the value of b with chance better than half. In other words, MB cannot tell if t_0 or t_1 is encrypted in c. We can see why this property holds intuitively: if MB does not have $\mathsf{AES}_k(t_b)$, this value is indistinguishable from a random value by the pseudorandom permutation property of AES. Hence, $\mathsf{Enc}_k(\cdot, t_b)$ maps each salt to a random value, and there are no repetitions among these random values due to the choice of salt in BlindBox Detect. Thus, the distributions of ciphertexts for each value of b are essentially the same, and thus indistinguishable.

As part of our privacy model, BlindBox reveals a small amount of information to make detection faster: BlindBox does not hide the number of tokens in a packet. Also, if a suspicious keyword matches at an offset in the traffic stream, MB learns this offset. Hence, BlindBox necessarily weakens the privacy guarantees of SSL to allow efficient detection. (Note that BlindBox preserves the authenticity property of SSL.)

5.3 Protocol II: Limited IDS

This protocol supports a limited form of an IDS. Namely, it allows a rule to contain: (1) multiple keywords to be matched in the traffic, and (2) absolute and relative offset information within the packet. In our industrial dataset, the average *rule* contained three *keywords*; a rule is "matched" if all keywords are found within a flow.

This protocol supports most of the functionality in the rule language of Snort [151]. A few functional commands are not supported, the most notable being *pcre*, which allows arbitrary regular expressions to be run over the payload. This command is supported by Protocol III.

For example, consider rule number 2003296 from the Snort Emerging Threats ruleset:

```
alert tcp $EXTERNAL_NET $HTTP_PORTS
                                    -> $HOME_NET 1025:5000 (
flow: established,from_server;
content: "Server|3a| nginx/0.";
offset: 17; depth: 19;
content: "Content-Type|3a| text/html";
content: "|3a|80|3b|255.255.255.255"; )
```

This rule is triggered if the flow is from the server, it contains the keyword "Server|3a| nginx/0." at an offset between 17 and 19, and it also contains the keyword "Content-Type|3a| text/html" and "|3a|80|3b|255.255.255.255". The symbol "|" denotes binary data.

Protocol II builds on Protocol I in a straightforward way. The sender processes the stream the same as in Protocol I (including the encryption) with one exception: if the delimiter-based tokenization is used, the sender attaches to each encrypted token the offset in the stream where it appeared. In the window-based tokenization, the offset information need not be attached to each encrypted token because a token is generated at each offset and hence the offset can be deduced.

Detection happens similarly to before. For each encrypted token, MB checks if it appears in the rule tree. If so, it checks whether the offset of this encrypted token satisfies any range that might have been specified in the relevant rule. If all the fields of the relevant rule are satisfied, MB takes the action indicated by the rule.

Security Guarantee: The security guarantee is the same as in Protocol I: for each rule keyword, the middlebox learns if the keyword appears in the traffic and at what offset, but it learns nothing else about the parts of the traffic that do not match keywords. Note that the security guarantee is defined per keyword and not per rule: MB learns when a keyword matches even if the entire rule does not match.

5.4 Protocol III: Full IDS with Probable Cause Privacy

This section enables full IDS functionality, including regexp and scripts, based on our probable cause privacy model. If a keyword from a rule (a suspicious keyword) matches a stream of traffic, MB should be able to decrypt the traffic. This enables the middlebox to then run regexp (e.g., the "pcre" field in Snort) or scripts from Bro [122] on the decrypted data. However, if such a suspicious keyword does not match the packet stream, the middlebox cannot decrypt the traffic (due to cryptographic guarantees), and the security guarantee is the same as in Protocol II.

Protocol insight: The idea is to somehow embed the SSL key k_{SSL} into the encrypted tokens, such that, if MB has a rule keyword r that matches a token t in the traffic, MB should be able to compute k_{SSL}. To achieve this goal, we replace the encrypted token $\mathsf{Enc}_k(\mathsf{salt}, t)$ with $\mathsf{Enc}_k(\mathsf{salt}, t) \oplus k_{\mathsf{SSL}}$, where $\oplus$ is bitwise XOR. If $r = t$, MB has $\mathsf{AES}_k(t)$ and can construct $\mathsf{Enc}_k(\mathsf{salt}, t)$, and then obtain k_{SSL} through a XOR operation. The problem is that this slows down detection to a linear scan of the rules because the need to compute the XOR no longer allows a simple tree lookup of an encrypted token into the rule tree (described in Sec. 5.2.2).

Protocol: To maintain the efficiency of the detection, we retain the same encrypted token as in DPIEnc and use it for detection, but additionally create an encrypted token that has the key embedded in as above. Now, the encryption of a token t becomes a pair $[c_1 = \mathsf{Enc}_k(\mathsf{salt}, t)$, $c_2 = \mathsf{Enc}^*_k(\mathsf{salt}, t) \oplus k_{\mathsf{SSL}}]$, where $\mathsf{Enc}^*_k(\mathsf{salt}, t) = \mathsf{AES}_{\mathsf{AES}_k(t)}(\mathsf{salt} + 1)$ and the salt is generated as in BlindBox Detect (§5.2.2). Note that it is crucial that the salt in Enc^*_k differs from the salt in any c_1 encryption of t because otherwise an attacker can compute $c_1 \oplus c_2$ and obtain k_{SSL}. To enforce this requirement across different occurrences of the same token in BlindBox Detect, the sender now *increments the salt by two*: it uses an even salt for c_1 (and so does MB for the rules in the tree), while it uses an odd salt for c_2. MB uses c_1 to perform the detection as before. If MB detects a match with rule r using BlindBox Detect, MB computes $\mathsf{Enc}^*_k(\mathsf{salt}, r)$ using $\mathsf{AES}_k(r)$, and computes $\mathsf{Enc}^*_k(\mathsf{salt}, r) \oplus c_2$, which yields k_{SSL}. We prove the security of this protocol in our extended paper [144].

5.5 Discussion

In this section, we discuss adoption of BlindBox and privacy implications of the choice of rules and tokenization strategy in BlindBox.

5.5.1 Adoption and Deployment

ISP Adoption. In enterprises and private networks, BlindBox provides a good trade-off between the desires of users (who want privacy, and may want processing) and the network administrator (who wants to deploy processing primarily, and is willing to respect privacy if able to do so). Hence, deploying BlindBox is aligned with both parties' interests. However, in ISPs, sales of user data to marketing and analytics firms are a source of revenue – hence, an ISP has an incentive *not* to deploy BlindBox. Consequently, deployment in ISPs is likely to take place either under legislative requirement through privacy laws, or through a change in incentives. In outsourcing/cloud scenarios, where clients *pay directly for middlebox processing* we expect provider adoption of BlindBox's schemes to be more attractive, as it can attract more customers paying for the traffic inspection itself.

Client Adoption. BlindBox proposes a new end-to-end encryption protocol to replace HTTPS altogether. A truly ideal solution would require no changes at the endpoints – indeed, the success of middlebox deployments is partly due to the fact that middleboxes can be simply "dropped in" to the network. Unfortunately, existing HTTPS encryption algorithms use strong encryption schemes, which do not support *any* functional operations and cannot be used for our task; hence one must change HTTPS. Nonetheless, we believe that, in the long run, a change to HTTPS to allow inspection of encrypted traffic can be generic enough to support a wide array of middlebox applications, and not just the class of middleboxes in BlindBox. We believe these benefits will merit widespread "default" adoption in end host software suites.

Other Middleboxes. A set of other middleboxes do not fit into the DPI model adopted by

BlindBox, such as proxies, caches, compression engines, protocol accelerators, and transcoders [163, 143]. Hence, there remains work to resolve the tension between SSL/TLS and middleboxes in general. We believe that computation over encrypted data will remain a useful approach for these devices in general.

5.5.2 Generating Rules

The choice of rules affects privacy significantly. In the extreme, rules that match each letter ‘a’, ‘b’, . . . , ‘z’, result in no privacy at all. With BlindBox, rule designers must consider privacy implications of the rules they choose: ideally, a keyword should not match benign traffic at all.

Note that we do *not* provide guidelines on how to choose keywords or rules that are safe to use with our protocols; we contribute *only* the mechanism for implementing matching and probable cause decryption once the rules are chosen carefully. Choosing rules in a way that preserves matching and privacy requires careful thought. In fact, we emphasize that some existing rules are *not safe* to use with our protocols, and need to be changed. This is not surprising considering that existing rules were not written with privacy in mind. For example, in the Snort community rules, there are rules with keywords that match often such as ‘.exe’. This can cause frequent matching in Protocol II or frequent decryption in Protocol III.

An interesting future work question is to design a scheme that enables the middlebox to learn if a rule matches in an all-or-nothing way: that is, if a rule has more than one keyword, the middlebox should learn only if all strings match, and not if a subset of them match.

How to Tokenize Existing Rules Consider a set of rules. Define “effective keyword” to be a keyword that must be matched by BlindBox on the traffic. For Protocol II, every keyword of each rule is an effective keyword. For Protocol III, there is one effective keyword per rule as defined in §5.5.2. Since effective keywords have different lengths, the tokenization can happen in lengths of 2, 4, 8, 16, 32, 64, and 128. If an effective keyword is of size ℓ, a rule keyword is tokenized using the *largest* token size at most ℓ. For example, if the string is size 65, it is broken into two strings each of size 64 that overlap in 63 positions. One must *not* break the effective keywords in smaller tokens because this will leak more than necessary. Hence, for example, if there is only one effective keyword that is short, say of length 4, tokens of size 4 in the traffic will match only this effective keyword and not other rules.

Tokenization for Protocol III Given a rule for Protocol III, to decrease the frequency of decryption, the probable cause decryption must be triggered by a string in this rule that appears in the traffic as infrequently as possible (ideally, only when the traffic is suspicious). This string can be a keyword or a substring of a regular expression that is matched by equality. Probable cause decryption should be triggered by only one such string per rule (because a rule matches only when all such strings match). For example, for a rule with content =‘abc’ and content = ‘abcdefghij’, the trigger should be the second string. If the rule additionally contains pcre = ‘[1-9]abcdefghij123’, the trigger should be ‘abcdefghij123’.

5.6 System Implementation

We implemented two separate libraries for BlindBox: a client/server library for transmission called BlindBox HTTPS, and a Click-based [107] middlebox.

BlindBox library. The BlindBox HTTPS protocol is implemented in a C library. When a client opens a connection, our protocol actually opens three separate sockets: one over normal SSL, one to transmit the "searchable" encrypted tokens, and one to listen if a middlebox on path requests garbled circuits. The normal SSL channel runs on top of a modified GnuTLS [31] library which allows us to extract the session key under Protocol III. On send, the endpoint first sends the encrypted tokens, and then sends the traffic over normal HTTPS. If there is a middlebox on the path, the endpoints generate garbled circuits using JustGarble [53] in combination with the OT Extension library [20].

The middlebox. We implemented the middlebox in multithreaded Click [107] over DPDK [99]; in our implementation, half of the threads perform detection over the data stream ("detection" threads), and half perform obfuscated rule encryption exchanges with clients ("garble" threads). When a new connection opens, a detection thread signals to a garble thread and the garble thread opens an obfuscated rule encryption channel with the endpoints. Once the garble thread has evaluated all circuits received from the clients and obtained the encrypted rules, it constructs the search tree. The detection thread then runs the detection based on the search tree, and allows data packets in the SSL channel to proceed if no attack has been detected.

When a detection thread matches a rule, under Protocols I and II, the middlebox blocks the connection. Under Protocol III, it computes the decryption key (which is possible due to a match), and it forwards the encrypted traffic and the key to a decryption element. This element is implemented as a wrapper around the open-source ssldump [28] tool. The decrypted traffic can then be forwarded to any other system (Snort, Bro, *etc.*) for more complex processing. We modeled this after SSL termination devices [56], which today man-in-the-middle traffic before passing it on to some other monitoring or DPI device.

5.7 Evaluation

When evaluating BlindBox, we aimed to answer two questions. First, can BlindBox support the functionality of our target applications – data exfiltration (document watermarking), parental filtering, and HTTP intrusion detection? Second, what are the performance overheads of BlindBox at both the endpoints and the middlebox?

5.7.1 Functionality Evaluation

To evaluate the functionality supported by BlindBox, we answer a set of sub-questions.

Can BlindBox implement the functionality required for each target system? Table 5.1 shows what fraction of "rules" for different target applications rely solely on single-exact match (as in Protocol I), multiple exact-match strings (as in Protocol II), or regular expressions or scripts (as in Protocol III). We evaluate this using public datasets for document watermarking [146], parental filtering [33], and IDS rules (from the Snort community [136] and Emerging Threats [12]). In addition, we evaluate on two industrial datasets from Lastline and McAfee Stonesoft to which we had (partial) access.

Document watermarking and parental filtering can be completely supported using Protocol I because each system relies only on the detection of a single keyword to trigger an alarm. However, Protocol I can support only between 1.6-5% of the policies required by the more general HTTP IDS applications (the two public Snort datasets, as well as the datasets from McAfee Stonesoft and Lastline). This limitation is due to the fact that most IDS policies require detection of multiple keywords or regular expressions.

Protocol II, by supporting multiple exact match keywords, extends support to 29-67% of policies for the HTTP IDS applications. Protocol III supports all applications including regular expressions and scripting, by enabling decryption when there is a probable cause to do so.

What fraction of existing rules can be used with Protocol II and a given minimum token length? Protocol II allows a middlebox to search for multiple exact-match strings to detect an attack. A rule generator may choose to restrict the minimum size of transmitted tokens to avoid many false positive matches (trivially, the set of rules 'a', 'b'... 'z' would allow a middlebox to decrypt all text), requiring tokens of 4, 8, or 16 bytes. Figure 5.3 shows the number of rules from the Emerging Threats Snort ruleset such that all search strings in the rule are n characters long or more. This further reduces the number of rules that can be implemented with BlindBox 'as is.' A rule generator may be able to rewrite these rules such that they do not require searches for such short tokens, removing the short terms and potentially adding in additional search terms to avoid increasing false positives. We leave an exploration of such a mechanism to future work.

Does BlindBox fail to detect any attacks/policy violations that these standard implementations would detect? The answer depends on which tokenization technique one uses out of the two techniques we described in §5.2: window-based and delimiter-based tokenization. The window-based tokenization does not affect the detection accuracy of the rules because it creates a token at every offset. The delimiter-based tokenization relies on the assumption that, in IDSes, most rules occur on the boundary of non-alphanumeric characters, and thus does not transmit all possible tokens – only those required to detect rules which occur between such "delimiters". To test if this tokenization misses attacks, we ran BlindBox over the ICTF2010 [158] network trace, and used as rules the Snort Emerging Threats ruleset from which we removed the rules with regular expressions. The ICTF trace was generated during a college "capture the flag" contest during which students attempted to hack different servers to win the competition, so it contains a large number of attacks. We detected 97.1% of the attack keywords and 99% of the attack rules that would have been detected with Snort. (Recall that an attack rule may consist of multiple keywords.)

Dataset	I.	II.	III.
Document watermarking [146]	100%	100%	100%
Parental filtering [33]	100%	100%	100%
Snort Community (HTTP)	3%	67%	100%
Snort Emerging Threats (HTTP)	1.6%	42%	100%
McAfee Stonesoft IDS	5%	40%	100%
Lastline	0%	29.1%	100%

Table 5.1: Fraction of attack rules in public and industrial rule sets addressable with Protocols I, II, and III.

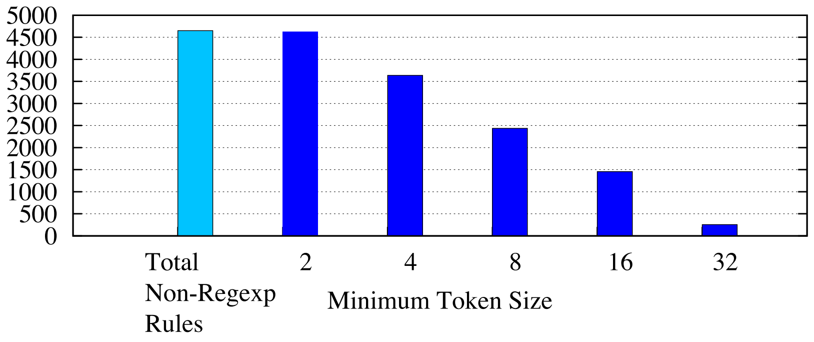


Figure 5.3: Number of rules in the Emerging Threats dataset which (a) do not require regular expressions, and (b) search for exact match strings of minimum lengths 2 - 32.

5.7.2 Performance Evaluation

We now investigate BlindBox's performance overheads at both the client and the network. For all experiments, the client software uses Protocol II, which has higher overhead than Protocol I. We do not evaluate Protocol III directly; the differences we would expect from Protocol III relative to II would include a secondary middlebox to perform regular expression processing, and an increase in bandwidth due to the key being embedded in each encrypted token.

Our prototype of the client software runs on two servers with 2.60 GHz processors connected by a 10GbE link. The machines are multicore, but we used only one thread per client. The CPU supports AES-NI instructions and thus the encryption times for both SSL and BlindBox reflect this hardware support. Since typical clients are not running in the same rack over a 10GbE links, in some experiments we reduced throughput to 20Mbps (typical of a broadband home link) and increased latency to 10ms RTT. Our prototype middlebox runs with four 2.6GHz Xeon E5-2650 cores and 128 GB RAM; the network hardware is a single 10GbE Intel 82599 compatible network card. All of our experiments were performed on this testbed. For microbenchmarks (as in Table 5.2), we measured time to complete a loop of 10,000 iterations and took an average. For flow completion benchmarks we took an average of five runs.

		HTTPS	**FE Strawman**	**Searchable Strawman**	**BlindBox**
Client	*Encrypt* (128 bits)	13ns	70ms	2.7μs	69ns
	Encrypt (1500 bytes)	3μs	15s	257μs	90μs
	Setup (1 Keyword)	73ms	N/A	N/A	588 ms
	Setup (3K Rules)	73ms	N/A	N/A	97 s
MB	*Detection:*				
	1 Rule, 1 Token	NP	170ms	1.9μs	20ns
	1 Rule, 1 Packet	NP	36s	52μs	5μs
	3K Rules, 1 Token	NP	8.3 minutes	5.6ms	137ns
	3K Rules, 1 Packet	NP	5.7 days	157ms	33μs

Table 5.2: Connection and detection micro-benchmarks comparing Vanilla HTTPS, the functional encryption (FE) strawman, the searchable strawman, and BlindBox HTTPS. NP stands for not possible. The average rule includes three keywords.

To summarize our performance results, BlindBox is practical for long-lived connections: the throughput of encryption and detection are comparable with rates of current (unencrypted) deployments. Additionally, BlindBox is 3 to 6 orders of magnitude faster than relevant implementations using existing cryptography; these solutions, by themselves, are incomplete in addition to being slow. The primary overhead of BlindBox is setting up a connection, due to the obfuscated rule encryption. This cost is small for small rulesets, but can take as long as 1.5 minutes for rulesets with thousands of rules; hence, BlindBox is not yet practical for systems with thousands of rules and short-lived connections that need to run setup frequently. We now elaborate on all these points.

Strawmen

BlindBox is the only system we know of to enable DPI over encrypted data. Nevertheless, to understand its performance, we compare it to standard SSL as well as two strawmen, which we now describe.

A searchable encryption scheme due to Song et al. [147]: This scheme does not enable obfuscated rule encryption or probable cause decryption, but can implement encryption and detection as in Protocols I and II (but not Protocol III). We used the implementation of Song et al. from [126], but replaced the use of SHA512 with the AES-NI instruction in a secure way, to speed up this scheme.

Generic functional encryption (FE) [87, 93]: Such schemes, if enhanced with our obfuscated rule encryption technique, can in theory perform Protocols I, II, and III. However, such encryption schemes are prohibitively expensive to be run and evaluated. For example, one such scheme [93] nests fully homomorphic encryption twice, resulting in an overhead of at least 10 orders of magnitude. Instead, we chose and implemented a simple and specialized functional encryption scheme due to Katz et al. [104]. The performance of this scheme is a generous lower bound on the performance of the generic protocols (the Katz et al. scheme does not support Protocol III because it can compute only inner product).

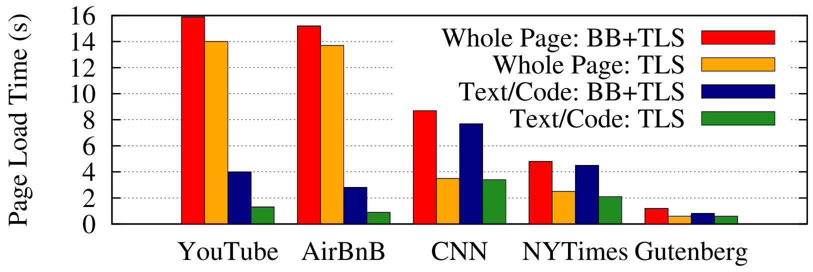


Figure 5.4: Download time for TLS and BlindBox (BB) + TLS at 20Mbps×10ms.

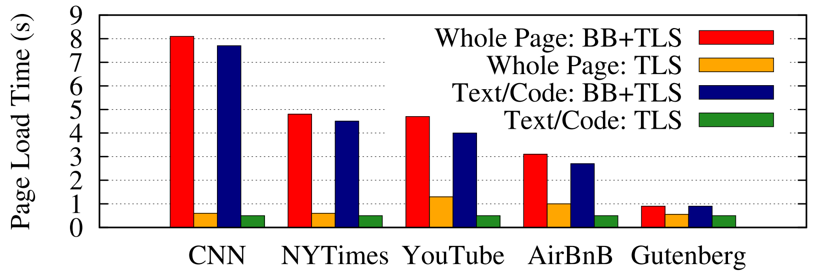


Figure 5.5: Download time for TLS and BlindBox (BB) + TLS at 1Gbps×10ms.

Client Performance

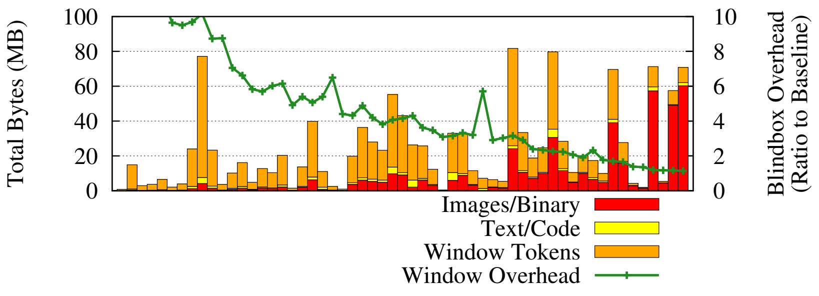
How long does it take to encrypt a token? Table 5.2 provides micro-benchmarks for encryption, detection, and setup using BlindBox, HTTPS, and our strawmen. With HTTPS (using GnuTLS), encryption of one 128-bit block took on average 13ns, and 3μs per 1400 byte packet. BlindBox increases these values to $69ns$ and 90μs respectively. These figures include the time to perform HTTPS transmission in the primary channel, as well as the overheads from BlindBox: the tokenization process itself (deciding which substrings to tokenize) as well as the encryption process (encrypting and then hashing each token with AES). The searchable strawman performs encryption of a single token on average 2.7μs and 257μs for an entire packet; the primary overhead relative to BlindBox here is multiple calls to `/dev/urandom` because the scheme requires random salts for every token. With fixed or pre-chosen salts, we would expect the searchable strawman to have comparable encryption times to BlindBox. As we discuss, the detection times for this strawman are slower. The FE strawman takes six orders of magnitude longer than BlindBox and is even further impractical: a client using this scheme could transmit at most one packet every 15 seconds.

How long does the initial handshake take with the middlebox? The initial handshake to perform obfuscated rule encryption runs in time proportional to the number of rules. In the datasets we worked with, the average Protocol II rule had slightly more than 3 keywords; a typical 3000 rule IDS rule set contains between 9-10k keywords. The total client-side time required for 10k keywords was 97 seconds; for 1000 keywords, setup time was 9.5s. In a smaller ruleset of 10 or 100 keywords (which is typical in a watermark detection exfiltration device), setup ran in 650ms and 1.6 seconds, respectively. These values are dependent on the clock speed of the CPU (to generate the garbled circuits) and the network bandwidth and latency (to transmit the circuits from client to sender). Our servers have 2.6GHz cores; we assumed a middlebox on a local area network near the client with a 100μs RTT between the two and a 1Gbps connection. Garbling a circuit took 1042μs per circuit; each garbled circuit transmission is 599KB.

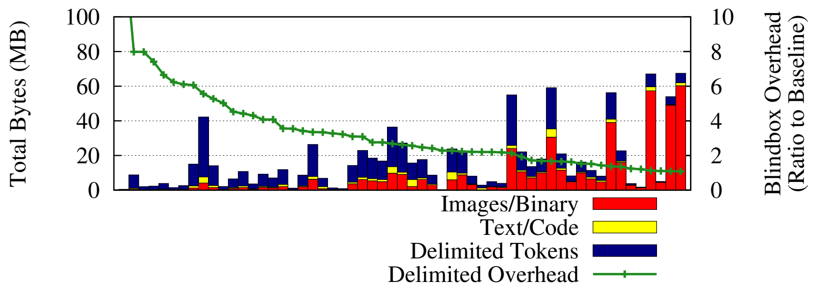
Neither strawman has an appropriate setup phase that meets the requirement of not making the rules visible to the endpoints. However, one can extend these strawmen with BlindBox's obfuscated rule encryption technique, and encrypt the rules using garbled circuits. In this case, for the scheme of Song et al., the setup cost would be similar to the one of BlindBox because their scheme also encrypts the rule keywords with AES. For the scheme of Katz et al., the setup would be much slower because one needs garbled circuits for modular exponentiation, which are huge. Based on the size of such circuits reported in the literature [53], we can compute a generous lower bound on the size of the garbled circuits and on the setup cost for this strawman: it is at least $1.8 \cdot 10^3$ times larger/slower than the setup in BlindBox.

How long are page downloads with BlindBox, excluding the setup (handshake) cost? Figure 5.4 shows page download times using our "typical end user" testbed with 20Mbps links. In this figure, we show five popular websites: YouTube, AirBnB, CNN, The New York Times, and Project Gutenberg. The data shown represents the post-handshake (persistent connection) page download time, with tokenization on 8-byte boundaries. YouTube and AirBnB load video, and hence have a large amount of binary data which is not tokenized. CNN and The New York Times have a mixture of data, and Project Gutenberg is almost entirely text. We show results for both the amount of time to download the page including all video and image content, as well as the amount of time to load only the Text/Code of the page. The overheads when downloading the whole page are at most $2\times$; for pages with large amount of binary data like YouTube and AirBnB, the overhead was only 10-13%. Load times for Text/Code only – which are required to actually begin rendering the page for the user – are impacted more strongly, with penalties as high as $3\times$ and a worst case of about $2\times$.

What is the computational overhead of BlindBox encryption, and how does this overhead impact page load times? While the encryption costs are not noticeable in the page download times observed over the "typical client" network configuration, we immediately see the cost of encryption overhead when the available link capacity increases to 1Gbps in Figure 5.5 – at this point, we see a performance overhead of as much as $16\times$ relative to the baseline SSL download time. For both runs (tr15/figs. 5.4 and 5.5), we observed that the CPU was almost continuously fully utilized to transfer data during data transmission. At 20Mbps, the encryption cost is not noticeable as the CPU can continue producing data at around the link rate; at 1Gbps, transmission with BlindBox stalls relative to SSL, as the BlindBox sender cannot encrypt fast enough to keep up with the line rate. This result is unsurprising given the results in Table 5.2, showing that BlindBox takes $30\times$ longer to encrypt a packet than standard HTTPS. This overhead can be mitigated with extra cores; while we ran with only



(a) Window-Based Tokenization



(b) Delimiter-Based Tokenization

Figure 5.6: Bandwidth overhead over top-50 web dataset.

one core per connection, tokenization can easily be parallelized.

What is the bandwidth overhead of transmitting encrypted tokens for a typical web page? Minimizing bandwidth overhead is key to client performance: less data transmitted means less cost, faster transfer times, and faster detection times. The bandwidth overhead in BlindBox depends on the number of tokens produced. The number of encrypted tokens varies widely depending on three parameters of the page being loaded: what fraction of bytes are text/code which must be tokenized, how "dense" the text/code is in number of delimiters, and whether or not the web server and client support compression.

Figures 5.6 (a) and (b) break down transmitted data into the number of text-bytes, binary-bytes, and tokenize-bytes using the window-based and delimiter-based tokenization algorithms (as discussed in §5.2); the right hand axis shows the overhead of adding tokens over transmitting just the original page data. We measured this by downloading the Alexa top-50 websites [2] and running BlindBox over all page content (including secondary resources loaded through AJAX, callbacks, etc.) The median page with delimited tokens sees a 2.5× increase in the number of bytes transmitted. In the best case, some pages see only a 1.1× increase, and in the worst case, a page sees a 14× overhead. The median page with window tokens sees a 4× increase in the number of bytes transmitted; the worst page sees a 24× overhead. The first observable factor affecting this overhead, as seen in these figures, is simply what fraction of bytes in the original page load required tokenization. Pages consisting mostly of video suffered lower penalties than pages with large amounts of text, HTML, and Javascript because we do not tokenize video.

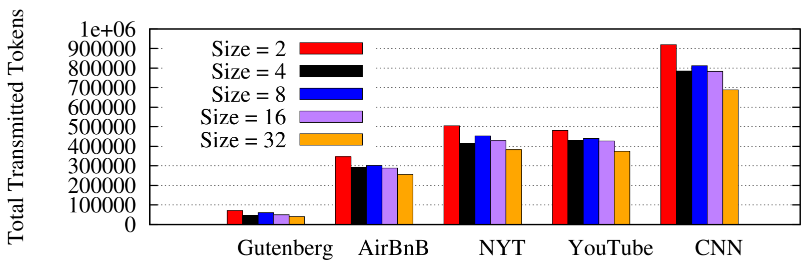


Figure 5.7: Tokens generated for each of six popular websites using delimiter-based tokenization and a minimum token size between 1-32 bytes.

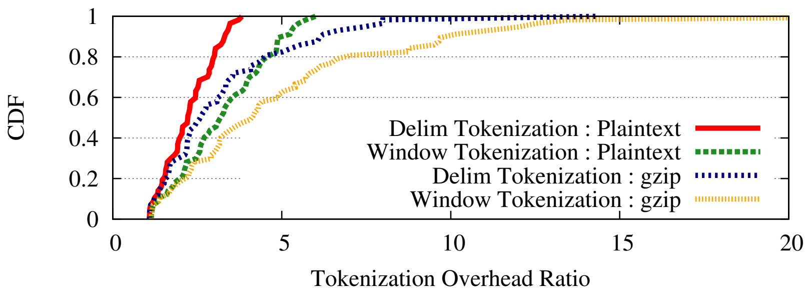
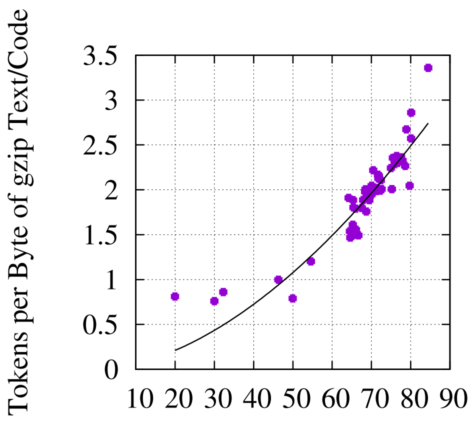
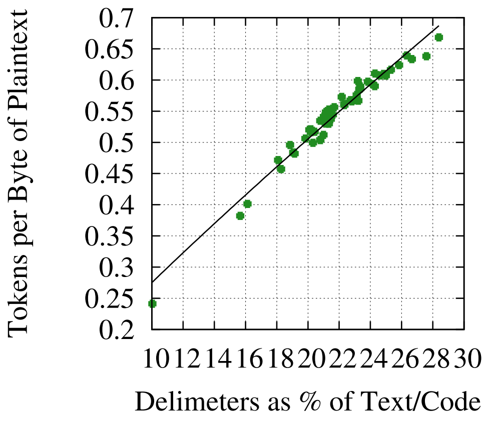


Figure 5.8: Ratio: transmitted bytes with BlindBox to transmitted bytes with SSL.

A second factor, better observed in Figures 5.8 and 5.9(a) is whether or not the web server hosting the page supports gzip compression. Many web servers will compress content before sending it to clients, which then unzip the data before passing to rendering in the browser. Where window based tokenization imposes a penalty of one token (five bytes) per plaintext byte (and delimiter-based tokenization imposes less than half of a token – 2.2 bytes – by eliminating tokens which are redundant to the DPI engine), compressing the plaintext makes the perceived penalty higher: the baseline data can be compressed, but encrypted tokens cannot. In Figure 5.8 we show a CDF of the ratio of BlindBox bytes to SSL bytes when gzip is not enabled, and when gzip is enabled exactly as in the original trace (i.e. we compare against the bytes gzipped when we downloaded the dataset from the webservers; if any data was not compressed we left it as-is and did not try to compress it further). When compared against plaintext, both window and delimiter based tokenization have "tight" tails – the worst page with window based tokenization has slightly more than $5\times$ overhead, and the worst page with delimiter tokenization has around $4\times$ overhead. But, for pages which benefit strongly from compression, the penalty can begin to look dramatic at the tail, going as high as $24\times$ for one page (Craigslist.com, which is mostly text/code and benefits strongly from compression). Figure 5.9(a) shows for each page the number of tokens produced on average per byte, plotted against the page reduction achieved by the web server by using gzip.



(a) Gzip



(b) Delimiter Density

Figure 5.9: Impact of compression and delimiter density on tokenization overhead for delimiter-based tokenization only.

The final factor is simply the number of delimiters seen in a page – text-only pages like Project Gutenberg do well in this metric, since there are few code-like characters in the text. The worst performers in this area are pages which make large use of compressed javascript code, where a large fraction of characters result in tokenization. Figure 5.9(b) illustrates this effect for the same dataset as previously.

Middlebox Performance

We investigate performance at the middlebox using both micro-benchmarks and overall throughput.

What throughput can BlindBox sustain and how does this compare to standard IDS? When running our BlindBox implementation over synthetic traffic, we measured a throughput of 166Mbps; when running Snort over the same traffic, we measured a throughput of 85Mbps. Hence, BlindBox performed detection twice as fast as Snort, which inspects *unencrypted* traffic. The reason behind this situation is twofold. First, BlindBox reduces all detection to exact matching, pushing all regular expression parsing to a secondary middlebox, invoked rarely. Second, our implementation is built over DPDK-click, a faster packet-capture library than what Snort uses by default. Hence, it is unsurprising that BlindBox performs detection more quickly. Nevertheless, the point of this experiment is not to show that BlindBox is faster than Snort, but instead to demonstrate that BlindBox provides competitive performance to today's deployments.

How does BlindBox compare in detection time against other strawmen approaches? While we did not implement a version of BlindBox which relied on our strawmen, we can compare against it using a smaller benchmark. Once again, in Table 5.2, the FE strawman is seen to be prohibitively impractical: detection over a single packet against a 3000 ruleset takes more than a day.

The searchable strawman is also prohibitively slow: it performs detection over a 1500 byte packet in 157 ms, which is equivalent to no more than 6-7 packets per second. This performance is three orders of magnitude slower than the performance of BlindBox's middlebox. This overhead results from the fact that the searchable strawman must perform an encryption operation over every keyword to perform a comparison against a client token, resulting in a task linear in the number of keywords. In contrast, BlindBox's DPIEnc scheme encrypts the data in such a way that the middlebox can use a fast, pre-computed search tree (which gives a logarithmic search) to match encrypted tokens to rules.

5.8 Related work

Related work falls into two categories: insecure proposals, and work on computing on encrypted data.

5.8.1 Insecure Proposals

Some existing systems mount a man-in-the-middle attack on SSL [100, 98] by installing fake certificates at the middlebox [105, 137]. This enables the middlebox to break the security of SSL and decrypt the traffic so it can run DPI. This breaks the end-to-end security of SSL, and results in a host of issues, as surveyed by Jarmoc [100].

Some proposals allow users to tunnel their traffic to a third party middlebox provider, e.g. Meddle [134], Beyond the Radio [155], and APLOMB [143]. These approaches allow the middlebox owner to inspect/read all traffic. The situation is preferable to the status queue (from the client's perspective) in that the inspector is one with whom the client has a formal/contractual relationship – but, unlike BlindBox, the client still must grant *someone* access to the plaintext traffic. Further, this approach is not preferable to service providers, who may wish to enforce policy on users in the network, e.g., that no hosts within the network are infected with botnet malware.

5.8.2 Computing on Encrypted Data

Fully homomorphic encryption (FHE) [90] and general functional encryption [87, 93] are encryption schemes that can compute any function over encrypted data; hence, they can in principle support the complexity of deep packet inspection tasks. However, they do not address all the desired security properties in our threat model, and more importantly, they are prohibitively slow, currently at least 8 orders of magnitude slower than unencrypted computation [91].

Some recent systems such as CryptDB [126] and Mylar [127] showed how to support some specialized computation efficiently on encrypted data. However, these systems perform different tasks than is needed for middleboxes and do not meet our threat model.

There has been a large amount of work on searchable encryption [147, 103, 57, 52]. No searchable encryption scheme provides a strategy for encrypting the rules securely and for supporting arbitrary regexps, both of which BlindBox provides. Moreover, existing schemes cannot provide the performance required for packet processing. For example, BlindBox is three orders of magnitude faster than a system using the symmetric-key searchable scheme of Song et al. [147]. Public-key searchable encryption schemes, such as [57], are even slower because they perform a cryptographic pairing (which takes hundreds of microseconds per pairing), for every pair of token to rule content (a linear, rather than logarithmic task in the number of rules).

5.9 Conclusion

In this chapter, we presented BlindBox, a system that resolves the tension between security and DPI middlebox functionality in networks. To the best of our knowledge, BlindBox is the first system to enable Deep Packet Inspection over encrypted traffic without requiring decryption of the underlying traffic. BlindBox supports real DPI applications such as IDS, exfiltration detection, and parental filtering. BlindBox performs best over long-running, persistent connections using SPDY-like or tunneled protocols. Using BlindBox Detect, a middlebox running BlindBox can perform detection on a single core at 186Mbps – competitive with many deployed IDS implementations.

We envisage that BlindBox is the first step towards a *general protocol* to resolve the tension between encryption and all categories of middleboxes. BlindBox currently supports middleboxes for DPI filtering only, however, we believe that the general blueprint BlindBox provides – computation over encrypted traffic – can be extended to implement other middlebox capabilities, including caches, protocol accelerators, compression engines.

Chapter 6

Conclusion, Lessons Learned & Thoughts for the Future

In this thesis, we have argued that by following the blueprint of outsourcing and cloud computing, middleboxes can be made easier to manage, more cost-effective, and more efficient.

With **APLOMB** (Chapter 3) we designed, implemented, and evaluated a system that shows the feasibility of the outsourcing architecture overall. APLOMB allows an enterprise to remove almost all of its middlebox infrastructure, offloading the processing to a third party provider instead. APLOMB imposes only modest performance overheads while improving manageability (by pushing many difficult management tasks from enterprise administrators to experts at the cloud provider) and cost (by allowing infrastructure to 'scale on demand' and administrators pay for usage).

With **FTMB** (Chapter 4) we discussed how moving middleboxes to the cloud dovetails with software middlebox implementations. We showed how to take advantage of elastic resources available in cloud environments to automatically fail over to a backup when a middlebox fails. FTMB avoids the common pitfall in today's middlebox deployments of custom, per-device solutions, instead designing for arbitrary packet processors. FTMB provides reliability guarantees with performance overheads on the order of tens of μs – orders of magnitude better than competing designs.

Finally, with **BlindBox** (Chapter 5) we confronted privacy as an obstacle to outsourcing middlebox infrastructure. Where classic 'Deep Packet Inspection' devices must decrypt traffic to operate over it, BlindBox uses functional cryptography to allow middleboxes to operate over data while it remains encrypted, learning only what is necessary to detect attacks in the plaintext. BlindBox shows that outsourcing middleboxes need not come with heavy sacrifices to user privacy.

Before concluding, we turn to the present and future of middleboxes and the cloud blueprint, discussing the Network Functions Virtualization movement (NFV) and lessons learned from this research for middlebox deployments going forward.

6.1 The Rise of Network Functions Virtualization

In 2012, the European Telecommunications Standards Institute (ETSI) issued a proposal they called Network Functions Virtualization (NFV) [82]. NFV aims to move middlebox packet processing from dedicated, special purpose hardware on to general-purpose infrastructure using software and virtualization – just as we propose that cloud providers do in APLOMB and this thesis.

NFV grew out of ISPs desire to improve the manageability and efficiency of their own infrastructure, which was composed of fixed-function, vendor-specific hardware middlebox implementations. Since then, NFV has expanded to include enterprises and datacenters also re-deploying their own, internal infrastructure with software middlebox implementations. Further, there is some early interest among some ISPs to extend the benefits of NFV with outsourcing opportunities for clients to offload processing to their ISPs [50]. Hence, despite different starting motivations, the goals of NFV and this thesis strongly overlap.

Along these lines, several projects and systems designed in the context of NFV give solutions to open challenges in the cloud computing blueprint. For example, the industrial and research designs in the NFV space have proposed schedulers/orchestraters for automatic instantiation of middleboxes, optimizing middlebox placement, instantiating new middlebox instances as demand scales up, and monitoring availability and health of running middlebox instances [119, 10, 18]. Other projects have looked at scaling and shared data abstractions for scaled out middleboxes [133, 89, 161]. The IETF's Service Function Chaining working group is actively investigating how to best implement routing through multi-middlebox topologies and enforce policies about which traffic receives processing by which middleboxes [130].

At present, the space is actively growing with over 270 members in the ETSI NFV working group [81]. It is likely that the future of the cloud computing blueprint for middlebox processing rests in the success of NFV.

6.2 Lessons Learned and Thoughts for the Future

We now discuss a few broad lessons learned over the course of this research, and what they suggest about future middlebox deployments.

Processing data at packet-sized scales magnifies the impact of even small overheads and hence requires new algorithms and system designs.

Implementing the APLOMB redirection infrastructure (as discussed in Chapter 3), we the authors found ourselves surprised at how well our prototype of redirection infrastructure performed. However, implementing the software for middlebox infrastructure that would run within the cloud was quite the opposite experience. We had (perhaps naively) assumed that we would be able to take advantage of existing algorithms and systems for scaling, fault-tolerance, scheduling, *etc.* 'out of the box.' The failure of existing fault-tolerance approaches to cope with the performance constraints of packet processing led to the development of FTMB. As we discussed in Chapter 4, the overheads assumed to be reasonable by traditional cloud services like web servers or big data analytics frameworks include millisecond latency overheads and increases in processing time, which amount to drastic performance penalties when packet processing tasks release packets ever microsecond or so. Hence this thesis refers to the 'blueprint' of cloud computing – that the cloud can provide resources for failover and scaling, that utility computing can improve management, extensibility and portability of middlebox software – but not that exactly the same mechanisms and implementations to do so will achieve success.

For this reason we are skeptical of NFV architectures which rely in the packet-processing dataplane on existing software from the cloud domain. We have already seen this point play out, *e.g.*, when it comes to virtualization. Early NFV proposals used standard virtualization network interfaces (indeed, we did in implementing APLOMB), but these interfaces could not sustain multi-gigabit line rates required by the largest middlebox deployments. Only later did specialized proposals like ClickOS' [115] netmap-based [135] Xen network interface, or NetBricks' [121] Zero-Copy Software Isolation (ZCSI) enable the classic cloud benefit of multitenant isolation with acceptable overheads for packet processing workloads.

Middlebox processing is not always embarassingly parallel.

We observed throughout the work in this thesis that middlebox processing maintains complex state on the dataplane. As we discussed in Chapter 4, state that is shared between many cores or machines inhibits parallelism. At the same time, network bandwidth demands are increasing – in 2014, the average user consumed 18.5 GB of data per month, while in 2013 this figure was just 2.9GB [118]. With the end of Moore's law, scaling effectively requires the ability to parallelize. This pushes middlebox architects directly in conflict with Amdahl's law, as the growth of complex state in middleboxes and growth in demand for network throughput are inherently at odds with each other. Consequently, middlebox architects will be forced to either cut back on shared and aggregate data, or develop ways to partition and distribute data more efficiently between parallel processors.

Middlebox tussles can sometimes be converted to multiparty computation challenges.

Middleboxes have often been cited as an example of a network 'tussle space', where 'players who make up the Internet millieu' have 'interests directly at odds with each other' [64]. For example, firewalls represent the interests of network administrators who wish to restrict what protocols and types of traffic can be sent on their network, and their use is at odds with the interests of users who want to send banned traffic. In Chapter 5, we describe another tussle in which admins seek to decrypt data and inspect it for attacks, which is at odds with user's desire for privacy. As the number of players in the space increases, the number of competing interests rise as well. APLOMB proposes adding cloud administrators to the picture. Recent press suggests that auditors increasingly have a stake in what middlebox processing is performed and how, in ensuring that data is processed according to commercial and government standards [108], and government agencies seem increasingly ambitious in inserting wiretapping middleboxes in to public ISPs [78]. Tussles in packet processing often (but not always) center on the privacy of the data being transmitted and analyzed.

BlindBox shows how secure multi-party computation approaches – like searchable encryption – can ease tussle. Secure multi-party computation (SMPC) is a class of techniques that allow multiple parties to jointly compute a function over their inputs while keeping those inputs private [160]. SMPC eases tussle when conflicting parties do not object to the actions or goals of the other parties, but only object to the incidental loss of privacy due to the other parties' actions. BlindBox is one such case: users likely do not object to their traffic being inspected for attacks, but do object to the incidental privacy loss due to their data being decrypted in the process of detecting attacks. We suspect that some other tussles surround middleboxes can be aligned by multiparty computation techniques as well. For example, mobile phone ISPs must identify users at base-stations to authenticate their device and determine whether or not they have paid their bills; this identification step may worry users since it allows the ISP to physically track their location. Could a cryptographic technique at the authenticating middlebox allow the ISP to identify *that* the phone is permitted on the network and it's bills are paid without actually learning which user the device belongs to specifically?

Nonetheless, SMPC techniques only ease tussle when the conflict between players centers on incidental loss of privacy. When a player objects to another players action outright (e.g., a government wishes to inspect and store all data about a specific user who does not wish to be tracked, or an ISP wishes to filter traffic that a client wishes to send) SMPC offers little towards a solution.

Building cloud-inspired, general-purpose middlebox infrastructure opens the door to new network service deployments.

The cloud-computing blueprint for middlebox processing may not only serve to port *existing* network processing needs to new and more efficient infrastructure. As more clouds and ISPs deploy generic software packet processing infrastructure on their public networks, networking researchers, startups, and developers will have a platform for deploying new services. New proposals in protocol design, packet scheduling, network services, and security extensions might all be tested and deployed on such a platform. If NFV and the cloud-computing blueprint achieve widespread adoption, we may arrive at a future where network innovations are deployed as quickly and easily as startups and researchers deploy their code to clouds like EC2 and Azure today.

Bibliography

[1] A Peek into Extended Page Tables. https://communities.intel.com/community/itpeernetwork/datastack/blog/2009/06/02/a-peek-into-extended-page-tables.

[2] Alexa: The web information company. http://www.alexa.com/.

[3] Amazon Direct Connect. http://aws.amazon.com/directconnect.

[4] Amazon Route 53. http://aws.amazon.com/route53.

[5] Amazon Web Services Launches Brazil Datacenters for Its Cloud Computing Platform. http://phx.corporate-ir.net/phoenix.zhtml?c=176060&p=irol-newsArticle&ID=1639908&highlight=.

[6] Aryaka WAN Optimization. http://www.aryaka.com.

[7] Barracuda Web Security Flex. http://www.barracudanetworks.com/ns/products/web_security_flex_overview.php.

[8] CISCO: Quality of Service Design Overview. http://www.ciscopress.com/articles/article.asp?p=357102.

[9] Clang Static Analyzer. http://clang-analyzer.llvm.org/.

[10] CORD: Central Office Reimagined as a Datacenter. http://opencord.org/.

[11] Embrane. http://www.embrane.com/.

[12] Emerging Threats: Open Source Signatures. https://rules.emergingthreats.net/open/snort-2.9.0/rules/.

[13] Intel PRO/1000 Quad Port Bypass Server Adapters. http://www.intel.com/content/www/us/en/network-adapters/gigabit-network-adapters/pro-1000-qp.html.

[14] LXC - Linux Containers. https://linuxcontainers.org/.

[15] M57 packet traces. https://domex.nps.edu/corp/scenarios/2009-m57/net/.

[16] McAfee Network Security Platform. http://www.mcafee.com/us/products/network-security-platform.aspx.

[17] Network Monitoring Tools. http://www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html.

[18] Open Platform for NFV. https://www.opnfv.org/.

[19] OpenVPN. http://www.openvpn.com.

[20] OT Extension library. https://github.com/encryptogroup/OTExtension.

[21] Palo Alto Networks. http://www.paloaltonetworks.com/.

[22] Qosmos Deep Packet Inspection and Metadata Engine. http://www.qosmos.com/products/deep-packet-inspection-engine/.

[23] Radisys R220 Network Appliance. http://www.radisys.com/products/network-appliance/.

[24] Remus PV domU Requirements. http://wiki.xen.org/wiki/Remus_PV_domU_requirements.

[25] RightScale Cloud Management. http://www.rightscale.com/.

[26] Riverbed Completes Acquisition of Mazu Networks. http://www.riverbed.com/about/news-articles/press-releases/riverbed-completes-acquisition\-of-mazu-networks.html.

[27] Riverbed Virtual Steelhead. http://www.riverbed.com/us/products/steelhead_appliance/virtual_steelhead.php.

[28] ssldump. http://www.rtfm.com/ssldump/.

[29] Symantec | Enterprise. http://www.symantec.com/index.jsp.

[30] Symantec: Data Loss Protection. http://www.vontu.com.

[31] The GnuTLS Transport Layer Security Library. http://www.gnutls.org/.

[32] Tivoli Monitoring Software. http://www-01.ibm.com/software/tivoli/products/monitor.

[33] University of Toulouse Internet Blacklists. http://dsi.ut-capitole.fr/blacklists/.

[34] VMWare vCloud. http://vcloud.vmware.com.

[35] VMWare vSphere. https://www.vmware.com/support/vsphere.

[36] Vyatta Software Middlebox. http://www.vyatta.com.

[37] Wikipedia:seqlock. http://en.wikipedia.org/wiki/Seqlock.

[38] ZScaler Cloud Security. http://www.zscaler.com.

[39] Multi-Service Architecture and Framework Requirements. http://www.broadband-forum.org/technical/download/TR-058.pdf, 2003.

[40] Transparent caching using Squid. http://www.visolve.com/squid/whitepapers/trans_caching.pdf, 2006.

[41] Cloud Computing - 31 companies Describe Their Experiences. http://www.ipanematech.com/information-center/download.php?link=white-papers/White%20Book_2011-Cloud_Computing_OBS_Ipanema_http://www.ipanematech.com/information-center/download.php?link=white-papers/White%20Book_2011-Cloud_Computing_OBS_Ipanema_Technologies_EBG.pdf, 2011.

[42] A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. EndRE: An End-System Redundancy Elimination Service for Enterprises. In *Proc. USENIX NSDI*, 2010.

[43] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proc. ACM SOSP*, 2009.

[44] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet Caches on Routers: The Implications of Universal Redundant Traffic Elimination. In *Proc. ACM SIGCOMM*, 2008.

[45] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. ACM Symposium on Operating Systems Principles*, 2001.

[46] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[47] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, April 2010.

[48] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.80 edition, May 2014.

[49] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In *Proc. ACM CCS*, 2013.

[50] AT&T. Domain 2.0 White Paper. http://www.att.com/Common/about_us/pdf/AT&T%20Domain%202.0%20Vision%20White%20Paper.pdf.

[51] H. Ballani and P. Francis. CONMan: a Step Towards Network Manageability. In *Proc. ACM SIGCOMM*, 2007.

[52] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and Efficiently Searchable Encryption. In *Proc. IACR CRYPTO*, 2007.

[53] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient Garbling from a Fixed-Key Blockcipher. In *Proc. IEEE S&P*, 2013.

[54] T. Benson, A. Akella, A. Shaikh, and S. Sahu. CloudNaaS: a Cloud Networking Platform for Enterprise Applications. In *Proc. ACM Symposium on Cloud Computing*, 2011.

[55] BlueCoat. Comparing Explicit and Transparent PRoxies. https://bto.bluecoat.com/webguides/proxysg/security_first_steps/Content/Solutions/SharedTopics/Explicit_Transparent_Proxy_Comparison.htm.

[56] BlueCoat. SSL Encrypted Traffic Visibility and Management. https://www.bluecoat.com/products/ssl-encrypted-traffic-visibility\-and-management.

[57] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proc. IACR EUROCRYPT*, 2004.

[58] T. C. Bressoud and F. B. Schneider. Hypervisor-based Fault Tolerance. In *Proc. ACM SOSP*, 1995.

[59] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. USENIX OSDI*, 2008.

[60] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues. RFC 3234.

[61] S. Cheshire and M. Krochmal. NAT Port Mapping Protocol (NAT-PMP). RFC 6886, Apr. 2013.

[62] D. R. Choffnes and F. E. Bustamante. Taming the Torrent: a Practical Approach to Reducing Cross-ISP Traffic in Peer-to-Peer Systems. In *Proc. ACM SIGCOMM*, 2008.

[63] J. Chung, J. Seo, W. Baek, C. CaoMinh, A. McDonald, C. Kozyrakis, and K. Olukotun. Improving Software Concurrency with Hardware-assisted Memory Snapshot. In *Proc. ACM SPAA*, 2008.

[64] D. D. Clark et al. Tussle in cyberspace: defining tomorrow's Internet. *ToN*, June 2005.

[65] P. R. Clearinghouse. Chronology of data breaches . http://www.privacyrights.org/data-breach.

[66] E. G. Coffman, M. Elphick, and A. Shoshani. System Deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[67] Comcast. A Terabyte Internet Experience. http://corporate.comcast.com/comcast-voices/a-terabyte-internet-experience.

[68] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *Proc. ACM SOSP*, 2013.

[69] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proc. USENIX NSDI*, 2008.

[70] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic Systems. In *Proc. USENIX OSDI*, 2014.

[71] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *Proc. ACM ASPLOS*, 2009.

[72] Digital Corpora. 2009-M57-Patents packet trace. http://digitalcorpora.org/corp/nps/scenarios/2009-m57-patents/net/.

[73] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy. ETTM: a Scalable Fault Tolerant Network Manager. In *Proc. USENIX Network Systems Design and Implementation*, 2011.

[74] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *SOSP*, 2009.

[75] Dobrescu, Mihai and Argyraki, Katerina. Software dataplane verification. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 101–114, Berkeley, CA, USA, 2014. USENIX Association.

[76] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan. COLO: COarse-grained LOck-stepping Virtual Machines for Non-stop Service. In *Proc. ACM SoCC*, 2013.

[77] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proc. ACM SIGPLAN/SIGOPS VEE*, 2008.

[78] Electronic Fronteir Foundation. NSA Spying on Americans. https://www.eff.org/nsa-spying.

[79] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, May 1992.

[80] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[81] ETSI. List of Members. http://portal.etsi.org/NFV/NFV_List_members.asp.

[82] European Telecommunications Standards Institute. NFV Whitepaper. https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[83] S. Even, O. Goldreich, and A. Lempel. A Randomized Protocol for Signing Contracts. *Commun. ACM*, 28(6):637–647, June 1985.

[84] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. Generic Routing Encapsulation (GRE). RFC 2784.

[85] M. Flajslik and M. Rosenblum. Network Interface Design for Low Latency Request-Response Protocols. In *Proc. USENIX ATC*, 2013.

[86] S. Frankel and S. Krishnan. IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. RFC 6071.

[87] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Proc. IEEE FOCS*, 2013.

[88] A. Gember, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proc. ACM SIGCOMM*, 2014.

[89] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM.

[90] C. Gentry. Fully Homomorphic Encryption using Ideal Lattices. In *Proc. ACM STOC*, 2009.

[91] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic Evaluation of the AES Circuit. In *Proc. IACR CRYPTO*, 2012.

[92] G. Gibb, H. Zeng, and N. McKeown. Outsourcing Network Functionality. In *Proc. ACM Workshop on Hot Topics in Software Defined Networking*, 2012.

[93] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. Reusable Garbled Circuits and Succinct Functional Encryption. In *Proc. ACM STOC*, 2013.

[94] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall. Improving the Reliability of Internet Paths with One-hop Source Routing. In *Proc. USENIX Operating Systems Design and Implementation*, 2004.

[95] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An Application-Level Kernel for Record and Replay. In *Proc. USENIX OSDI*, 2008.

[96] M. Hajjat, X. Sun, Y.-W. E. Sung, D. A. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani. Cloudward Bound: Planning for Beneficial Migration of Enterprise Applications to the Cloud. In *Proc. ACM SIGCOMM*, 2012.

[97] S. Han et al. Packetshader: A GPU-accelerated software router. In *SIGCOMM*, 2010.

[98] L.-S. Huang, A. Rice, E. Ellingsen, and C. Jackson. Analyzing Forged SSL Certificates in the Wild. In *Proc. IEEE S&P*, 2014.

[99] Intel. Data Plane Development Kit. `http://dpdk.org/`.

[100] J. Jarmoc. SSL/TLS Interception Proxies and Transitive Trust. *Presentation at Black Hat Europe*, 2012.

[101] D. Joseph and I. Stoica. Modeling middleboxes. *IEEE Network*, 22(5):20–25, September 2008.

[102] D. A. Joseph, A. Tavakoli, and I. Stoica. A Policy-Aware Switching Layer for Data Centers. In *Proc. ACM SIGCOMM*, 2008.

[103] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic Searchable Symmetric Encryption. In *Proc. ACM CCS*, 2012.

[104] J. Katz, A. Sahai, and B. Waters. Predicate Encryption Supporting Disjunctions, Polynomial Equations, and Inner Products. In *Proc. IACR EUROCRYPT*, 2008.

[105] A. Kingsley-Hughes. Gogo in-flight Wi-Fi serving spoofed SSL certificates. *ZDNet*, 2015.

[106] R. Kohavi and R. Longbotham. Online experiments: Lessons learned. *Computer*, 40(9):103–105, 2007.

[107] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[108] R. Kumar. Security Audit Policy is Essential in Ensuring Network Security. `http://www.infosecurity-magazine.com/opinions/security-audit-policy-essential/`.

[109] V. Kundra. 25 Point Implementation Plan to Reform Federal Information Technology Management. Technical report, US CIO, 2010.

[110] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *Proc. ACM SIGMETRICS*, 2010.

[111] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[112] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. IEEE CGO*, 2004.

[113] Y. Lindell and B. Pinkas. A Proof of Security of Yao's Protocol for Two-Party Computation. *J. Cryptol.*, 22:161–188, April 2009.

[114] J. R. Lorch, A. Baumann, L. Glendenning, D. Meyer, and A. Warfield. Tardigrade: Leveraging Lightweight Virtual Machines to Easily and Efficiently Construct Fault-Tolerant Services. In *Proc. USENIX NSDI*, 2015.

[115] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proc. USENIX NSDI*, 2014.

[116] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Recursively Cautious Congestion Control. In *Proc. USENIX NSDI*, 2014.

[117] M. Naor and B. Pinkas. Oblivious Transfer with Adaptive Queries. In *Proc. IACR CRYPTO*, 1999.

[118] Ofcom. Average monthly fixed broadband data volume per capita in 2008 and 2014 (in GB). Statista - The Statistics Portal.http://www.statista.com/statistics/374998/fixed-broadband-data-volume-per-capita/.

[119] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 121–136, New York, NY, USA, 2015. ACM.

[120] A. Panda, K. Argyraki, M. Sagiv, M. Schapira, and S. Shenker. New Directions for Network Verification. In *SNAPL*, 2015.

[121] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *Proc. USENIX Operating Systems Design and Implementation*, 2016.

[122] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, pages 2435–2463, 1999.

[123] PCI Security Standards Counsel. Payment Card Industry Data Security Standard. https://www.pcisecuritystandards.org/.

[124] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proc. ACM EuroSys*, 2012.

[125] B. Pfaff, J. Pettit, T. Koponen, E. J. 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 *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 117–130, Berkeley, CA, USA, 2015. USENIX Association.

[126] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proc. ACM SOSP*, 2013.

[127] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building Web Applications on Top of Encrypted Data using Mylar. In *Proc. USENIX NSDI*, 2014.

[128] R. Potharaju and N. Jain. Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters. In *Proc. ACM IMC*, 2013.

[129] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 27–38, New York, NY, USA, 2013. ACM.

[130] P. Quinn and T. Nadaeu. Problem Statement for Service Function Chaining. RFC 7498, 2015.

[131] M. O. Rabin. How to Exchange Secrets with Oblivious Transfer. TR-81, Aiken Computation Lab, Harvard University `http://eprint.iacr.org/2005/187.pdf`, 1981.

[132] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico Replication: A High Availability Framework for Middleboxes. In *Proc. ACM SoCC*, 2013.

[133] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. USENIX NSDI*, 2013.

[134] A. Rao, J. Sherry, A. Legout, W. Dabbout, A. Krishnamurthy, and D. Choffnes. Meddle: Middleboxes for Increased Transparency and Control of Mobile Traffic. In *Proc. CoNEXT Student Workshop*, 2012.

[135] L. Rizzo. netmap: a Novel Framework for Fast Packet I/O. In *Proc. USENIX ATC*, 2012.

[136] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proc. USENIX Large Installations Systams Administration*, 1999.

[137] Runa. Security vulnerability found in Cyberoam DPI devices (CVE-2012-3372). *Tor Project Blog*, 2012.

[138] R. Russell. virtio: Towards a De-facto Standard for Virtual I/O Devices. *ACM OSR*, 42(5):95–103, 2008.

[139] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. ACM SOSP*, 1997.

[140] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[141] V. Sekar, N. Egi, S. Ratnasamy, M. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. USENIX Network Systems Design and Implementation*, 2012.

[142] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The Middlebox Manifesto: Enabling Innovation in Middlebox Deployment. In *Proc. ACM Workshop on Hot Topics in Networking (HotNets)*, 2011.

[143] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. ACM SIGCOMM*, 2012.

[144] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. Cryptology ePrint Archive, Report 2015/264, 2015. http://eprint.iacr.org/.

[145] Shira Levine. Operators look to embed deep packet inspection (DPI) in apps; Market growing to $2B by 2018. Infonetics Research. http://www.infonetics.com/pr/2014/2H13-Service-Provider-DPI-Products-Market-Highlights.asp.

[146] G. J. Silowash, T. Lewellen, J. W. Burns, and D. L. Costa. Detecting and Preventing Data Exfiltration Through Encrypted Web Sessions via Traffic Inspection. Technical Report CMU/SEI-2013-TN-012, CERT Program, 2013.

[147] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *Proc. IEEE S&P*, 2000.

[148] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. *IEEE/ACM Transactions on Networking*, 12(2):205–218, April 2004.

[149] R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, Aug. 1985.

[150] A. Su, D. Choffnes, A. Kuzmanovic, and F. Bustamante. Drafting behind Akamai (Travelocity-Based Detouring). In *Proc. ACM SIGCOMM*, 2006.

[151] The Snort Project. Snort users manual, 2014. Version 2.9.7.

[152] United States Congress. Family Educational Rights and Privacy Act. Public Law 93-380, 1974.

[153] United States Congress. Health Insurance Portability and Accountability Act of 1996. Public Law 104-191, 1996.

[154] V. Valancius, N. Laoutaris, L. Massouli'e, C. Diot, and P. Rodriguez. Greening the Internet with Nano Data Centers. In *Proc. ACM CoNEXT*, 2009.

[155] N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, N. Weaver, and V. Paxson. Beyond the Radio: Illuminating the Higher Layers of Mobile Networks. In *Proc. ACM MobiSys*, 2015.

[156] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proc. ACM ASPLOS*, 2012.

[157] Verizon. 2015 Data Breach Investigations Report. http://www.verizonenterprise.com/DBIR/2015/.

[158] G. Vigna. ICTF Data. https://ictf.cs.ucsb.edu/#/.

[159] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *OSDI*, 2004.

[160] Wikipedia. Secure Multi-Party Computation. https://en.wikipedia.org/wiki/Secure_multi-party_computation.

[161] S. Woo, J. Sherry, S. Han, S. Ratnasamy, S. Shenker, and S. Moon. State Abstractions for Scaling Stateful Network Functions. 2016.

[162] X. Xu, Y. Jiang, T. Flach, E. Katz-Bassett, D. Choffnes, and R. Govindan. *Investigating Transparent Web Proxies in Cellular Networks*, pages 262–276. Springer International Publishing, 2015.

[163] X. Xu, Y. Jiang, T. Flach, E. Katz-Bassett, D. Choffnes, and R. Govindan. Investigating Transparent Web Proxies in Cellular Networks. In *Proc. Passive and Active Measurements (PAM)*, 2015.

[164] A. C. Yao. How to Generate and Exchange Secrets. In *Proc. IEEE FOCS*, 1986.