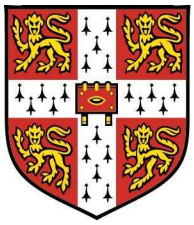
Structural Traffic Analysis for Network Security Monitoring

Christian Peter Kreibich

Hughes Hall



A dissertation submitted to the University of Cambridge
for the degree of Doctor of Philosophy

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
UK

Email: christian.kreibich@cl.cam.ac.uk

January 9, 2007

To my family, for the love and support.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

No part of this dissertation has already been, or is currently being submitted for any such degree, diploma or other qualification.

This dissertation does not exceed sixty thousand words.

Abstract

Traffic on the Internet is constantly growing more complex and multi-faceted. This natural evolution is mirrored by novel kinds of malicious traffic: automated attacks subvert thousands of machines at a time, enabling a wide range of subsequent attacks and nuisances such as distributed denial-of-service attacks and generation of vast amounts of unsolicited electronic mail. Consequently, there is a strong need to be able to tell malicious traffic from the benign. In this dissertation, I take several steps toward this goal. By leveraging structural aspects of network traffic, typical as well as malicious activity on computer networks can be fingerprinted and contrasted.

A first avenue is the analysis of application-level flow content. I investigate the suitability of biological sequence alignment algorithms in the adversarial environment of the networking domain, and introduce a novel algorithm that is well over an order of magnitude faster than the commonly used Smith-Waterman algorithm while maintaining much of its flexibility. I introduce a novel and highly flexible model of traffic content based on sequence alignment, Common Substring Graphs, and demonstrate its versatility in a study of application-level protocol classification.

Switching focus to the malicious, I pioneer the use of honeypots and sequence analysis algorithms for automated fingerprinting of malware and thus demonstrate the feasibility of fully automated network-based malware signature generation. I propose a second approach to fingerprinting the malicious: Packet Symmetry focuses on network-level structure and leverages the intuition that well-behaved applications do not transmit vastly more packets than they receive. Traffic analysis confirms the feasibility of employing packet symmetry for edge-based, ingress-focused prevention strategies for volume-based attacks.

Acknowledgements

First of all, I cannot thank Mark Handley enough for using two widescreen flatpanels while at ICSI in 2000, because without them my jaw very likely would not have dropped in front of his office. The conversation that followed started it all. I am deeply indebted to both him and Vern Paxson for giving me the chance to do all the exciting work during that summer which eventually convinced me to undertake a Ph.D., and for recommending Jon Crowcroft as my future supervisor. The fact that I am now writing these lines of course means that this has proved to be an excellent recommendation, and I am immensely grateful to Jon for the amazingly fun time it has been, both inside and outside of the lab. Without all the support and freedom he gave me it never would have been the same. Thanks Jon! Similarly, my gratitude goes to Derek MacAuley and Intel Research Cambridge for funding my time in Cambridge. Thank you for making it all possible. Steven Hand has given helpful advice many a time and has always been encouraging and up for a laugh. Thanks Steve, "wir machen durch..."

I was lucky enough to end up in the best office of at least all of Cambridge, in terms of the view as well as the company—Alex, Anil, Eva, and of course esteemed Volta himself. Thank you so much and $\epsilon\upsilon\chi\alpha\rho\iota\sigma\tau\acute{\omega}$ $\pi o\lambda\acute{\upsilon}$ for the amazing time it has been. Thanks to you I now know that marathons are run by real people, might try to learn a sentence in every language I can think of, got to taste amazing food too many times to count, and have experienced the very large weight of marble busts. To our regular office visitors, particularly Andy and Euan, thanks for taking the time to stop by, discuss and provide feedback, and for all the lounging on the couch!

Contents

1

Introduction

"Mmm. This is good. This is really good. What is this?"
— Princess Fiona in *Shrek*.

In this dissertation I introduce novel ways to distinguish malicious traffic from the benign. By taking advantage of structural aspects of network traffic exposed by suitable filtering of input traffic, typical as well as malicious activity on computer networks can be fingerprinted and contrasted. I will show how application-level content and network-level flow composition can be used to extract the structure of application-layer protocols, to identify malicious flow content, and to prevent certain classes of denial-of-service attacks.

I motivate the topic of this dissertation and state my thesis in Section 1.1. Next, I briefly outline the structure of the dissertation in Section 1.2 and state its contributions explicitly in Section 1.3. Large parts of this dissertation have been previously published; I enumerate those publications Section 1.4 and also mention some which address related topics that I have worked on besides the dissertation.

Finally, it may help the reader to know that several elements of the electronic version of the dissertation are clickable to ease navigation. URLs are generally linked. Entries in the table of contents take you to the respective chapters and sections. Chapter titles and headers link back to the table of contents, while section numbers link to the beginning of the current chapter. Citations lead to the corresponding entry in the bibliography, where each entry lists the numbers of all pages on which the entry is cited. Those numbers link back to to the respective citations throughout the document.

1.1 Motivation

Traffic on the Internet is not only constantly growing in volume, it is also growing increasingly more complex and multi-faceted. New applications are introduced constantly, often without clear network-level specifications or even openly trying to obscure their presence, creating new usage patterns and traffic content. At the same time, older applications decline. This healthy evolution is challenged by an onslaught of equally inventive abusive traffic: a poorly secured network edge allows large-scale automation of system break-ins through the injection of malicious content into the communication, enabling a wide range of subsequent attacks and nuisances such as distributed denial-of-service attacks and generation of vast amounts of unsolicited electronic mail.

As a consequence, the importance of network traffic monitoring has increased in tandem, for two reasons. First, monitoring is necessary to improve one's understanding of the *typical* activity on a network, for example in order to keep track of current application usage, be able to provision a network economically, forecast application growth, or validate service differentiation. Second, traffic is monitored to detect and, ideally, filter out *abusive* activity, where abuse is typically defined by site-local administrative policy, and may include the use of undesirable applications, creation of malicious content, abusive volumes of traffic, aggressive scanning behaviour, etc.

Novel approaches to network monitoring are required to help operators maintain a clear picture of the normal and malicious activity on their networks. An essential requirement here is *automation*: on the one hand, analyst time generally is a scarce resource; on the other, malware has been shown to propagate at time scales that essentially exclude the possibility of human analysis.

This dissertation touches on all of these aspects. I argue the following thesis:

> *Network traffic exhibits structural properties which, given suitable filtering and vantage points, permit fully automated derivation of fingerprints of previously unknown network applications and attacks. The generated fingerprints enable accurate detection as well as filtering of such network activity.*

1.2 Outline

This dissertation is structured as follows. I begin by surveying the development of network security over the last decades in Chapter 2, to put my work in general context. Much of the dissertation dedicates itself to content-based traffic analysis, for which I lay the foundation in Chapter 3: I introduce techniques for tracking the message exchange between communication endpoints and use them for the study of sequence analysis models adapted from the bioinformatics domain. I present variants of sequence alignment algorithms suitable for network traffic and outline their trade-offs, and introduce the consideration of such algorithms from an adversarial perspective. I present a first use of these techniques in Chapter 4 in the form of a content-based model of application-layer protocol activity found on a network. In Chapter 5 I focus on malicious traffic. I present techniques for automatically detecting and fingerprinting the characteristics of attacks as they occur, as well as a proactive monitoring and enforcement technique for the prevention of denial-of-service attacks. Finally, I conclude the dissertation with Chapter 6.

1.3 Contributions

In this dissertation I make the following contributions:

- I investigate the suitability of a number of biological sequence alignment algorithms in the adversarial network environment, and find that such algorithms have to be employed with great care in order to yield the desired results. I introduce a novel variant of the Jacobson-Vo algorithm that is able to accommodate flexible alignment models akin to dynamic programming as used by the popular Smith-Waterman algorithm, while outperforming the latter by a factor of 33 on average and up to 58.5 times in the best case. (Chapter 3.)
- I introduce Common Substring Graphs (CSGs), a structural model of flow content taking into account the frequency, length, and location of strings common to a set of flows. The model retains information about the input flows with high fidelity, making it useful for a

number of applications. Using thorough evaluation, I demonstrate very good suitability of the model for one such application, namely application-layer protocol classification. (Chapter 4.)

- I pioneer the use of honeypots and sequence analysis algorithms for automated fingerprinting of malware and demonstrate the feasibility of fully automated and fast malware signature generation. (Section 5.3.)
- I present the notion of Packet Symmetry, a network-level prevention strategy for volume-based attacks, leveraging the insight that well-behaved applications should not transmit vastly more packets than they receive. Through traffic analysis I confirm the feasibility of highly universal packet-level differentiation between benign and malicious traffic. (Section 5.4.)

1.4 Published Work

Individual parts of the work presented in this dissertation have appeared in the following publications:

- J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. Voelker: Unexpected Means of Protocol Inference. Internet Measurement Conference (IMC), 2006, Rio de Janeiro, Brazil.
- C. Kreibich and J. Crowcroft: Efficient Sequence Alignment of Network Traffic. Internet Measurement Conference (IMC), 2006, Rio de Janeiro, Brazil.
- C. Kreibich, A. Warfield, J. Crowcroft, S. Hand, and I. Pratt: Using Packet Symmetry to Curtail Malicious Traffic. Fourth Workshop on Hot Topics in Networks (HotNets-IV), 2005, College Park/Maryland, USA.
- C. Kreibich and J. Crowcroft: Honeycomb — Creating Intrusion Detection Signatures Using Honeypots. 2nd Workshop on Hot Topics in Networks (HotNets-II), 2003, Boston, USA.

The following article presents a central component of my toolchain for offline network traffic manipulation and has been used in almost all of the work listed above.

- C. Kreibich: Design and Implementation of Netdude, a Framework for Packet Trace Manipulation. Usenix Technical Conference, Freenix Track, 2004, Boston, USA. Awarded Best Student Paper.

During my work on this dissertation I have contributed to other published work that is closely related to structural traffic analysis but slightly outside scope for inclusion:

- H. Dreger, C. Kreibich, R. Sommer, and V. Paxson: Enhancing the Accuracy of Network-based Intrusion Detection with Host-based Context. Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2005), Vienna, Austria.
- C. Kreibich and R. Sommer: Policy-controlled Event Management for Distributed Intrusion Detection. 4th International Workshop on Distributed Event-Based Systems (DEBS'05), 2005, Columbus/Ohio, USA.
- A. Moore, J. Hall, C. Kreibich, E. Harris, and I. Pratt: Architecture of a Network Monitor. Passive and Active Measurements Workshop, La Jolla, California, 2003.

Much of the work listed above originated from internships with Vern Paxson at ICSI in 2004 and 2005.

2

Background

"And leaving the door open is the worst mistake."
— Scare instructor in *Monsters, Inc.*

In this chapter I give an overview of the evolution of network security in the Internet from its conception to the present day, illustrating the growing application of content-based traffic analysis techniques for understanding and enforcing site-local traffic management policies on computer networks. Since network surveillance gives operators access to potentially highly sensitive information about the users, it is of great importance to be aware of the legal implications of such monitoring. Using Germany, the United Kingdom, and the Unites States as examples, I briefly survey the relevant legislation. The goal of this chapter is to put into perspective the techniques I introduce in the remaining chapters of this dissertation.

2.1 Network Security

Information has become one of the (if not *the*) most important asset in our society. Many pieces of information are public and meant to be accessible by anyone, while others need to be guarded carefully. In this context, network security is mentioned frequently enough in the media that it is worth clarifying what exactly I mean by it. When speaking of network security in this thesis, I adopt the description by Shaffer and Simon [138]:

> Network security refers to all hardware and software functions, characteristics, features, operational procedures, accountability measures, access controls, and administrative and management policy required to provide an acceptable level of protection for hardware, software, and information in a network.

The goals of network security are no different from those of any effort to protect information: *availability* guarantees that requested information can be provided when needed, *confidentiality* restricts access to information to authorised individuals, while *integrity* ensures that information remains unmodified and complete. Attacks on networked systems attempt to exploit one or more vulnerabilities in those systems that allow the mechanisms that enforce these goals to be bypassed. A successful attack can be composed of many steps; Schneier's attack trees structure these steps [133]. Malicious activity generally means any activity that aims to find or exploit vulnerabilities.

availability, confidentiality, integrity

attack trees

It is important to note that security is always a trade-off. Network security, as any security investment, is an additional expenditure and thus subject to calculations of return on investment (ROI). If an organisation believes the financial cost imposed by a security breach s smaller than that of the investment necessary to protect the network, it is unlikely to make that investment. Methods of risk estimation for ROI calculations are controversial, see the book by Alberts and Dorofee [1] for an example.

2.2 Evolution of Network Security in the Internet

In the following I present an overview of the evolution of network security over the decades, focusing on the awareness, understanding of the problem, and measures taken, rather than trying to give a complete account of the history of the Internet. Any historical facts that are not referenced explicitly are taken from the general literature [111, 121, 64].

2.2.1 1960s

The 1960s were dedicated to the exploration of the very fundamentals of computer networking: cross-platform communication, and the use of packet switching as opposed to circuit switching, for increased robustness against structural failures. Baran's seminal paper [11] addresses security in the sense of preserving connectivity in the presence of node and link failures. In contrast to popular opinion, this damage resilience was only

one of the motivators besides others, such as J.C.R Licklider's vision of a global network infrastructure [96].

2.2.2 1970s

By the early 1970s, the Arpanet had become an important platform for collaborative research. Email, TCP/IP and Ethernet are developed. Cerf and Kahn's TCP/IP paper [25] thoroughly treats addressing, routers, reliable transfer, and flow control, but has no notion of a threat model or similar security-focused analysis. In 1979, John Shoch and Jon Hupp of Xerox PARC develop a small, self-replicating program that finds groups of idle local machines that the authors want to use instead of the mainframe, for parallel computations. The program is flawed and causes a large fraction of the local host population to hang. Shoch and Hupp implemented a *worm*: a self-contained program that can replicate itself across multiple machines autonomously. Network connectivity is a basic requirement for computer worms, thus increased connectivity also implies a larger potential attack surface for worms. However, security flaws in the Arpanet are not a major concern at this point. At this point, attackers are much more interested in exploring the various ways in which the in-band signalling channels of the telephony systems can be tricked into permitting cheap long-distance calls [148].

computer worm

2.2.3 1980s

This decade marks the shift from time-sharing few machines to large deployments of interconnected end-user PCs with a common LAN technology, typically Ethernet, and the network officially becomes the Internet. Access is opened up to basically anyone with the right equipment, and thus laid the basis for the first well-known security violations on the Internet. Security-related terminology starts to consolidate. James Anderson lays one of the cornerstones of what is to become the field of intrusion detection when he establishes the idea of using audit data for detecting misuse in a 1980 paper [3]. In 1983, Ph.D. student Fred Cohen and his adviser Len Adleman are the first to academically term code that can propagate "in the wild" a computer virus [38]. Cohen's now commonly used

computer virus

non-mathematical definition of a virus is as follows:

> A virus is a program that can 'infect' other programs by modifying them to include a [...] version of itself.

Prior to Cohen and Adleman, the term had been used in science fiction [66, 21] and code was released that falls in the same category (such as Elk Cloner [152]). In contrast to worms, viruses require external activity to propagate. Cohen's work marks the beginning of the development of a myriad of different kinds of computer viruses. Szor [152] enumerates the many digital niches in which viruses appear in great detail.

In 1986, Cliff Stoll accidentally discovers a large-scale international operation to break into computers in the United States [150]. On 2 November 1988, Robert Morris releases the Morris Worm (also referred to as the Internet Worm), the first Internet-wide uncontrolled self-replicating code targeting a set of specifically preselected weaknesses in widely deployed software [145]. It leads to the development of the first Computer Emergency Response Team (CERT) [155]. In 1987, Dorothy Denning publishes "An Intrusion-Detection Model," describing a model for profiling normal behaviour and using deviations from the norm as the signal of misuse [50]. The work establishes the term "intrusion detection" and her model remains the basis of most anomaly-based intrusion detection systems (IDSs) today. The concept of a firewall as a filtering mechanism for unwanted traffic appears in the literature in 1988 [103] and leads to a wide array of implementations across a large design space spanning protocol layers, filtering dynamics, and physical deployment scenarios such as demilitarised zones.

Morris worm

intrusion detection

2.2.4 1990s

At the beginning of the decade, vulnerabilities in server software, then exclusively a UNIX domain, are published regularly. Heber et al. establish the notion of a specifically network-based intrusion detection system (NIDS) in a paper appearing in 1990 [72]. Their system is anomaly-based, following Denning's model. Large-scale break-ins, often facilitated through unsafe configuration of the `rlogin` tools, appear in isolation [27, 28]. Social engineering, essentially non-technical means to trick people

NIDS

into handing out sensitive information, finds a new market through email scams [26]. Attackers attempt to monitor network traffic in strategic location in order to extract user account details from the then entirely unencrypted traffic [29]. In 1995, the SSH protocol suite appears and with its ability to encrypt connections ends the era of large-scale password sniffing from interactive user sessions.

1996 is the year of the first denial-of-service (DoS) attacks, in which attackers flood victim servers with large amounts of request traffic [30] or with maliciously crafted packets [31, 32, 33], thus crashing or slowing down the machine or its access links, and preventing it from servicing legitimate requests. Denial-of-service attacks remain one of the largest security threats on the Internet to the present day. On March 26 1999, the first email-borne worm and macro virus, Melissa, appears and clogs mail servers around the planet [34], causing massive financial damage. Estimates vary widely but are generally placed in the order of hundreds of millions of US dollars. That year also sees the advent of distributed denial-of-service attacks (DDoS attacks), in which attackers use a large number of subverted machines to flood a victim with traffic.

DoS

DDoS

At the end of the decade, two major open-source NIDS implementations appear: Bro [119] and Snort [128]. Bro becomes the predominant system for research endeavours, and is the platform for much of the work presented in this dissertation.

The 1990s are the decade in which the Internet is opening up to commercial use. With it comes the entry into the network, or more precisely, into the edge, of large numbers of hosts running Microsoft operating systems. This much younger codebase, combined with a typically weak sense of security on part of the "operators" of these machines, offers a ripe ground for exploitation and leads to a shift in attacker focus toward the Windows operating systems.

2.2.5 2000-Present

The boundaries between worms, viruses, and the activities they perform become increasingly blurred. "Malware" becomes an umbrella term referring to any kind of malicious software. Widespread broadband access

malware

leads to around-the-clock connectivity of large numbers of poorly guarded end-user PCs.

The beginning of the new millennium is an era of severe worm outbreaks. Sadmind, CodeRed, CodeRed II, Nimda, Slammer, Blaster, and Sober repeatedly cause significant financial damage, denials of service on affected systems, and hindered general network operations world-wide [147, 165]. Weaver et al. [164] present a taxonomy of different kinds of computer worms. Among other things, worms highlight the deficiencies of thinking of network defense as a perimeter problem: once malware is active inside the network, defenses against the outside are useless. The widespread use of wireless networking and laptops exacerbates this problem.

honeypots

They idea of mining regular networks with traps for the attackers to step into, termed *honeypots*, becomes mainstream. Such systems are specifically set up for thorough analysis of attacker activity, and little else [146]. Honeypots are a central component of the work I present in Chapter 5.

economic motives

botnets

A major shift in attacks in recent years is *commercial* motivation for exploitation of vulnerabilities. Instead of only bragging about the latest break-ins, the control of large numbers of machines becomes automated to the degree that allows attackers to control such *botnets* for extorting money: only after payment is the victim relieved of DDoS attacks, or returned encrypted files. Delivery of unsolicited email, often happening through the infected machine's legitimate SMTP server, is another common application. Current botnets have a large command set suitable for automated updates of the malware, sniffing user input, scanning for vulnerabilities, etc. At present, they often use IRC and IRC-like systems as the communications layer since IRC lends itself well to commanding large numbers of clients [42, 65]. Economic incentives have also started to be employed by security vendors: *vulnerability markets* [132] promise to pay money to discoverers of new vulnerabilities for the vendor's privilege of learning of — and potentially disarming — the vulnerabilities before the competition.[1]

[1] See for example `http://labs.idefense.com/vcp.php`.

2.3 Arms Races in Network Security

The constant theme in the evolution of network security over the decades is one of measures and countermeasures, that is, an arms race. When attackers adopted a new strategy, site administrators developed countermeasures, and vice versa. This phenomenon is in no way unique to the network security domain but can be observed in any attack/defense scenario. In the field of network security, the anti-virus industry is an embodiment of such a race. Another good example is the introduction of network intrusion detection systems. After the first systems saw widespread deployment, it quickly became clear that extracting flow content unambiguously is hard, and that there is potential for evading the monitor [125]. IDSs improved their algorithms and additional anti-evasion techniques were proposed [71, 139], but the arms race has never been clearly decided in favour of the attackers or the defenders. I discuss this further in Section 5.2.1, in the context of finding ways to fingerprint the malicious.

Several aspects of the network security arms race are worth noting. First of all, it is of limited predictability: external influences can shift motivations of the involved parties in unforeseeable ways. Second, not taking a step that makes life harder for the other side is a chance missed, unless taking that step brings with it negative side effects that are bigger than the improvement itself. Third, it is also a race in a more literal sense: the involved timescales are constantly decreasing. This holds for both the time from discovery of a vulnerability to attempted exploitation as well as to the development of a fix for the vulnerability and of signatures identifying exploitation attempts [6]. The development of a technique to derive such signatures without human intervention is the main contribution of Chapter 5 of this dissertation. Fourth, the race is here to stay. While there can be no doubt that present-day software engineering produces suboptimal products due to economic disincentives and lacking expertise, no matter how well the architecture of the future Internet will work, people will always attempt to attack it. The question is only how easy it will be for them to succeed. From a technical point of view, the arms race is a constantly driving factor behind the development of more sophisticated attack and defence techniques.

2.4 Detecting Malicious Behaviour

2.4.1 Purpose, Mode, and Consequence of Detection

There are many reasons why a site might operate an infrastructure for detecting malicious activity. They are closely related to the security policy a site operates under, that is, detection will be a building block for enforcing the policy. Typically the goal will be better understanding of activity on the network and to prevent abuses of system resources. Detection mechanisms can be classified along multiple dimensions.

2.4.1.1 *Strategy and Policy*

There are two major approaches to detecting malice [8]. First, we can define what behaviour we see as normal, and report deviation from such a profile. This is known as anomaly detection and has seen a considerable amount of research over the years. At the network level, the focus is often on modelling statistical traffic parameters such as connection rates, transfer volumes, contact habits, etc, frequently with probabilistic classifiers. Alternatively, we can formulate precise signatures of malicious behaviour, and detect their occurrence in the network. Such misuse detection has been embraced by industrial vendors early on and mostly consists of content-based signature detection for binary classification. A major benefit of binary classification, assuming it is correct in its judgement of input traffic, is that it makes a clear statement that eases processing of the classified event. I discuss binary classification further in Section 2.4.2.

anomaly detection

misuse detection

Anomaly and misuse detection can be combined in whichever way best serves a site's security policy. The effect of a detection is similarly up to policy and could range from warnings issued to the administrator to immediate termination of a machine's network connectivity. Policy is so important because there cannot be a single standard for what is worth alerting to. For example, one can ask whether the occurrence of an attempted Microsoft IIS exploit in a UNIX-only LAN is worth an alarm, or at which point a scanning source is deemed aggressive enough to ban it from further communication. It will depend entirely on the preferences of the local site.

2.4.1.2 *Location, Distribution, and Focus*

The field of vision that a detection system has on a network depends on where it is deployed and strongly affects potential results and performance. The more input the system has, the more it can potentially alert to, but the higher will be the computational requirements. Many deployment scenarios are feasible; all of them have different advantages and drawbacks.

Host-based systems can monitor in detail all activity on a host, but are restricted to just that host. Network-based systems, on the other hand, see a wide range of activity, but must infer from the network the activity on the end hosts, which is nontrivial [125]. OS virtualisation in combination with servicing a large IP address range from a small set of physical hosts is an interesting way to widen a host-based system's focus. A key aspect in network-based monitoring is the depth to which the traffic is analysed. I will discuss this separately below in Section 3.2. This thesis presents a variety of techniques ranging from shallow packet counts to deep content analysis of reassembled traffic flows, combining several of the approaches outlined above.

Even network-based detectors can only monitor what they observe, and they too remain blind to highly distributed activity unless the observations of multiple systems are combined into a larger, distributed detector. Timely distributed detection is an absolute requirement for stopping fast-spreading malware, since the chances of stopping such phenomena become extremely small once the infection has crossed its epidemic threshold [109, 166].

Unfortunately, the current sophistication level of distributed monitoring is rather primitive: I know of *no* set of sites sharing descriptions of attacker activity in even semi-automated fashion besides elementary shared blacklists of IP addresses, for example to filter out known originators of unsolicited electronic mail [123, 17]. This is very far from the sophisticated, fully automated propagation and filtering architectures required to make real-time containment of Internet-scale epidemics a reality. As an inital step in that direction, I have contributed to the extension of the Bro IDS to a fully distributed, event-based system [85], providing a testbed for experimenting with such infrastructures. The system is fully imple-

mented, publically available,[2] and actively used in test environments at institutions such as ICIR, Lawrence Berkeley National Laboratory, and Technische Universität München.

2.4.2 Binary Classification

Many kinds of network security systems attempt to detect malicious behaviour, for example virus scanners or intrusion detection systems. Like any detection system, such devices can err in two ways: they can report a detection when none exists (a false positive), or they can remain silent even though there is malicious activity in progress (a false negative). The goal is to minimise both kinds of errors while at the same time maximising detection.

false positive & negative

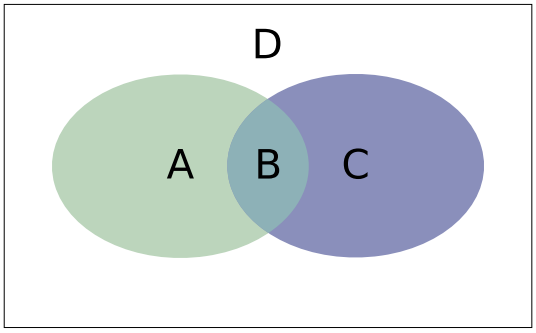


Figure 2.1. Outcome possibilities in binary classification.

Two measures for the degree to which this succeeds are *precision* Pr and *recall* Re, as illustrated in Figure 2.1. Let the circle to the left be the actual set we want to detect, and the one on the right the one we do detect. Thus, A represents the false negatives, B and C constitute the true and false positives, respectively, and D comprises the remaining true negatives. Precision and recall are then defined as follows:

precision, recall

$$Pr = \frac{B}{B+C} \qquad Re = \frac{B}{A+B}$$

In binary classification, *sensitivity* Se and *specificity* Sp measure similar notions. Sensitivity is the same as recall. Specificity is closely related to pre-

[2] See `http://www.bro-ids.org`

cision, but focuses on the entire test set:

$$Se = \frac{B}{A+B} \qquad Sp = \frac{D}{C+D}$$

The *false positive ratio* FP is the ratio of false positives to all negatives and can be expressed through Sp:

$$FP = \frac{C}{C+D} = 1 - Sp$$

The above measures point out two things.

- Firstly, true positives alone do not make a good detector if false positives are substantial, and vice versa. That is, precision/specificity and recall need to be evaluated in combination.
- Secondly, and more generally, there exists a duality between knowing the benign and detecting the malicious. If we know exactly what is benign, then anything that falls outside of that definition has to be malicious, and vice versa. This duality is an underlying theme of this thesis and will be referred to repeatedly.

In the network security context, false positives can lead to denial of service by preventing users from accessing legitimate files, or because legitimate network connections are incorrectly terminated [24, 151]. Much of the security arms race comes down to attackers trying to making it harder to identify the malicious while the defenders improve their classification techniques. Chapter 4 of this dissertation is dedicated to the identification of the benign, while Chapter 5 discusses approaches that automate the identification of the malicious.

2.5 Legal Implications of Network Monitoring

Monitoring network traffic can easily provide site operators access to potentially sensitive information about their users. Just as is the case with wiretapping, network monitoring does not exist in a legal vacuum: site operators are typically limited in the way the obtained data may be processed

and stored. Generally, one has to differentiate between the legal environment governing the procedures inside corporate networks (affecting the relationship between employer and employee), and those in place to protect the privacy of citizens (in their role of customers of service providers).

2.5.1 Corporate Law

While details vary strongly from country to country (and possibly from state to state), corporate law generally provides employers with the possibility to establish acceptable use policies that the employees have to comply with, and allow monitoring and filtering tools to enforce this compliance.

In Germany, employers may monitor employee communication only after explicit notification, or in case of substantial evidence of resource abuse. Private use of the employer-provided Internet connection is not sufficient reason for terminating the work contract without prior warning, or if comprising less than 100 hours per year (§626 BGB).

In the United Kingdom, the situation is essentially similar. The Regulation of Investigatory Powers Act (RIPA) of 2000 [117] states that employers may only conduct monitoring if both parties have consented, or if the monitoring is required to conduct the employer's business. Beyond that, the Data Protection Act (DPA) [118], most recently revised in 1998, strengthens the employee's position by requiring the employer to assess the impact of the monitoring activity on the user's privacy, and renders monitoring potentially illegal under certain circumstance even after employee consent.

RIPA

DPA

In the United States, the employee's position is considerably weaker, as monitoring can be perfomed even without explicit consent by the employee. As of 2005, approximately 75% of employers monitor employee web surfing, 65% block accesses to certain URLs, and around 50% monitor electronic mail [127].

2.5.2 Civil Law

Germany introduced data protection laws as early as 1969, making it the first country in Europe to offer such protection. Today, the Teleservices

Data Protection Act (TDDSG) [22], introduced 1997, enforces protection of private user data in telecommunication networks that offer communication as a service. It does not cover policies inside corporate networks. Users have to be informed about the extent to which personal data are collected, processed, and used. The service provider may only collect personal information without explicit consent by the user as far as required for providing the service, to bill for it, or to enforce legal recourse in case of abuse. Detailed log files may not be kept for more than six months. *TDDSG*

In the United Kingdom, data protection was initially enforced more reluctantly than in Germany. As with corporate law, today's user rights and operator responsibilities are addressed by the Regulation of Investigatory Powers Act Data Protection Act, much resembling the situation in Germany.

In the United States, citizen's rights to protection of private data are generally weaker than in Europe. Individuals usually have to take action to protect their data, for example to prevent credit bureaus from passing on personal information.[3] The Freedom of Information Act (FOIA) [39], most recently amended in 2002, ensures the public the right to access to records held by the U.S. government. Beyond that, there exists no comprehensive law regulating the treatment of private data; however, there is a wealth of acts[4] governing individual protection aspects, such as the Health Insurance Portability and Accountability Act (HIPAA) [40], enacted 1996, or the Children's Online Privacy Protection Act (COPPA) [41], effective as of 2000. Anderson's book [4] provides further detail in particular about the legal differences between Europe and the United States. *FOIA* *HIPAA* *COPPA*

2.5.3 Implications for this Dissertation

There are certainly far more legal regulations in place at the many sites on the Internet that I am able to consider in this dissertation. However, given the examples I have investigated, I believe that the techniques I introduce in this dissertation do not necessarily require additional legal clearances beyond those required to do other, similar kinds of monitoring. In particular, the fully content-based techniques of Chapter 4 help administrators

[3] See `https://www.optoutprescreen.com/opt_form.cgi`.

[4] See `http://www.informationshield.com/usprivacylaws.html`.

understand the range of known and unknown applications in use on their networks, which certainly seems an acceptable monitoring target for corporate networks at the very least. The malware fingerprinting techniques I introduce in Section 5.3 likewise should not pose fundamental problems, due to the use of honeypots as a traffic source [146]. Finally, the volume-based filtering technique I introduce in Section 5.4 only requires access to at most the transport layer, rendering the legal aspects less complicated than in the previous cases, since access to flow content is not required.

2.6 Summary

Detecting malicious activity on today's networks is a challenge that comes down to how well one can contrast the benign against the malicious. Lack of understanding of either side will invariably lead to poor detection performance. Detection mechanisms operate from different vantage points and at different levels of accuracy. Improvements to any detection strategy do not occur in a vacuum; rather, they are driven by a constant arms race: while the defenders improve their monitoring infrastructure, the attackers try to conceal and evade. This battle is here to stay, regardless of what technological improvements the future holds. This evolution is not exclusively governed by technology: legal requirements try to protect the users' rights to data privacy, with details varying strongly from country to country.

In the following chapters I present ways to enhance the contrast between the benign and the malicious. The essential tools for doing so are techniques to perform structural traffic analysis, which I introduce next.

3

Structural Traffic Analysis

"Uh, 'P'. Okay, 'P'. 'Shh-eer...Sher–P. Sher–P. Shirley? P.–'. Oh! The first line's 'P. Sherman'!"
— Dory, in *Finding Nemo*.

3.1 Introduction

In this chapter I present techniques for structural analysis of network traffic. I use the term "structural" to refer explicitly to types of analysis that extract or leverage characteristic properties of traffic flows at the abstraction levels suitable for the analysis. Examples of such properties might be patterns in flow content, counts of transmitted packets, etc. I do *not* mean traffic analysis for purposes such as detecting steganography, defeating anonymity, and generally methods that are strongly statistical in nature (see Section 3.7 for an overview). *"Structure"*

I begin by describing the various types of analysis possible by monitoring network traffic at various abstraction levels, in Section 3.2. Much of the work I present in the rest of the dissertation uses flow content extracted at the application layer. Once operating at this abstraction level, it often is desirable to extract the message dialog of the communicating endpoints. In Section 3.3, I introduce a technique that achieves this. It serves as the basis for a number of sequence alignment algorithms that I introduce in the remainder of this chapter and that will be used in Chapters 4 and 5. Some of the algorithms are directly inspired by applications in bioinformatics, and it is worth investigating similarities and differences encountered when moving these algorithms into the networking domain. This is the topic of Section 3.4. In Section 3.5 I present a number of sequence alignment strategies and discuss their suitability for network traffic. This

includes the design, complexity analysis, and performance evaluation of a novel extension to Jacobson-Vo, an algorithm not previously applied in the networking domain and able to outperform established algorithms drastically. As I have shown in the previous chapter, attack resilience has to be a central concern in the design of network applications. Structural traffic analysis is no exception, therefore I discuss this aspect, and particularly the implications for the algorithms of Section 3.5, in more detail in Section 3.6. I conclude the chapter with a survey of related work in Section 3.7.

3.2 Abstraction Levels for Network Monitoring

Any kind of network-based traffic analysis necessarily involves inspection of the packets observed on the wire. Starting from the raw packets, the analysis can be performed up to varying levels of *depth*. This depth corresponds roughly to the layer in the network layering model at which the analysis is performed; the higher up, the deeper is the analysis, the more *costly* in terms of CPU cycles, and the more *invasive* to the actual content transferred in the flows. The following list proceeds upward through the layers of the OSI network model [173], describing each layer's relevance to network monitoring.

- Physical Layer: the physical layer defines how a monitoring station can tap into the traffic. For shared media this is typically easy since the standard access method is sufficient to observe all traffic. For optical networks the task is complicated by having to split off a fraction of the optical input signal to be fed into the monitoring engine.
- Data Link Layer: here one can perform statistical analysis of elementary attributes of frames passing the location of the monitor, such as the frame frequency, byte size, and particularly inter-arrival times. Flow granularity is typically irrelevant at this level, though MAC addresses can be used to identify LAN-wide endpoints if necessary.
- Network Layer: at this level, analysis leverages per-packet protocol header information, typically to extract IP addresses and focus flow granularity to the host-pair level. Technologies such as network address translation and proxying weaken the value of IP addresses as

a unique host identifiers, an issue I will be discussing further in Section 5.4.

- Transport Layer: this is the lowest layer carrying end-to-end flow information. TCP and UDP port numbers allow further narrowing of flow granularity to individual sessions via its *five-tuple* of originator and recipient IP addresses, port numbers, and IP protocol type.

five-tuple flow specification

- Session, Presentation, and Application Layers: these levels of analysis understand flow content as perceived by the endpoint applications. They require normalisation of the payload depending on the transport-layer protocol used: for TCP flows, this requires flow reassembly to recombine the individual TCP segments of a connection into the data streams transmitted by the source host. For connectionless protocols such as UDP, the application-layer flow semantics depend on the application-layer protocol. Frequently, all packets carrying the same IP address/UDP port quadruple are considered as a single flow, or alternatively, pairs of datagrams are considered a self-contained request-response dialog (as is the case with DNS, for example). Some degree of transformation of the application-layer content may be necessary in order to render the content accessible as individual data flows. For example, SCTP [149] processing would require the extraction of individual flows from the multiplexed channels carried over a single connection. Furthermore, application-level analysis cannot penetrate encryption without external provision of keying material or termination of the encrypted channel at the monitoring station.

flow reassembly

The trade-off between the different depths is one between the level of information provided on the one hand, and computational overhead and state-keeping requirements on the other. The higher the traffic volume on the observed network, the more stringent are the processing requirements [58]. In this dissertation I use analysis techniques at multiple levels: the content analyses performed in Chapters 4 and 5 operate at the application layer, while the one presented later in Section 5.4 can operate at both the link- and network layers.

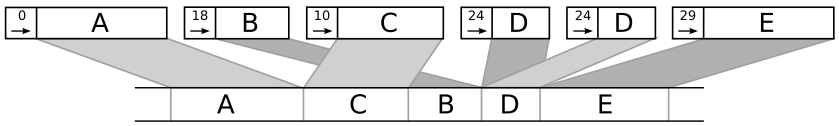


Figure 3.1. Flow reassembly: TCP packets *A* to *E*, shown on top, are observed sequentially on the wire, all belonging to the same connection and going the same direction. *B* and *C* arrive out of order; *D* is duplicated. The packet contents are reassembled into the byte stream shown below, using sequence numbers.

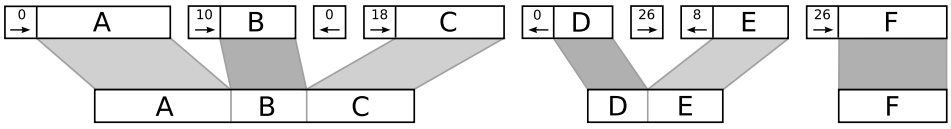


Figure 3.2. Message extraction: payload-carrying TCP packets are reassembled as in flow reassembly, but changes in the direction of payload (packets *D* and *F*) trigger the beginning of new messages. Packets not carrying payload (such as pure ACKs, here shown after packets *B* and *D*) have no effect on message composition.

3.3 Flow Reassembly & Heuristic Message Extraction

Analysis of traffic at the application layer requires recombination of the content of TCP segments (for TCP connections) and UDP datagrams (for UDP flows), as observed at the monitoring location, into the byte sequences that the peering applications are exchanging. This byte sequence becomes the input for flow content analysis algorithms. The recombination process is challenging for various reasons. TCP implements a clear notion of a bidirectional byte stream between endpoints, making *flow reassembly* feasible in principle: TCP sequence numbers, taken from the packet header, are used for pasting together the streams of bytes as transmitted by the originating end host as well as possible. This is illustrated in Figure 3.1. The process is complicated by attackers trying to sneak content past the reassembler or to attack its state management, both of which will be discussed further in Section 3.6.2. UDP, on the other hand, provides no context beyond individual datagrams, thus the only faithful content analysis of UDP flows without knowledge of the application layer protocol can be done at the packet level, treating every datagram as a separate message.

flow reassembly

message extraction

Based on flow reassembly, application-layer *message extraction* takes another step toward recovering the application-layer activity of the end-host processes by attempting to split reassembled flows into semantically consistent messages. As an example, in this model a web browser's HTTP request (possibly consisting of multiple packets) forms a message going from the client to the web server, whose response back to the browser is the second message in the flow. Since TCP provides no hints as to where in the flow message boundaries are to be found, flow splitting has to be done heuristically. The strategy I am using is to split unidirectional flows into messages whenever the monitoring point observes a switch in the direction of transfer of *new* application-layer content. This bidirectional approach is suggested by the causality commonly present between corresponding messages: a server cannot send a reply before the request has been received.[1] In order for this to work, the monitoring point must be able to observe both directions of the communication, which is complicated in principle by the possibility of asymmetric routing but usually doable in practise by careful placement of the monitor. Just as is in flow reassembly, message extraction has to be aware of duplicate transmissions in order to remain synchronised with the endpoints. The process is illustrated in Figure 3.2, and Appendix A.1 shows a Bro policy implementation of message extraction. I will discuss potential shortcomings of the approach in Section 3.6.

Assuming one is not only interested in the first few dozen bytes of a flow, message extraction has two major advantages. First, when comparing corresponding message pairs, it provides a means to detect the beginning of such messages. This would be significantly harder if only operating at flow granularity. Second, it helps to reduce runtime overhead: For any string operation with super-linear runtime in size of its input strings, the cost of repeated computation on individual messages will be smaller than that on the entire flow. Assume a string comparison operation of strings S_1 and S_2 of lengths s_1 and s_2, respectively, requires $O(s_1 s_2)$ runtime. As I will show in the next section, this is frequently the case. If flows consist of m messages, each of length n, then the total cost for message-pair compar-

[1] The degree to which a balance ensues between the volumes of data sent and received by the communication endpoints is itself a useful structural property and will be the topic of Section 5.4.

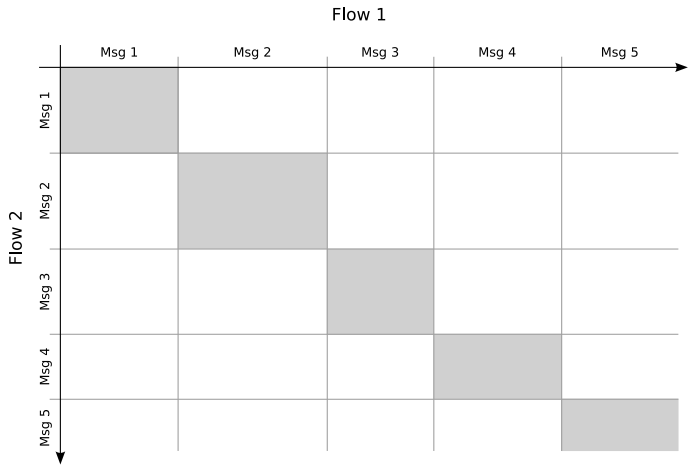


Figure 3.3. Comparison of entire flows vs. corresponding messages. Focusing on corresponding message pairs, shown as shaded areas, yields substantial computational savings.

ison using this algorithm will be $O(m\,n^2)$ as opposed to $O(m^2n^2)$—keeping runtime linear in the number of messages as opposed to quadratic. Figure 3.3 illustrates this.

At present, flow reassembly is considered reliable at line speeds of several gigabits per second when realised in hardware [135], and several hundred megabits per second in software [58]. Message extraction only slightly complicates the reassembly process by tracking the directionality of consecutive novel payload and can thus be expected to achieve comparable throughput rates.

3.4 Sequence Alignment Algorithms

Once operating at the application level, *byte sequences* (or *byte strings*) are a natural abstraction of flow content. Many algorithms are available to operate on structural patterns in such strings. For example, regular expressions encode a *known* content pattern and permit detection of that pattern in flows. They are a cornerstone of misuse detection in network monitoring. Another goal is to *identify* content common to a set of flows. In this dissertation, I use several such *sequence analysis* techniques; more specifi-

cally, I will employ and adapt several *sequence alignment* algorithms. Very generally, these algorithms try to find and align similar regions in the input strings, and are frequently used in the bioinformatics domain.

sequence alignment

3.4.1 Inspiration from Bioinformatics

The field of bioinformatics aims to develop techniques which enable the analysis of DNA sequences. These sequences make up the genome of living organisms and are massively long strings of four different nucleotides: adenine, cytosine, guanine, and thymine. A frequent challenge given DNA sequences is *motif detection*, i.e., the detection of patterns of nucleotides that are of some biological significance. Research on motif detection has benefited the development of many exact as well as approximate string search algorithms as well as more elaborate techniques for inferring second-level properties, for example using Hidden Markov Models [60]. The parallels to network traffic analysis are straightforward: 256 single-byte values replace the four nucleotides, network flows are the equivalent of DNA sequences, and motifs are patterns relevant to the purpose of the traffic analysis. Computation of *common substrings* is a frequent theme in sequence analysis. Chapters 4 and 5 will both be making use of these techniques.

motif detection

3.4.2 Similarities to Biology

Gusfield [68] justifies the importance of sequence analysis techniques in biology stating that

> In biomolecular sequences [...], high sequence similarity usually implies significant functional or structural similarity.

This observation holds true when substituting biomolecular sequences with byte sequences in computer networks. In biology, the sequence alphabet consists of the four nucleotides adenine, cytosine, guanine, and thymine, comprising a life form's DNA.[2] In network flows, the obvious equivalent to nucleotides is the byte, yielding an alphabet of 256 possi-

[2]More precisely, the genetic code for creating proteins based on the DNA sequence operates at the granularity of *codons*, consisting of sequences of three nucleotides at a time, yielding $4^3 = 64$ different codons. Since the algorithms operating on DNS sequences however typically operate at the nucleotide level, I focus on the lower level of abstraction.

ble characters. The parts of the DNA sequence affecting an organism's functions are the ones carrying genes, while the rest is commonly referred to as "junk DNA". In network flows, arguably all parts of the network flow serve some purpose, though the functioning of the application-layer protocols involved critically depends on certain regions of the flow carrying the correct "genes", protocol-intrinsic strings such as `GET` and `POST` for HTTP, `USER` and `PASS` for FTP, etc. If one wanted to stretch the analogy further, one could call the payload carried by an application-layer protocol the junk DNA of the flow, as it serves no active function in the protocol state machine.

As I will show in Chapter 4, high sequence similarity among key regions in multiple flows strongly indicates the presence of the same application-layer protocols. Gusfield goes on to point out that the reverse implication, inferring sequence from function, is not necessarily true in bioinformatics. This certainly is a similarly valid statement for network flows: similar protocol functionality does not necessarily imply similar sequences. For example, file transfers clearly are implemented in many different ways, and both FTP and HTTP, among many others, can serve this purpose.

3.4.3 Differences from Biology

The similarity to the DNA setting is striking, but has its limits. Three differences are of immediate practical significance. First, the desirable timescales for operating on the sequences are different. While off-line processing of network traces clearly has its uses, applications such as the ones relating to containment of Internet epidemics, will require very fast *on-line* processing of input sequences. Such a requirement does not exist in bioinformatics, where the need for fast algorithms is mostly motivated by processing very large sequences. Note however that while sequences analysed in bioinformatics can get large, that is *not* necessarily the case: the average length of sequences stored in the GenBank database has remained close to 1,000 nucleotide pairs [23, 63], a length realistic for sequence analysis problem settings when operating on network flows as well [172, 84]. Second, there is a strong need for fully automated and precise operation in the networking domain, whereas in bioinformatics heuristic approaches are often used to get an initial approximation of an alignment that may

then be optimised manually [73, 59]. Third, he difference in input alphabet sizes means that sequence alignment algorithms can operate on more fine-grained input and more flexible alignment scoring schemes are conceivable. At the same time, however, per-possible-character statistics will require more state, and the overall complexity of the alignment models may increase as well depending on their sophistication.

Other differences are more abstract, but nevertheless affect the effectiveness of the algorithms when applied to network flows. The most significant is the observation that nature at most introduces limited *random* mutation, while the network security domain has to deal with a potentially *malicious* modification and obfuscation by a conscious adversary. Therefore, the algorithms have to be designed with the forethought of being actively attacked and evaded. Furthermore, the notion of random mutation and its modelling in approximate sequence alignment algorithms is only of limited applicability in network traffic: flow content does not generally undergo random mutation, rather, the protocols evolve using more or less well-specified implementations and payloads are highly variable by definition. See Section 3.7 for work that explicitly models protocol and content evolution.

3.5 String Alignment Models for Network Traffic

Flow reassembly and message extraction provide byte strings suitable as input for alignment algorithms. As I will show in the remainder of the dissertation, extracting alignments from network flows enables a number of applications relevant to network security monitoring.

An alignment generally describes which parts of a set of input strings are found in all of the input strings, and which parts vary. In textual protocols, these might be common keywords (such as 'HTTP' or 'PASS'); in binary protocols commonalities exist when fields in different flows have the same values. *Precise* alignments find exact commonalities among the compared strings, while *approximate* alignments allow for limited deviation between alignments of sequences. These deviations are expressed through the notion of *edit operations* required to transform one input string into the

precise vs. approximate alignment

other. The editing operations for non-matching string elements are insertions, deletions, and substitutions. To compute a distance metric between sets of edits, a penalty function assigns each editing operation a particular score. An alignment then is the resulting pairing of individual elements of a string given a set of edit operations, and the lower the accumulative penalty for transforming one input string into the other, the better the alignment found.

global vs. local alignment

Two strings can be aligned *globally* or *locally*. Global alignment implies the underlying assumption that two strings essentially line up well and that only minor misalignments have to be figured out. In contrast, local alignment assumes no inherent similarity between strings in general and focuses on finding islands of similarity. As an example, the following is a global alignment of the two strings 'SECURITY' and 'SURE THING' ('✓' is a match, '×' a mismatch, '+' an insertion, and '−' a deletion):

```
SECUR____ITY
S__URE THING
✓−−✓✓++++✓××
```

Below is a local alignment of 'A REASSURING FACT' to 'NO SURE THING'. Since the precise nature of the *gaps* between similar substrings is less important now, I only highlight the matches, though often edit operations might still be important within the aligned substrings:[3]

```
A REASSUR____ING FACT
   NO SURE THING
      ✓✓✓      ✓✓✓✓
```

The classical approach to computing alignments is the algorithm proposed by Smith and Waterman [141]. Different variants can compute local or global alignments; when the latter is done, the algorithm is commonly referred to as Needleman-Wunsch [112].[4] The difference lies mainly in the scoring: local alignment maximises an alignment score, while global alignment minimises the edit distance.

[3]Note that whitespace is included in the alignment. Where the existence of whitespace is important and not typographically obvious, I indicate it with '␣' symbols.

[4]Gusfield [68] points out that Needleman and Wunsch only discussed the global alignment problem, but proposed a different (and slower) algorithm.

3.5.1 Longest Common Substrings

A simple alignment procedure is to find a string common to both input strings and to maximise the length of that substring. This is referred to as the longest common substring problem in the literature, however I will refer to it as the *longest common region* (LCR), to differentiate the abbreviation from longest common subsequences, which I introduce in the next section.

longest common region (LCR)

Given strings S_1 and S_2 of lengths s_1 and s_2, respectively, the LCR problem can be solved in $O(s_1 + s_2)$ using suffix trees. Gusfield defines suffix trees as follows [68]:

suffix trees

> A suffix tree T for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labelled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i. That is, it spells out $S[i..m]$.

Furthermore, to ensure that each suffix is unique and thus cannot coincide with any prefix of another suffix, a non-alphabet stop character is appended to the input strings prior to suffix tree construction. Several algorithms exist for building a suffix tree in time linear to the input string size [167, 101]. I have chosen Ukkonen's algorithm [154] for my experiments, which achieves the linear time bound by building the suffix tree incrementally, carefully leveraging existing structure, and avoiding per-character scanning operations whenever possible. My implementation comprises roughly 1200 lines of C and is publically available as a stand-alone library, `libstree`.[5] The fact that it generalises the notion of a string has made it popular and lead to its use in other work [94].

Once a single-string suffix tree can be built in linear time, the extension to multiple strings is straightforward: each edge in the tree is labelled with the numbers of the strings that contribute this edge. At this point, the LCR problem is equivalent to finding the longest path from the root of the suffix tree that has been contributed by all input strings. The algorithm

[5] See `http://www.cl.cam.ac.uk/~cpk25/libstree` for details.

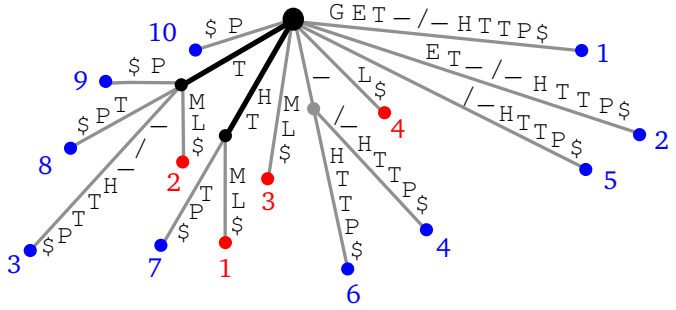


Figure 3.4. The suffix tree for strings 'GET / HTTP' and 'HTML'. End-of-string markers are shown as $ symbols. The blue leafs and leaf labels correspond to the first input string, the red ones to the second. The prefixes of suffixes common to both strings are in bold. The LCR is 'HT', since it is the longest accumulative path label common to both strings.

effectively finds the longest common prefix of a set of suffixes comprising all input strings. Figure 3.4 illustrates the suffix tree structure in the LCR computation on two input strings.

The advantage of this LCR algorithm is that it is fast and can easily be extended to multiple input strings; its disadvantage is that it returns only a single common substring. I will present an LCR application in Section 5.3.

3.5.2 Longest Common Subsequences

The natural extension of LCRs is the computation of alignments consisting of multiple common substrings. Such an alignment is called the *longest common subsequence* (LCS) of the input sequences. A common subsequence is a sequence of common substrings of two strings, possibly together with the offsets into the two strings at which the commonalities occur; a longest common subsequence is the common subsequence of maximum cumulative length. Consider the following HTTP URL request strings:

longest common subsequence (LCS)

```
'GET / HTTP/1.1'
'GET /cgi/HT/TP/cvs?ver=1.1 HTTP/1.0'
```

Their LCSs have length 13, but note the plural. There are a number of LCSs with that maximum length, such as the following:

```
'GET /' - '␣HTTP/1.'
'GET /' - 'HT' - 'TP/' - '1.1'
'GET␣' - '/HT' - 'TP/' - '1.1'
'GET␣' - '/' - '␣HTTP/1.'
'GET␣' - '/' - 'HT' - 'TP/' - '1.1'
```

The key difference is that the LCSs vary in the *number of gaps*: the first one has just one gap while the last one contains 4. Depending on the goals of the LCS computation we could leave it up to the dynamics of the algorithm which version we obtain, or ensure the computation of a particular variant. My point here is thus not to show a simplistic example of the iconic HTTP 'GET' string working flawlessly, but rather the opposite: even for such a classic example, the algorithm can be mislead easily without attention to the details.

gap minimisation

The algorithms I present in the next sections compute an LCS with the *minimum number of gaps* and *longest possible substrings*. I argue that in the networking context, this yields the most meaningful results. In context of the HTTP protocol, for example, the LCS 'GET /' - '␣HTTP/1.' is more meaningful than 'GET ' - '/' - 'HT' - 'TP/' - '1.1', since the former captures the semantic meaning of the alignment (the HTTP request method and the protocol version), while the latter contains disjointed substrings with potentially higher probability of individual occurrence, such as '/'). Furthermore, short common substrings can lead to incorrect results if, when given an LCS, we ask for substrings of at least a minimum number of characters. In the above example, if the algorithm produces LCS 'GET ' - '/' - 'HT' - 'TP/' - '1.1' and the minimum string length requirement is five characters, no result would be found even though an LCS fulfilling that length restriction does in fact exist, namely the one minimising the number of gaps.

I next present two LCS-computing algorithms that minimise gaps while maximising common substring length, and compare their relative performances on a wide range of network traffic.

	R	E	A	S	S	U	R	I	N	G
N	←0	↖0	↖0	↖0	↖0	↖0	↖0	↖0	↖**1**	←1
O	↑0	↑0	↑0	↑0	↑0	↑0	↑0	↑0	↑1	↑1
	↑0	↑0	↑0	↑0	↑0	↑0	↑0	↑0	↑1	↑1
S	↑0	↑0	↑0	↖**1**	↖**1**	←1	←1	←1	↑1	↑1
U	↑0	↑0	↑0	↑1	↑1	↖**101**	←101	←101	←101	←101
R	←**1**	←1	←1	↑1	↑1	↑101	↖**201**	←201	←201	←201
E	↑1	↖**101**	←101	←101	←101	↑101	↑201	↑201	↑201	↑201
	↑1	↑101	↑101	↑101	↑101	↑101	↑201	↑201	↑201	↑201
T	↑1	↑101	↑101	↑101	↑101	↑101	↑201	↑201	↑201	↑201
H	↑1	↑101	↑101	↑101	↑101	↑101	↑201	↑201	↑201	↑201
I	↑1	↑101	↑101	↑101	↑101	↑101	↑201	↖**202**	←202	←202
N	↑1	↑101	↑101	↑101	↑101	↑101	↑201	↑202	↖**302**	←302
G	↑1	↑101	↑101	↑101	↑101	↑101	↑201	↑202	↑302	↖**402**

Figure 3.5. A Smith-Waterman matrix, computing the longest-common subsequence of strings 'REASSURING' and 'NO SURE THING'. The best LCS is 'SUR' - 'ING', indicated by cells shaded in dark grey. Common substrings can be found along diagonals; trace-back paths not following a diagonal indicate gaps between the common substrings. Notice how suboptimal possible alignments such as 'RE' are not included since the back-pointer path from the last cell walks past them.

3.5.3 Smith-Waterman: Dynamic Programming

Given a pair of input strings S_1 and S_2 of lengths s_1 and s_2, respectively, Smith-Waterman uses dynamic programming to compute the LCS incrementally, requiring $O(s_1 s_2)$ space and time.[6] The algorithm operates by filling a matrix row-by-row, recording in each cell the best alignment of the prefixes of S_1 and S_2 up to the cell's row/column indices by picking an *edit operation* on the pair of characters at the current row/column. These operations can (*i*) skip characters of either string, (*ii*) align the characters directly, or (*iii*) accept mismatching characters via substitution. Each operation is assigned a cost/score, and the best resulting alignment is the one with the highest score (for local alignment) or lowest cost (for global alignment). The alignment can be extracted by walking backward through the matrix, starting in the bottom-right corner, following the decision taken at each cell.

edit operation

In my implementation I use a scoring function for computing alignments; my affine alignment scoring uses an *alignment start* score of 1 for every

affine alignment scoring

[6] Fast implementations are feasible by leveraging FPGAs or GPUs [115, 158]. The space requirement can be pushed down to $O(s)$ with $s = \min(s_1, s_2)$ while at most doubling worst case time [68].

first matching character after a gap and strongly enforces continued alignments using an *alignment growth* score of 100 for further matching characters. Gaps are penalised relatively by leaving the sequence score unaffected. Similar results could be obtained by penalising gaps more strongly, relative to a more modest encouragement of aligned regions. As mentioned earlier, this model ignores the possibility of per-character-pair byte substitution scores by computing exact common subsequences interleaved with gap regions. Not only does this simplify the approach, it also avoids the need to develop a robust scoring scheme for character distributions. The derivation of general yet robust substitution scores is highly problem-specific and generally non-trivial [60, 68]. It is a topic for future research how to apply the concept to sequence alignments of network flows. Figure 3.5 shows an example of Smith-Waterman with the scoring scheme just described.

Smith-Waterman's strength lies in its flexibility: given the completeness of information about S_1 and S_2 stored in the completed matrix, variations are readily implemented. For example, it is easy to adapt the implementation to return *all common substrings* (ACS) of at least a given minimum length still within $O(s_1 s_2)$, since we can register separately all common substrings as they exceed the minimum length, and continuously check whether growing common substrings have already been registered. I will show the importance of this particular variant in Section 3.6 and use it later on in Section 4.3.

all common substrings (ACS)

3.5.4 Jacobson-Vo: Combinatorial Reduction

Jacobson and Vo [78] and Pevzner and Waterman [122] independently presented a method for computing LCSs that works fundamentally differently from Smith-Waterman, and potentially faster. I summarise its operation in this section and show that unfortunately Jacobson-Vo has a shortcoming: the LCSs it produces neither necessarily minimise the number of gaps, nor maximise common substrings. As outlined in Section 3.5.2, these goals are highly desirable, and Section 3.5.5 will discuss a way to overcome this limitation and analyse its effect on performance.

Jacobson-Vo reduces a related combinatorial problem for which there is a potentially faster solution than $O(s_1 s_2)$ to the LCS problem. This combi-

longest increasing subsequence (LIS)

natorial problem is the identification of a longest increasing subsequence (LIS) in a sequence of numbers. I will first present the workings of the algorithm pragmatically and later summarise the reasoning behind each step. Consider the following input strings S_1 and S_2 of lengths $s_1 = 10$ and $s_2 = 17$, respectively:

```
'GET / HTTP'
'GET /a/a.HTM HTTP'
```

The algorithm is based on the observation that an LIS has a one-to-one correspondence with an LCS if the sequence of numbers is produced from the input strings in the following fashion: iterating over the characters in S_1, one lists once per occurring character all indices in S_2 at which that character occurs, in descending order. This yields:

```
G  →  0              /  →  6 4
E  →  1              H  →  13 9
T  →  15 14 10 2     P  →  16
_  →  12 3
```

These character occurrence lists are then concatenated into a numerical sequence Π of length π. For S_1 and S_2 the beginning of Π looks as follows (dots indicate occurrence list merge points):

```
0 · 1 · 15 14 10 2 · 12 3 · 6 4 · 12 3 · 13 9 · ...
```

Given Π, the next step is greedy extraction of a *cover* of Π. A cover is a set of subsequences with *non-increasing* indices that in combination comprise all numbers in Π. One can perform this extraction in a tabular fashion by building up each subsequence in one column into a *subsequence table*. Let $\mathcal{S}_n$ be the nth non-increasing subsequence. An arbitrary element in $\mathcal{S}_i$ is denoted e_i, and $I_{e_i}^{S_1}$ and $I_{e_i}^{S_2}$ are e_i's indices in S_1 and S_2, respectively. Context will make it clear which table is being referred to.

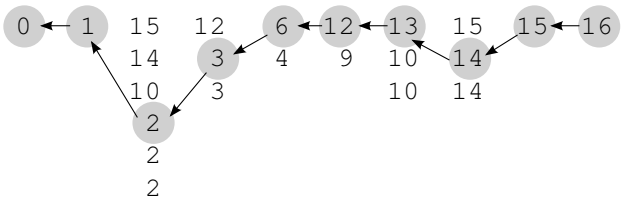
subsequence table

Iterating over the elements of Π, one selects for each element the leftmost subsequence (i.e., column in the table) that the element can extend. Extension is possible whenever the last number in a sequence is larger than or equal to the new element. If no subsequence fulfils this requirement, a new one is added to the table. For the example above, the resulting subsequence table is as follows:

To extract an LCS, first an arbitrary element in the last subsequence is se-

```
0   1  15  12   6  12  13  15  15  16
      14   3   4   9  10  14
      10   3           10  14
       2
       2
       2
```

lected. Afterward, the remaining subsequences are scanned downward in right-to-left order, selecting the first element e_i in each $\mathcal{S}_i$ for which $I_{e_i}^{S_2} < I_{e_{i+1}}^{S_2}$, where e_{i+1} is the element chosen in $\mathcal{S}_{i+1}$. The resulting sequence of S_2 indices is an LCS of S_1 and S_2:



Using this procedure, the resulting LCS is '`GET␣`' - '`/`' - '`␣HTTP`'. It requires some thought to see how this construction computes LCSs. The original papers and Gusfield present the required lemmata in detail, so I settle for summarising them here as follows:

1. The existence of a cover C of Π consisting of c non-increasing subsequences and of an *increasing* subsequence I, also of length c, that selects exactly one element from each of the non-increasing subsequences mean that C is a smallest cover and I is a longest increasing subsequence. This follows from the fact that the indices of each subsequence are non-increasing and an increasing subsequence can thus contain at most one element from each non-increasing subsequence.[7]

2. The nature of the cover construction guarantees that for every element e_i in $\mathcal{S}_i$, there is an element e_{i-1} in $\mathcal{S}_{i-1}$ such that $I_{e_{i-1}}^{S_2} < I_{e_i}^{S_2}$, forms a two-element increasing subsequence. Since an increasing subsequence of length c can thus be constructed by selecting exactly one element from each resulting subsequence, the number of sub-

[7]For all practical purposes the role of the increasing subsequence is that of the LCS, so in order to avoid confusion with the non-increasing subsequences that make up the subsequence table, I will refer to the increasing subsequence as LCS whenever possible, and mean non-increasing subsequences whenever I just speak of a subsequence.

sequences equals the length of the increasing subsequence, and the greedily constructed cover is thus a smallest cover.

3. Note that each element in Π not only specifies an index in S_2, but also one in S_1 if we keep track of the S_1 index that contributed an occurrence list to Π. Consider the beginning of Π shown above for S_1 and S_2, this time also tracking the corresponding S_1 indices in the lower numbers:

```
0   .  1   .  15 14 10 2   .  12 3   .  6 4   .  12 3   .  13 9   .  ...
0      1         2           3        4      5       6
```

Thus every increasing subsequence of length l in Π specifies exactly one common subsequence among S_1 and S_2 of length l, and each LIS of Π corresponds to an LCS of S_1 and S_2.

To estimate the runtime complexity of this procedure, observe that the last numbers of the subsequences are sorted in increasing order at all times when scanning the table left-to-right. We can thus find the correct column for insertion via binary search. Let S_1 be the shorter of the two strings, without loss of generality. Since there can never be more than s_1 sequences in the table and we insert π elements in total, this algorithm runs in $O(\pi \log s_1)$.

3.5.5 Improving Jacobson-Vo: Targeted LCS Selection

Note that for S_1 and S_2, standard Jacobson-Vo yields '`GET␣`' - '`/`' - '`␣HTTP`', an LCS that violates the goals of gap minimisation and substring maximisation. I will now extend the algorithm in a number steps to overcome this limitation, borrowing several concepts from Smith-Waterman: I introduce dynamic programming to Jacobson-Vo to track incrementally the LCS that yields the smallest number of gaps and longest-possible substrings throughout the computation, and collect the optimal LCS via backpointer traversal. As I will show, these extensions render Jacobson-Vo gap-minimising and substring-maximising, while retaining the same algorithmic complexity as the original algorithm. Finally, I compare the performance of Smith-Waterman to that of unmodified Jacobson-Vo as well as the extended version when applied to a representative spectrum of network traffic, and show that that the extended Jacobson-Vo algorithm in-

volves negligible runtime overhead compared to the original algorithm while running up to almost 60 times faster than Smith-Waterman.

3.5.5.1 *Path Selection through Dynamic Programming*

The unmodified Jacobson-Vo algorithm does not consider possible alternatives in the selection of each subsequence's LCS member. The first step therefore is to consider the choices we have whenever an LCS member in $\mathcal{S}_{i-1}$ is selected after having selected one in $\mathcal{S}_i$. Adding the S_1 indices of each element in Π to the subsequence table (in small type), we obtain the following:

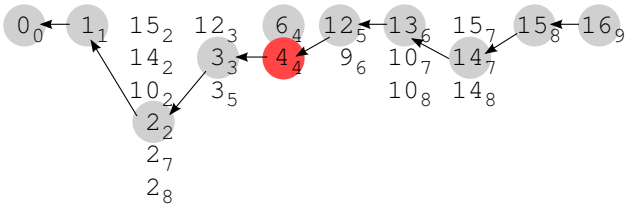
$$
\begin{array}{cccccccccc}
0_0 & 1_1 & 15_2 & 12_3 & 6_4 & 12_5 & 13_6 & 15_7 & 15_8 & 16_9 \\
 & & 14_2 & 3_3 & 4_4 & 9_6 & 10_7 & 14_7 & & \\
 & & 10_2 & 3_5 & & & 10_8 & 14_8 & & \\
 & & 2_2 & & & & & & & \\
 & & 2_7 & & & & & & & \\
 & & 2_8 & & & & & & &
\end{array}
$$

Observe that while the S_2 indices are non-increasing in each subsequence when reading top-down, the S_1 indices are non-*decreasing*. This follows from the mechanics of the algorithm—later insertions into the table appear further down in the subsequences and are made using elements further to the right in Π, and those elements have equal or larger S_1 indices. Assume now that we have just chosen an element e_{i+1} in $\mathcal{S}_{i+1}$. Since every element in $\mathcal{S}_i$ has least one element in $\mathcal{S}_{i-1}$ that can be chosen as its predecessor, we can pick any element e_i in $\mathcal{S}_i$ as LCS member subject to the condition that $I_{e_i}^{S_1} < I_{e_{i+1}}^{S_1}$ and $I_{e_i}^{S_2} < I_{e_{i+1}}^{S_2}$ since only then does e_i appear before e_{i+1} in both S_1 and S_2. Given the opposite growth directions of the indices in each subsequence in the table, this means that for each e_{i+1} there exists a *window* of possible predecessors in subsequence i, and, by symmetry, for each e_i there exists a window of possible successors in subsequence $i+1$. More formally, the sets of elements $W_p(e_i)$ in the predecessor window of e_i and $W_s(e_i)$ in its successor window are defined as

predecessor/ successor window

$$
\begin{aligned}
W_p(e_i) &= \left\{ e_{i-1} \in \mathcal{S}_{i-1} : \;\; I_{e_{i-1}}^{S_1} < I_{e_i}^{S_1} \wedge I_{e_{i-1}}^{S_2} < I_{e_i}^{S_2} \right\} \\
W_s(e_i) &= \left\{ e_{i+1} \in \mathcal{S}_{i+1} : \;\; I_{e_{i+1}}^{S_1} > I_{e_i}^{S_1} \wedge I_{e_{i+1}}^{S_2} > I_{e_i}^{S_2} \right\}
\end{aligned}
$$

Returning to our running example, we see that $W_p(12_5) = \{6_4, 4_4\}$, i.e., when choosing the predecessor of element 12_5 (up to which there are no alternatives since window size is always 1) we can choose between 6_4 and 4_4. If we choose the latter, we end up with the desired LCS 'GET /' - '␣HTTP/1.':



Thus the goal is to use the limited freedom in selecting LCS members to minimise gap counts and maximise substring lengths in the resulting LCS. By tracking those properties incrementally on all possible paths through the table and identifying the path with least gaps and longest substrings, the algorithm will compute the desired LCS. This LCS is collected by traversing the table left-to-right from the element in the last subsequence with the highest score, using back-pointers. Note that the search still starts in the last subsequence, since scanning right-to-left has the benefit of eliminating more elements from consideration. As in the original approach, all elements of the last subsequence are potential starting points. The core strategy is to perform a *parallel downward scan* of pairs of subsequences adjacent in the table. If the table contains n subsequences, the first scan uses $\mathcal{S}_{n-1}$ and $\mathcal{S}_n$, the second $\mathcal{S}_{n-2}$ and $\mathcal{S}_{n-1}$, etc., until eventually $\mathcal{S}_1$ and $\mathcal{S}_2$ are reached.[8]

parallel scanning

Assume the scan currently examines $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$. The scan considers all elements in $\mathcal{S}_i$ in top-down order that have a non-empty window in $\mathcal{S}_{i+1}$, ignoring the ones at the beginning of $\mathcal{S}_i$ with too high an S_2 index as well as those at the end of the subsequence with too high an S_1 index. The elements linked by back-pointers thus form a *corridor* through the subsequence table, and the upper boundary of the corridor is the LCS selected by the original Jacobson-Vo algorithm. This is illustrated in Figure 3.6.

As the scan proceeds from one $\mathcal{S}_i$ element to the next, the successor win-

[8]Single-column tables don't permit this approach, however their occurrence means that the LCSs consist only of a single character and any member of the sole column will do.

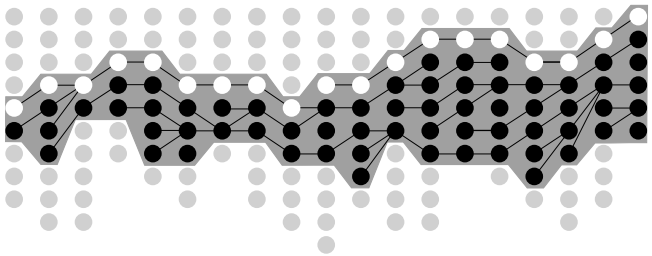


Figure 3.6. Corridor of linked LCS elements (shaded in grey background) through an idealised subsequence table. The elements along the upper boundary of the corridor (in white) form the LCS selected by the original Jacobson-Vo algorithm.

dow $W_s(e_i)$ moves down $\mathcal{S}_{i+1}$. Let the currently considered element in subsequence i be e_i. The idea is to compute alignment scores, akin to Smith-Waterman, incrementally for all LCSs as they are considered, tracking the best-scoring one. As with Smith-Waterman, it depends on the alignment model whether "best" means maximisation (of alignment similarity) or minimisation (of edit distances). Alignment scores can penalise gaps and encourage long common substrings, but also realise other alignment policies. By prefixing e_i to the partial LCSs starting with the elements in $W_s(e_i)$ and ending in $\mathcal{S}_n$, e_i's score can be computed for each LCS depending on whether e_i introduces a gap, starts a common substring, or extends one. The best-scoring element $e^*_{i+1} \in W_s(e_i)$ is remembered by setting e_i's backpointer to e^*_{i+1}, and storing the corresponding best score in e_i.

In order to be able to score common substrings differently from gaps, the algorithm must be able to track common substrings as they occur. Common substrings consisting of at least two characters exist whenever two LCS elements e_i and e_{i+1} have the property that $I^{S_1}_{e_i} + 1 = I^{S_1}_{e_{i+1}}$ and $I^{S_2}_{e_i} + 1 = I^{S_2}_{e_{i+1}}$. To notice when this is the case, the algorithm tracks the the element inside $W_s(e_i)$ whose S_1 and S_2 indices are as close as possible to, but strictly larger than, e_1's. Let this element be called e_i's *neighbour*, denoted e^n_i. A *direct neighbour* is a neighbour e^n_i for which $I^{S_1}_{e_i} + 1 = I^{S_1}_{e^n_i}$ and $I^{S_2}_{e_i} + 1 = I^{S_2}_{e^n_i}$, i.e., one that e_i can extend as a common substring. To formalise the neighbour definition, let the distance $\mathcal{D}$ of subsequence members e_i and e_{i+1} be defined as $\mathcal{D}(e_i, e_{i+1}) = (I^{S_1}_{e_{i+1}} - I^{S_1}_{e_i}) + (I^{S_2}_{e_{i+1}} - I^{S_2}_{e_i})$. Then

(direct) LCS neighbour

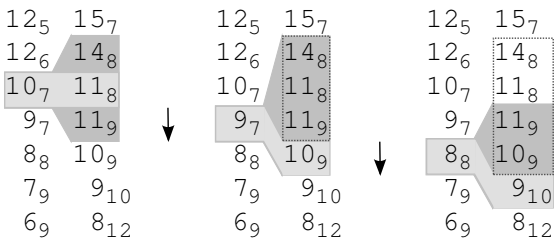


Figure 3.7. Parallel subsequence scanning with sliding windows. As the iteration proceeds over the left sequence's elements 10_7, 9_7, and 8_8, the window of possible successor elements in the right subsequence slides downward, updating the top and bottom boundaries of the window accordingly. The dotted border indicates the previous window. Along with the window boundaries, the current element's neighbour (shown with lighter background) moves down as well: while 10_7 can extend the substring ending at 11_8, for 9_7 and 8_8 the introduction of a gap is unavoidable. (The string indices shown are hypothetical and not related to the running example.)

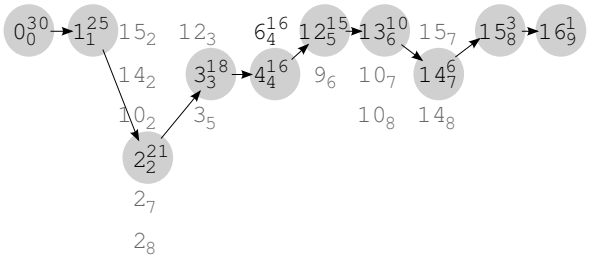
e_i^n is defined as follows:

$$e_i^n \;=\; e \in W_s(e_i): \;\; I_e^{S_1} > I_{e_i}^{S_1} \;\wedge\; I_e^{S_2} > I_{e_i}^{S_2} \;\wedge\; \mathcal{D}(e_i, e) = \min_{e' \in W_s(e_i)} \left[\mathcal{D}(e_i, e')\right]$$

The neighbour always resides within W_s, since it is a legitimate successor of e_i, all of which are by definition contained in W_s. As the elements inside e_i's window are considered, a direct neighbour can be scored in a way that ensures extension of an existing common substring as opposed to introducing a gap. Figure 3.7 illustrates sliding windows with neighbour tracking.

The introduction of alignment scoring to the algorithm adds significant flexibility to the algorithm, since many different scoring models are conceivable. Below I show the subsequence table for the running example, with each visited element's alignment score in the top right corner, showing previous pointers where set, and using a scoring scheme that quadratically favours longer common substrings (by adding the length of the common substring to the score, for each substring character) while linearly increasing the score for gaps:

The subsequence elements in grey are outside of the corridor and not considered for back-pointer linking.



3.5.5.2 *Overcoming Greedy Substring Extension*

The modified Jacobson-Vo algorithm is now gap-minimising if a scoring scheme favouring common substrings over gaps is used, because such a scoring scheme will never introduce a gap if it can extend a common substring. Whichever path has the least amount of gaps globally will be the one with the largest overall score. One problem remains, however: the greediness of common substring extension means that a common sequence will *always* be extended when possible due to its locally higher score, even when it would be beneficial to stop a substring and begin a new one. This situation occurs when one common substring's suffix is a later common substring's prefix and is illustrated in Figure 3.8.

global & local score tracking

Thankfully the problem is easy to fix: in addition to tracking with every element e_i the *globally* best score it obtains by linking with the best element in $\mathcal{S}_{i+1}$, we now also track the *local* score the element has when following the common substring it is part of through to the end. If this common substring turns out to be longer than the one it overlaps with, the local score will eventually exceed the global one and take its stead. What's left to do is to adjust the back-pointer that cuts off the tail of the longer substring back into the substring.

3.5.5.3 *Complexity Analysis*

The extended Jacobson-Vo is identical to the original one as far as construction of the subsequence table is concerned. After that, it differs in the following ways:

- The parallel scanning phase for setting back-pointers does not exist in the original version.
- In the original version, the LCS is collected by potentially scanning

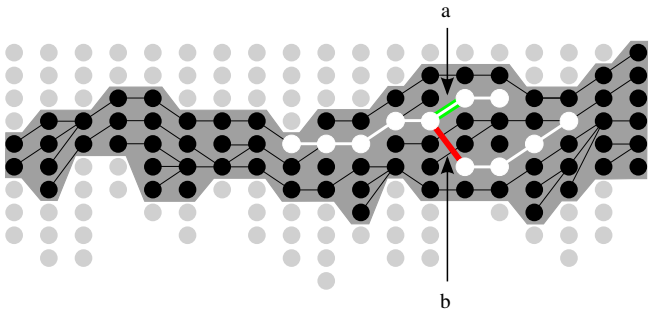


Figure 3.8. Overlap of an longer, earlier common substring (in white, top) with a shorter, later one (bottom). For maximising common substrings, the algorithm must not select link (b), as it would cut off the tail of the longer common substring. Instead, it should follow the longer common substring to its end by using link (a).

all members of the table at most once, while in the extended version LCS collection is a matter of iterating over just the optimal LCS's members exactly once.

It is clear that the extended variant's runtime complexity cannot beat the original algorithm's $O(\pi \log s_1)$, since the extended variant does additional work. The question is how costly the extension of the algorithm is. The parallel scanning phase considers every element in the left subsequence at most once, implying $O(\pi)$ additional cost. Naïvely, for each element e_i in $\mathcal{S}_i$, every element in $W_s(e_i)$ must be considered. This implies a non-constant amount of additional work per Π element which would certainly affect the overall runtime complexity negatively.

The following observation comes to the rescue: unless e_i's neighbour in $W_s(e_i)$ is direct, *all* elements in the window are going to introduce gaps. In this case, and unless our alignment model scores different gaps differently, there is no reason to consider each window member. We only need to know which window member's score is *best*, and update that score according to our scoring schema. This trick reduces the amount of work needed per e_i element to a constant, since we only need to track the best-scoring node in the window and e_i's neighbour. Three pointers suffice, and since the parallel scanning phase only slides the window over each subsequence once, each of those pointers will similarly visit each member of Π at most once.

At this point, the runtime complexity depends on the effort required to track the best-scoring node in the window. By storing the window elements in a priority queue, we can access the best-scoring element in constant time. More precisely, it is not necessary to store all window members, but only one representative of each different *score* present in the window. Assume the window members have n different scores in total, and the priority queue thus contains n elements. As the window moves downward over a subsequence, new elements are inserted into the priority queue as the low window boundary advances. Using a heap, this can be done in $O(\log n)$. At the same time, existing elements need to be removed from the priority queue whenever a member falls out of the window as the top boundary advances. Removal can likewise be done in $O(\log n)$. Access to the element that is to be removed can be gained in constant time if we maintain an array that maps the scores present in the priority queue to its members.

To estimate the maximum size n of the priority queue, we need to bound the maximum size that a subsequence of Π can obtain. Note that a single occurrence list can exist in a subsequence at most once in its entirety, and an occurrence list can be at most of size s_2. Beyond that, a subsequence can only grow by repeating the bottom-most index repeatedly, which can occur at most s_1 times. Therefore size of a subsequence is bounded from above by $s_1 + s_2$.

We can now summarise the runtime complexity of the extended Jacobson-Vo algorithm. As in the original approach, we insert each member of Π into the subsequence table using binary search, requiring $O(\pi \log s_1)$. The parallel scanning phase visits each element in Π at most once in the left subsequence, while each element in the right subsequence is at most once inserted into the priority queue and removed from it, which takes at most $O\left(\log(s_1 + s_2)\right)$. Combining subsequence table construction and parallel scanning phase, we obtain $O\left(\pi\left(\log s_1 + \log(s_1 + s_2)\right)\right)$. Since normally we can assume $s_1 \approx s_2$ and thus $O(s_1 + s_2) = O(s_1)$, thus $O(2\pi \log s_1) = O(\pi \log s_1)$.

Remarkably, extending Jacobson-Vo to target gap-minimising and substring-maximising LCSs does not hurt the runtime complexity bound, making only modest assumptions about the scoring schema, namely uniform

gap penalties.

3.5.5.4 *Practical Speed-ups*

Two more observations help improve performance in practise. They are applicable to both the original and extended algorithm, and are both related to finding the subsequence to which to append elements of Π when constructing the subsequence table.

First, note that subsequent insertions into Π of members of an occurrence list never occur further to the right than earlier insertions, i.e., the column indices of the subsequences elements are appended to are monotonically (but not necessarily strictly monotonically) decreasing. This follows from the fact that remaining elements in the occurrence list will be at most equal in size, or smaller. Hence, subsequent binary searches do not need to consider the full width of the subsequence table, but can instead place the right boundary at the column of last insertion. For example, the insertion of sequence `15 14 10 2` corresponding to the second '`T`' in '`GET / HTTP`' occurs in columns 9, 8, 7, and 2, so the actual number of columns to be considered before each of those insertions is 8 (leading to the creation of the 9th column), 9, 8, and 7, instead of 8 for the first and 9 for all remaining ones.

minimal binary search boundaries

Second, it is worth considering whether binary search is actually necessary. I have found that when considering network traffic, *subsequence locality*, that is, repeated insertions of consecutive Π members into the same subsequence, is high. I present an evaluation of this claim in the next section. Let the subsequence the previous Π member was inserted into be $\mathcal{S}_i$. Then, prefix the binary search with a constant-time check determining whether the next Π member is too large for subsequence $\mathcal{S}_{i-1}$ while admissible by $\mathcal{S}_i$, and if so, insert the element into $\mathcal{S}_i$ and skip the binary search.

subsequence locality

3.5.5.5 *Evaluation*

I have implemented Smith-Waterman and the original as well as extended Jacobson-Vo variants in roughly 500 and 600 lines of C++, respectively. The implementation was done using the framework of the Bro IDS [119]. My implementation of Jacobson-Vo is shown in Appendix A.2. To evaluate the implementations' performance, I selected a number of popular servers

	Prefix Length					
Protocol	50	100	250	500	1000	2000
FTP	9,870	9,730	5,460	✗	✗	✗
HTTP	10,000	10,000	8,778	561	✗	✗
HTTPS	10,000	10,000	9,870	9,316	2,346	630
SSH	10,000	10,000	10,000	9,730	8,385	5,253
SMTP	10,000	7,381	1,431	1,271	703	136
DNS	496	✗	✗	✗	✗	✗
DHCP	10,000	10,000	10,000	✗	✗	✗
NetBios NS	10,000	✗	✗	✗	✗	✗
SNMP	5,778	3,828	1,596	✗	✗	✗
Syslog	3,655	435	✗	✗	✗	✗

Table 3.1. Number of LCS computations per service and prefix length, TCP protocols on top, UDP ones below.

Prefix Length	50	100	250	500	1000	2000
Avg. Speed-up	1.8	5.0	10.7	20.1	28.7	33.0

Table 3.2. Average speed-up of extended Jacobson-Vo compared to Smith-Waterman, for various flow prefix lengths.

from a one-day full-content trace of the Computer Laboratory's uplink. I selected TCP services running FTP, HTTP, HTTPS, SSH, and SMTP as well as UDP services for DNS, DHCP, NetBios NS, SNMP, and Syslog, picking $n = 142$ flows each so that I could perform $\binom{n}{2} > 10,000$ LCS computations among flows pairs of the same service, an operation more meaningful than cross-service alignments. I reassembled the originator $\rightarrow$ responder flows using Bro and stored them in reassembled form for further analysis. Next I measured the runtime for pairwise LCS computations with minimum substring length 1 of the flows belonging to the same service, averaged over the accumulative runtime of 100 iterations, and varied the string length in separate runs among 50, 100, 250, 500, 1000, and 2000 bytes. Since the runtimes of all algorithms are deterministic, I observed very little variation in runtimes and 100 iterations seems a reasonable number of data points for obtaining an good average.

The experiments were conducted on an otherwise idle Pentium 4 running at 2.53GHz and with 512MB of memory. Since flows of at least 2000 bytes are less frequent in the dataset than those of at least 50 bytes, the actual number of string pairs varied per protocol. I chose 100 comparisons as

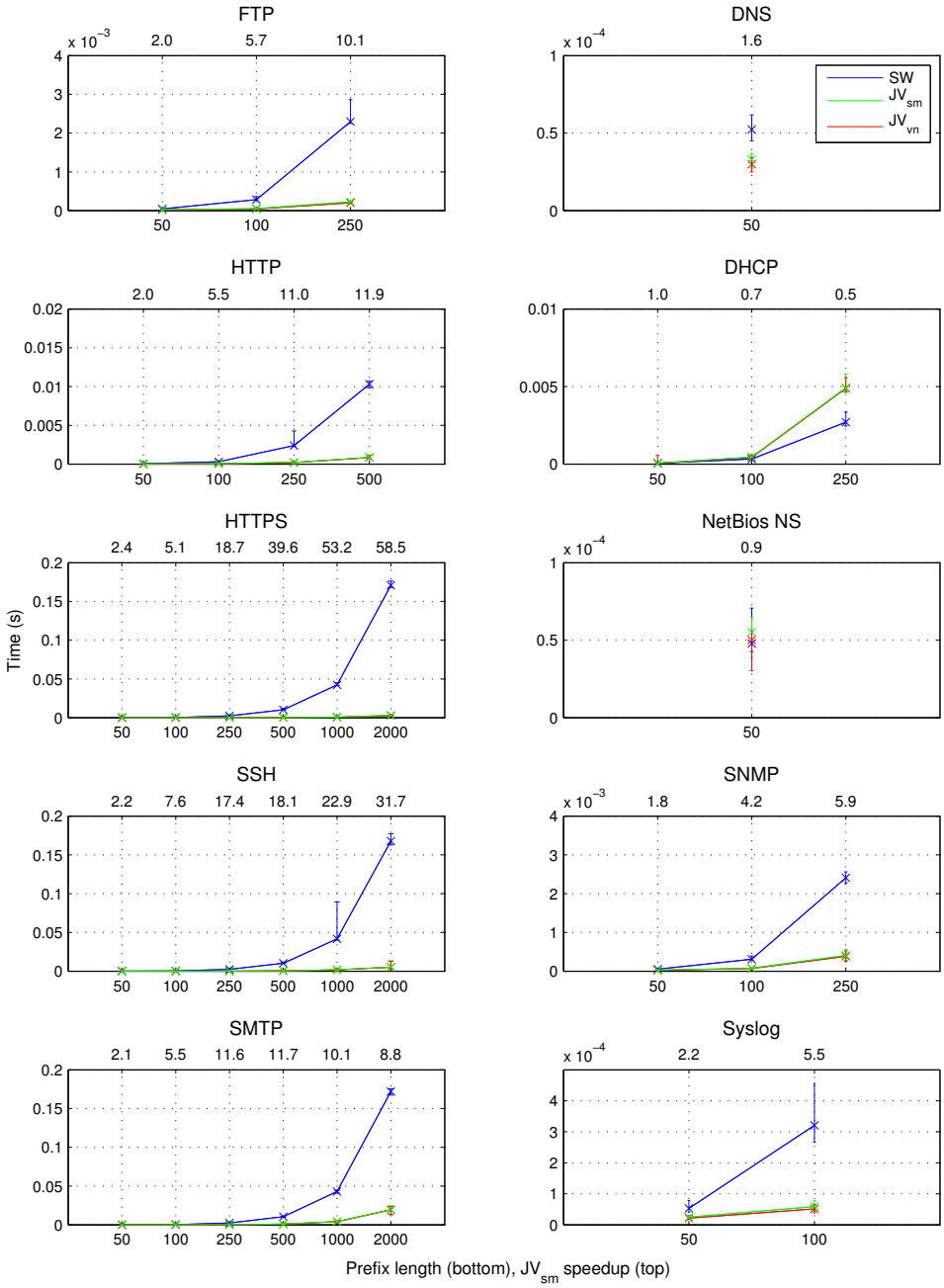


Figure 3.9. Performance comparison of Smith-Waterman, extended Jacobson-Vo, and unmodified Jacobson-Vo applied to intra-protocol alignments of various TCP and UDP protocol flows at different flow prefix lengths. Error bars (often barely noticeable) indicate the minimum and maximum runtime for each experiment.

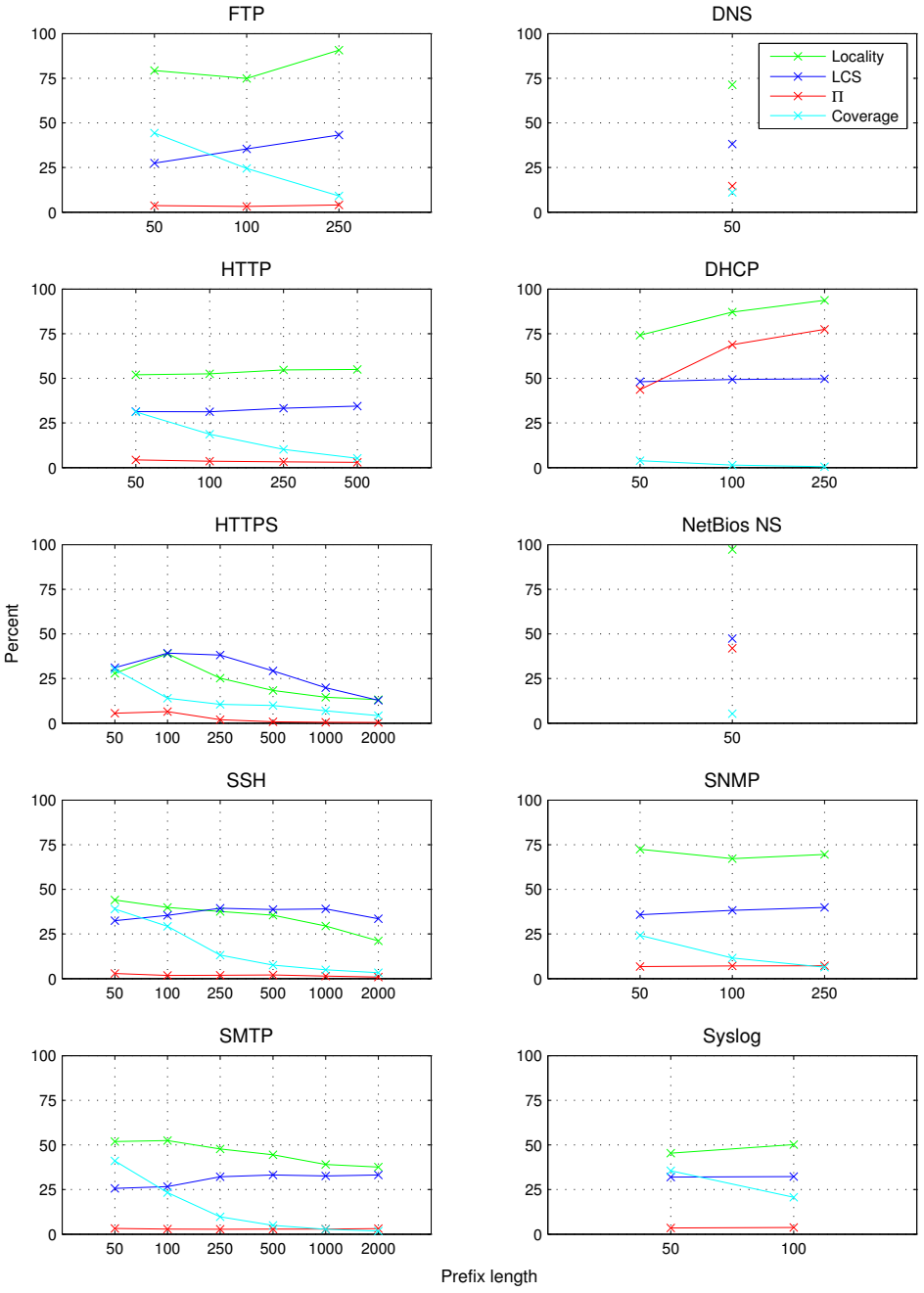


Figure 3.10. Behaviour of various Jacobson-Vo aspects with TCP and UDP protocol flows: subsequence locality during Π element insertions (green), length of LCS relative to $\min(s_1, s_2)$ (blue), length of Π relative to $s_1 \times s_2$ (red), and coverage of corridors relative to entire subsequence table (turquoise).

the lower bound to investigate, to ensure that idiosyncrasies in individual string pairs do not strongly affect the results. The results shown to not diverge noticeably from those obtained when using 150 comparisons as the minimum. The actual number of comparisons made per service and prefix length are shown in Table 3.1. Figure 3.9 shows the performance comparison for all protocols at various prefix length, including the speedup factors of Jacobson-Vo over Smith-Waterman. While the difference in runtime between the Jacobson-Vo variants and Smith-Waterman is initially tiny, Smith-Waterman quickly requires substantially more time to complete than the Jacobson-Vo variants as the prefix length increases. The extended Jacobson-Vo algorithm is up to 33 times faster on average (see Table 3.2) with the best speed-up factor being 58.5 for HTTPS flows of 2000 bytes, and the extended algorithm's additional workload is so marginal that the difference to the original algorithm is barely noticeable.

There are two cases where extended Jacobson-Vo is not the clear winner: NetBios NS and, more strongly, DHCP. To understand the reason, I repeated the alignment experiment just described, now measuring four different properties of extended Jacobson-Vo that may affect the runtime during the experiment. These properties are as follows. First, the *locality* of insertions into the subsequence table, i.e., the fraction of insertions that happen in the same column as the previous one, indicates how much time the algorithm saves by avoiding binary searches for the right column for insertion. Second, the length of the resulting LCS relative to the input string length indicates how much time is spent trying to find the right column for insertion during binary search, and for traversing the table to identify the right LCS. Third, the length of Π relative to $s_1 \cdot s_2$ gives an indication of the overall amount of work Jacobson-Vo has to do relative to Smith-Waterman (recall that the runtime performance of Jacobson-Vo is largely determined by the length of Π). Fourth, and finally, the *coverage* captures the fraction of elements in the subsequence table that are visited during the traversal phase at the end of the algorithm, selecting the optimal LCS. The higher the coverage, the more time is spent in that phase of the algorithm. Consider Figure 3.10. With NetBios NS and DHCP in particular, Π is substantially larger than $s_1 \cdot s_2$, indicating a likely cause for the worse performance. At the same time, relative LCS length is not substantially higher than with other protocols and thus can not explain the slower

runtime. Likewise, DHCP exhibits worse performance even though its subsequence locality is among the highest in the dataset. Finally, subsequence table coverage also does not explain DHCP's behaviour, since DHCP's coverage is among the lowest of the dataset.

In summary, these observations confirm that the length of Π is the defining factor when comparing Jacobson-Vo to Smith-Waterman. Another observvation confirms the results: Jacobson-Vo tends to perform better on content with a high number of characters in random distribution [68]. Indeed, both DHCP and NetBios NS contain a large number of zero-bytes and are of highly fixed structure: the LCSs reach up to 93% of the input string length for DHCP and 97% for NetBios NS, indicating that the input strings are nearly identical. Second, the encrypted HTTPS has high randomisation in large parts of the content, and brings out overall best performance. Knowledge of a protocol's statistical content distribution is thus a guideline for the choice of alignment algorithm.

3.6 Attacks and Caveats

Attacks on structural traffic analysis can be broadly classified into two categories: algorithmic complexity and evasion. Threats to application integrity through means such as buffer overflows and related attacks are a universal problem of the state of the art of software engineering and not included in this classification.

3.6.1 Algorithmic Complexity

Algorithmic complexity attacks potentially affect all layers of the network model. They work by feeding input to applications that makes their algorithms exhibit worst-case performance in space and time, potential leading to denial of service. An example of a vulnerability in this category is an algorithm with a hashtable that uses a fixed hash function, and feeding input to the table that causes all table entries to be mapped to the same overflow chain. Crosby and Wallach [45] present this attack category in detail.

In the presence of an adversary who can control the sequence of individual packet transmission, TCP stream reassembly must carefully prevent out-of-order reception of TCP segments from consuming unnecessary amounts of memory [52]. Beyond that, flow message extraction as presented in Section 3.3 must prevent message state from becoming stale and clogging up the state tables and consuming unnecessary amounts of memory. By using per-flow statekeeping timeouts as described by Dreger et al. [58], we can ensure that all state will eventually be destroyed. The challenge is reduced to one of finding the right expiration timeouts.

3.6.2 Evasion

Evasion attacks are another threat present across all network layers. I have mentioned them briefly in Section 2.3. The goal of these attacks is to sneak content past detection systems by exploiting divergence between the monitors' model of the protocol state machines of the flow endpoints and their actual states. Ptacek and Newsham [125] list the possibilities in detail. There are ways to make evasion harder, for example by normalising traffic before presenting it to the monitor [71, 163], passively and actively mapping the networks a monitor is observing [139], or feeding host-based context to the monitor to reduce ambiguity [56].

traffic normalisation

Flow message extraction possesses at least the same potential for evasion as TCP flow reassembly, since the latter is a requirement for the former. Potential for evasion is present due to inconsistent handling of duplicate content at the endpoints [125]. Message extraction will always work as long as both ends of the flow do not transmit messages simultaneously. The approach is rendered more complicated in the following cases:

- Interactive sessions that transmit user input at per-keystroke granularity, for example in the Telnet and rlogin services. These services are hardly used today. The monitoring application can be made aware of this special case by looking for tiny per-packet payloads that are echoed back to the originator.
- Applications that treat originator → responder and responder → originator flows as independent data channels, as is the case when tunneling multiple sessions through a single flow. The presence of mul-

tiple sessions in a single tunnel connection obfuscates the per-session message exchange, rendering message extraction infeasible. Examples of cases where this occurs are SCTP, where monitoring would have to be aware of individual channels inside the streams, and SSH, where encryption renders individual channels inaccessible.

Interestingly, the sequence alignment algorithms presented in this chapter also present potential for evasion, though only of minor scope. The difference to biology is particularly evident here through the presence of a malicious adversary.

- Longest common region: Computation of LCRs using suffix trees has no inherent potential for evasion. Whatever is the longest common region between two strings will be returned. The danger with LCRs is that an attacker may be able to inject long, identical substrings into multiple flows, thus causing the LCR computation into reporting these injected strings. It depends on the application setting whether this is a shortcoming or not. I will discuss this issue further in Chapter 5.

- Smith-Waterman: Smith-Waterman's flexibility turns into a weakness if the attacker knows the properties of the alignment model used. Most elementally, with knowledge of the minimum common substring length used, the attacker can try to keep below this threshold the length of all substrings that have to be present in the flows for an attack to succeed. It is therefore generally desirable to keep the minimum substring length as small as possible.

  If Smith-Waterman is used for LCS computations, then *sequencing attacks* are a threat. They potentially allow the attacker to conceal the presence of crucial substrings common to the input flows. Assume the attacker has to include strings 'AA' and 'BB' in the attack flow for the exploit to succeed. Assuming the exploit allows sufficient freedom of flow content, the attacker can then include an innocuous *decoy string* to confuse the LCS computation. I now show two variants of such confusion.

  sequencing attacks

  First, by swapping a later common substring with an earlier one in one of the strings, the attacker can prevent any common substrings

between the swapped strings as long as the remaining common subsequence is still longest. Assume the attacker chooses 'PASSWORD' as the decoy and consider the following two strings:

```
AABB PASSWORD
AA PASSWORD BB
```

Despite 'BB' being present in both strings, it is not reported as a common substring since it is not part of the LCS, which is 'AA' - '␣PASSWORD'. If the decoys are longer than the attack-critical strings in total, the attacker can evade detection of the relevant bits altogether by placing the decoys on different sides of the critical strings ('x' and 'y' stand for arbitrary content unique to each string):

```
xxxxxxxx AABB PASSWORD
PASSWORD AABB yyyyyyyy
```

Since 'PASSWORD' is longer than 'AABB', the LCS of the two strings is 'PASSWORD' and the attacker has managed to evade detection of the attack-relevant strings 'AA' and 'BB'. By adjusting the position of the decoy string, subsets of the critical strings can also be hidden selectively.

Two aspects significantly weaken this attack. First, using an ACS-computing variant of Smith-Waterman, the attack-critical common substrings will be detected anyway. Second, once more than a single pair of flows with the same attack is observed by the monitor, it is increasingly more likely that an LCS detecting the important substrings will be computed. This follows immediately if the attacker changes the decoy strings, and also holds when the attacker uses the same decoy strings repeatedly, since there are only a limited number of possible insertion points in the LCS for the decoy strings.

location attacks

A related, but weaker and more subtle variant are *location attacks*. Assume an attacker has to include a certain set of strings in a flow at certain offsets for an exploit to succeed. By including these strings *repeatedly* in the attack flows, in different concatenations, and in different locations, it depends on subtleties in the implementation of the algorithm and the scoring model which LCS is returned. Depending

on the leeway the exploit allows, the attacker has a real chance to obfuscate the location that is crucial to making the attack succeed. On the other hand, the repeated inclusion of the attack-critical information only increases the chance of detection when a Smith-Waterman variant such as ACS is used.

- Jacobson-Vo: In both the original and the extended version, this algorithm suffers from a shortcoming that permits sequencing attacks similar to LCS computations using Smith-Waterman, when given sufficiently flexible attack vectors. The weakness is caused by the fact that Jacobson-Vo *always* computes LCSs and *cannot* switch to a different computation such as ACS as readily as Smith-Waterman, because LCSs are at the very heart of the algorithm. Just as with Smith-Waterman, however, the observation of larger numbers of attack-carrying flows renders sequencing attacks more difficult.

Note that these attacks on sequence alignment are intrinsic to the algorithms and in no way depend on the networking domain to succeed. They apply equally in other domains, for example if they were to be applied in host-based environments to fingerprint binary code or system call sequences.

3.7 Related Work

Traffic analysis is a vast field of which I cover but a fraction in this chapter. This section attempts to put structural traffic analysis in general, and the methods I introduced in particular, in relation to other ways of investigating network traffic.

3.7.1 Other Forms of Traffic Analysis

As mentioned in Section 3.2, structural traffic analysis is feasible across all layers of the network model. In this dissertation, I mostly discuss methods operating at the application layer, with the exception of Section 5.4, which presents a method that operates at lower layers. Other methods of structural analysis are conceivable and have been presented in the literature,

for example analysis of the structure of *communication patterns*[79, 81, 80]. This line of work typically aims to classify traffic according to classes of applications or protocols, which is the topic of Chapter 4 and thus discussed in more detail there.

communication patterns

Other types of traffic analysis aim to detect weaknesses in aspects of individual distributed applications. In context of privacy-enhancing communication services[55], traffic analysis aims to assign network-level activity to individual communicating entities[110]. In context of cryptographic protocols, traffic analysis aims to identify weaknesses in individual aspects of the protocols. An enormous body exists on variants of statistical traffic analysis at low levels of the network model for predicting traffic queueing behaviour, router buffer size requirements, and quality of service guarantees.

3.7.2 Detection of Commonality

The application of sequence alignment algorithms to network flows with the purpose of detecting commonality among the flows has emerged only recently. Generally, detecting commonality is useful at varying levels of accuracy and granularity. At the high-accuracy end, the context has typically been automated malware signature generation, which I will discuss in detail in Chapter 5. Newsome et al. [114] use Smith-Waterman for LCS computations. Their description indicates that their alignment model is minimising gaps through penalties. They do not, however, leverage the precise offset information of the LCS substrings that Smith-Waterman provides. Given the time-critical environment of their system, they would most likely benefit from using Jacobson-Vo instead. Cui et al. [48] use a variant of Needleman-Wunsch for global alignment, trying to identify varying regions among multiple flows for application-level session replay. They guide the alignment process using a constraint matrix that prohibits classes of characters from being paired at certain offsets into the flows. They furthermore develop a message extraction strategy similar to the one I propose.

malware signature generation

A second-level application of alignment information is the derivation of ancestral hierarchy among different instances. Beddoe [13] mentions this possibility in the context of protocol implementations. The recent prolif-

eration of IRC protocol variants for "proprietary" botnet command and control channels could be a fruitful subject for exploring the viability of this approach.

Another line of work satisfies itself with detecting individual strings that occur frequently in a pool of flows, without precise alignment information. While sacrificing detail, these algorithms allow operation on a per-flow granularity instead of flow pairs at higher speeds. Kim and Karp [83] presented Autograph, which extracts frequent common substrings using Rabin fingerprints as previously used in the file system domain to detect redundant content. To automatically determine common substring lengths, they use a *breakmark* that has to be defined ahead of time. Singh et al. [140] likewise base their approach on Rabin fingerprints, but use no adaptive partitioning technique and instead require fixing the common substring length ahead of time.

Rabin fingerprints

Generalising the notion of common content, Bloom filters [18] detect commonality more approximately still and with the presence of false positives. Their use has been proposed in the literature for applications as diverse as object location in peer-to-peer networks [47], geographic routing [88], detecting "heavy-hitter" traffic flows [61], IP traceback [142] as well as IP prefix and signature matching [53, 54]. Related is the notion of sketches [87], which are compact probabilistic summaries of common traffic properties specifically designed to detect changes to those commonalities in high-bandwidth environments.

Bloom filters

sketches

3.8 Summary

In this chapter I have presented traffic analysis strategies feasible at different layers in the OSI network model. I have shown how the application-layer message flow can be extracted heuristically from reassembled flows. Next, I presented several sequence alignment algorithms adapted from bioinformatics that can operate on these messages, and discussed the commonalities and differences incurred when moving these algorithms into the network security domain.

The algorithms presented include longest common region (LCR) computation using suffix trees, flexible gap-minimising longest common sub-

sequence (LCS) computation using the dynamic programming approach proposed by Smith and Waterman, and fast LCS computation using the combinatorial algorithm introduced by Jacobson and Vo. Smith-Waterman's major advantage is its flexibility, which allows it to compute other alignments easily, such as all common substrings (ACS). I then introduced a novel variant of Jacobson-Vo that adds support for flexible alignment models to the algorithm while leaving the runtime complexity bounds intact and causing practically no noticeable overhead in practise. In my evaluation I have shown that this variant computes LCSs up to almost 60 times faster than Smith-Waterman. I then discussed attacks on structural traffic analysis and pointed out the potential of evasion attacks in the various sequence alignment algorithms, which stresses a crucial difference in the network-based application setting compared to the biological environment the algorithms were developed in: a malicious adversary.

In the next chapter I will put to use some of the algorithms introduced above to learn the structure of application-layer protocols and apply it to the problem of traffic classification.

4

Fingerprinting the Normal

"What the...? Who are you supposed to be?"
— Mr. Incredible in *The Incredibles*.

4.1 Introduction

Equipped with models of byte sequences and techniques to extract commonalities therefrom, I now present techniques for improving the understanding of "normal" behaviour on a network. By "normal", I here mean investigation without explicit focus on malicious activity. Normal behaviour on a network is largely characterised by the mixture of applications carried by it, and Section 4.2 outlines the difficulty identifying this range of applications on typical networks in operation today. As will be shown, even perfectly benign use of current network applications leads to a complex mixture of network activity that can often puzzle the administrator. However, as pointed out in Section 2.4.2, good understanding of the normal is a requirement for accurate detection of the malicious. I motivate the use of content-based traffic analysis as a possible solution and argue that high input fidelity is an important goal of traffic models that are intended for a wide range of uses. In Section 4.3, I introduce such a model: taking some of the sequence analysis algorithms introduced in the previous chapter as building blocks, I propose *common substring graphs* (CSGs), a content-based model of flow content suitable for a wide variety of future applications. I thoroughly evaluate the structural properties of CSGs and their runtime behaviour in Section 4.4 and present the task of classifying application-layer protocols as a detailed use case, comparing CSGs to two lower-fidelity traffic models. Finally, I review related work in Section 4.6 and summarise the chapter in Section 4.7.

4.2 Characteristics of Application-Layer Traffic

The Internet architecture uses the concept of port numbers to associate services to end hosts. In the past, the Internet has relied on the notion of *well known* ports as the means of identifying the application-layer protocol a server is using. These well-known ports are standardised de jure by IANA [75]. However, in recent years a number of factors have caused a shift to an increasingly divergent de facto usage of those port numbers. For example, the widespread adoption of firewalling has made ports that typically carry mission-critical applications (such as TCP ports 80 and 25 for HTTP and mail traffic, respectively) much less likely to incur any filtering, causing entirely different applications to switch to these ports or tunnel their traffic through the native protocol where possible. Another typical scenario is the use of dynamically allocated ports to separate application instances, or to explicitly avoid obvious classification. The popular Skype service initialises its listening port randomly at installation, entirely abandoning the notion of well known ports for normal clients [12]. Finally, some applications use non-standard ports explicitly to avoid classification. Peer-to-peer applications routinely allow users to change the default port for this purpose and some use combinations of tunnelling and dynamic port selection to avoid detection [137]. We can expect this trend of irregular port use to increase further in the future. The decreasing value of port numbers for determining flow content undermines the accuracy of network security enforcement, since filtering and access policies are often predicated on the assumption that individual application-layer protocols running on certain ports.

IANA port numbers

Skype

port number obsolescence

None of the typical enforcement mechanisms deployed today can deal with meandering port number usage without manual inspection or tuning. Thus, there is a strong need for techniques that can identify networked applications without considering well-known port numbers. Since it is hard to predict the requirements future tools might demand from such a fingerprinting tool, I argue that the input modelling technique should strive for *fidelity*, i.e., its operation should impose as little loss of information about the input traffic as possible. For example, irreversible abstractions of the input due to hashing, partitioning, or filtering all invariably reduce the operational flexibility of future model applications.

In this chapter I introduce and evaluate a technique called *common substring graphs* (CGSs) for modelling application-layer protocol activity using network flow content, that achieves this goal. The model is based on complete *protocol-typical* substrings in combination with the *positions* in the input flows that these substrings occur at and maintains this information in its entirety for querying after the model build-up phase. CSGs immediately enable *elementary* operations such as the incremental creation of a protocol model from a set of input flows, the pairwise comparison of CSGs to each other to produce a similarity measure between two models of traffic, and the matching of individual flows against CSGs, highlighting the use of protocol-intrinsic content in the flow. This renders CSGs useful for a wide range of analysis purposes such as protocol classification, demonstrated later in the chapter, and whitelisting of protocol-intrinsic content in flows, as discussed later in Section 5.3.3. The throughput achieved by these elementary operations, evaluated in Section 4.4.5, is high enough to allow a wide range of applications with on-line or nearly on-line requirements assuming suitable input filtering is performed, but not sufficient for handling full traffic load in high-bandwidth environments. There, lower-fidelity modelling techniques should be employed as far as the application permits.

The central idea behind CSGs is as follows. By comparing the contents of flows handled by the same destination service (i.e., destination host and listening port on that host[1]), sets of some strings will be common to many compared flows. By capturing the frequency, position, and sequence of such substrings in a graph we obtain a structural model that captures accurately the typical "look" of the application-layer protocol used by the service.

4.3 Protocol Modelling with Common Substring Graphs

The intuition behind CSGs is as follows: if multiple flows carrying the same protocol exhibit common substrings, comparing many such flows will most frequently yield those substrings that are most common in the

[1]Multiple destination hosts are possible in case of load balancing, anycast, etc — what matters is that a unique destination service instance is analysed.

protocol. By using LCSs, not only can we identify what these commonalities are, but we also expose their sequence and location in the flows. By furthermore comparing many of the resulting LCSs and combining redundant parts in them, frequency patterns in substrings and LCSs will emerge that are suitable for classification. CSGs capture much more structural information about flows they are built from than other content fingerprinting methods such as content n-grams or byte product distributions. In particular, CSGs

- are not based on a fixed token length but rather use longest common subsequences between flows,
- capture *all* of the sequences in which common substrings occur, including their offsets in the flows,
- ignore all byte sequences that share no commonalities with other flows,
- track the *frequency* with which individual substrings, as well as sequences thereof, occur.

I will now formalise these concepts. A CSG is a directed graph

$$G = (N, A, P, n_s, n_e)$$

in which the nodes N are labelled and the set of arcs A can contain multiple instances between the same pair of nodes: a CSG is a labelled multidigraph. P is the set of paths in the graph. Paths $p = (n_1, \ldots, n_i)$ are defined as the sequence of nodes starting from n_1 and ending in n_i in the graph, connected by arcs. $P(n)$ is the number of paths running through a node n. (If context doesn't clarify which graph is being referred to, I will use subscripts to indicate membership, as in N_G, P_G, etc.) A CSG has fixed start and end nodes n_s and n_e. Each path originates from n_s and terminates in n_e, i.e., $P_G(n_s) = P_G(n_e) = |P_G|$. These nodes are ignored for all other purposes; for example, when dealing with a path with a single node on it, we mean a path originating at the start node, visiting the single node, and terminating at the end node. Along the path, a single node can occur multiple times; that is, the path may loop. The node labels correspond to common substrings between different flows, and paths represent the sequences of such common substrings that have been observed between flows. CSGs

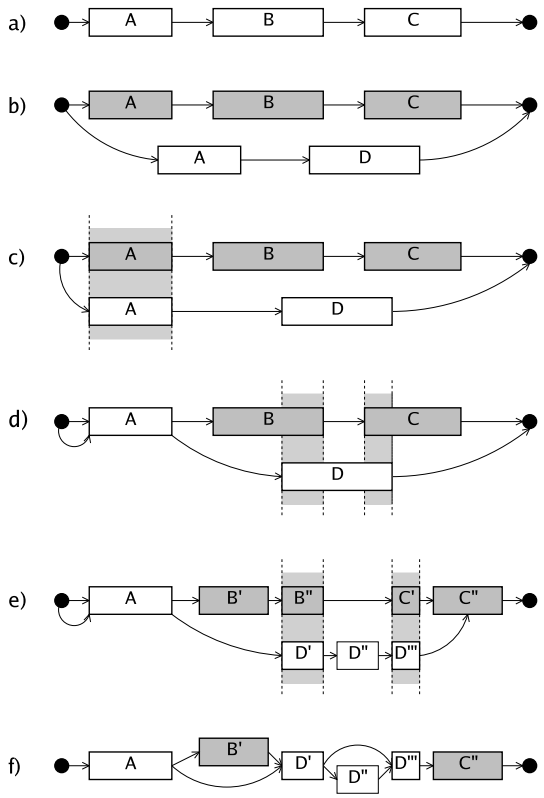


Figure 4.1. Constructing a CSG: introduction of a new path with subsequent merging of nodes. (a) A CSG with a single, three-node path. (b) An LCS (in white) is inserted as a new path. (c) New node A already exists and is therefore merged with the existing node. (d) New node D overlaps partially with existing nodes B and C. (e) Nodes B, C, and D are split along the overlap boundaries. (f) Identically labelled nodes resulting from the splits are merged. The insertion is complete.

grow at the granularity of new paths being inserted. Let the LCS between two strings s_1 and s_2 be $L(s_1, s_2)$ and its cumulative length be $|L(s_1, s_2)|$. For ease of explanation, nodes are synonymous with their labels, thus for example when saying that a node has overlap with another node, we mean that their labels overlap, and $L(n_1, n_2)$ is the LCS of the labels of nodes n_1 and n_2. $|n_i|$ denotes the length of the label of node n_i. Labels are unique, i.e., there is only a single node with a given label at any one time.

Local alignment is computed using Smith-Waterman as described in Section 3.5.3. I now describe four elementary operations on CSGs in more detail. These operations are (*i*) construction of a CSG out of input flows,

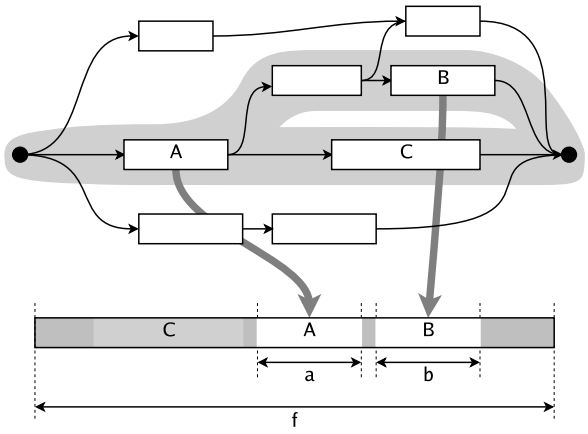


Figure 4.2. Scoring a flow against a CSG. The labels of nodes A, B, and C occur in the flow at the bottom. The shaded area in the graph indicates all paths considered for the scoring function. While the path containing A-C would constitute the largest overlap with the flow, it is not considered because A and C occur in opposite order in the flow. The best overlap is with the path containing A-B: the final score is $(a + b)/f$.

(*ii*) comparison of CSGs to each other to obtain a measure of similarity, (*iii*) merging of multiple CSGs into one, and (*iv*) scoring the degree to which a given flow fits a CSG. Together, they enable a wide range of CSG applications, as I will exemplify by their use in protocol classification in Section 4.4.4.

4.3.1 Construction

Insertion of a flow into a CSG works as follows. A flow is inserted as a new, single-node path. If there are no other paths in the CSG, the insertion process is complete. Otherwise, we compute the LCSs between the flow and the labels of the existing nodes. Where nodes are identical to a common substring, they are *merged* into a single node carrying all the merged nodes' paths. Where nodes overlap partially, they are *split* into neighbouring nodes and the new, identical nodes are merged. We only split nodes at those offsets that don't cause the creation of labels shorter than a minimum allowable string length.

For purposes of analysing protocol-specific aspects of the flows that are in-

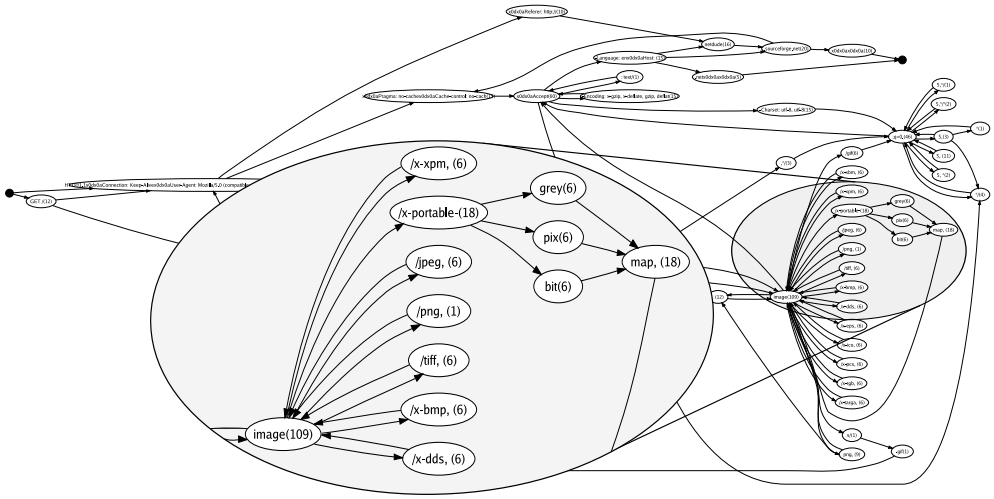


Figure 4.3. Detail of a CSG for the HTTP requests comprising a single website download. The numbers in each node represent the number of paths going through it.

serted into a graph, it is beneficial to differentiate between a new flow and the commonalities it has with the existing nodes in a graph. I therefore have implemented a slightly different but functionally equivalent insertion strategy that uses *flow pools*: a new flow is compared against the flows in the pool, and LCSs are extracted in the process. Instead of the flow itself we then insert the LCSs into the CSG as a path in which each node corresponds to a substring in the LCS. The node merge and split processes during insertion of an LCS are shown in Figure 4.1.

flow pools

Since many flows will be inserted into a CSG, state management becomes an issue. I limit the number of nodes that a CSG can grow to using a two-stage scheme in combination with monitoring node use frequency through a least recently used list. The list keeps the recently used nodes at the front, while the others percolate to its tail. A *hard limit* imposes an absolute maximum number of nodes in the CSG. If more nodes would exist in the graph than the hard limit allows, least recently used nodes are removed from the graph until the limit is obeyed. To reduce the risk of evicting nodes prematurely, I use an additional, smaller *soft limit*, exceeding of which can also lead to node removal but only if the affected nodes are not important to the graph's structure. In order to quantify the importance of a node n to its graph G I define as the *weight* of a node the ratio of the number of

paths that are running through the node to the total number of paths in the graph:

$$W_G(n) = \frac{P_G(n)}{|P_G|}$$

I say a node is *heavy* when this fraction is close to 1. As I will show in Section 4.4, only a small number of nodes in a CSG loaded with network flows is heavy. Soft limits only evict a node if its weight is below a minimum weight threshold. Removal of a node leads to a change of the node sequence of all paths going through the node; redundant paths may now exist. I avoid those at all times by enforcing a uniqueness invariant: no two paths have the same sequence of nodes at any one time. Where duplicate paths would occur, they are suppressed and a per-path redundancy counter is incremented. I do not limit the number of different paths in the mesh because it has not become an issue in practise. Should path elimination become necessary, an eviction scheme similar to the one for nodes could be implemented easily.

4.3.2 Comparison

In order to compare two CSGs, a graph similarity measure is needed. The measure I have implemented is a variant of feature-based graph distances [130]: the two features used for the computation are the weights and labels of the graph nodes. Our intuition is that for two CSGs to be highly similar, they must have nodes that exhibit high similarity in their labelling while at the same time having comparable weight. I have decided against the use of path node sequencing as a source of similarity information for performance reasons: the number of nodes in a graph is tightly controlled, while I currently do not enforce a limit on the number of paths.

When comparing two CSGs G and H I first sort N_G and N_H by the length of the node labels, in descending order. Iterating over the nodes in this order, I then do a pairwise comparison $(n_i, n_j) \in N_G \times N_H$, finding for every node $n_i \in N_G$ the node $n_j \in N_H$ that provides the largest label overlap, i.e., for which $|L(n_i, n_j)|$ is maximised. Let the LCS yielding n_i's maximum overlap with the nodes of N_H be denoted as $L_{max}(n_i, N_H)$. The sorting of the nodes allows us to abort the search once considering nodes that are

shorter than the best match previously encountered, so this algorithm is in $O(|N_G| \cdot |N_H|)$. The score contributed by node n_i to the similarity is then the ratio of the best overlap size to the node label's total length, multiplied by $P_G(n_i)$ to factor in n_i's importance. The scores of all nodes are summarised and normalised, resulting in our similarity measure $S(G, H)$ between two graphs G and H:

$$S(G,H) = \frac{\sum_{n_i \in N_G} P_G(n_i) \frac{|L_{max}(n_i, N_H)|}{|n_i|}}{\sum_{n_i \in N_G} P_G(n_i)}$$

4.3.3 Merging

The way the merge operation proceeds depends on whether the CSG that is being merged into another one needs to remain intact or not. If it does, then merging a CSG G into H is done on a path-by-path basis by duplicating each path $p \in P_G$, inserting it as a new LCS into H, and copying over the redundancy count. If G is no longer required, all paths can be unhooked from the start and end nodes, re-hooked into H, and a single pass made over G's old nodes to merge them into H.

4.3.4 Scoring

To be able to classify flows given a set of CSGs loaded with traffic, one needs a method to determine the similarity between an arbitrary flow and a CSG as a numerical value in the $[0, 1]$ interval. Intuitively I do this by trying to *overlay* the flow into the CSG as well as possible, using existing paths. More precisely, I first scan the flow for occurrences of each CSG node's label in the flow, keeping track of the nodes that matched and the locations of any matches. This is an exact string matching problem and many algorithms are available in the literature to solve it [68]. I am currently using a simple `memcmp()`-iterative approach. The union of paths going through the matched nodes is a candidate set of paths among which I then find the one that has the largest number of matched nodes *in the same order* in which they occurred in the input flow. Note that this gives us the exact sequence, location, and extent of all substrings in the flow

that are typical to the traffic the CSG has been loaded with—when using a single protocol's traffic, one can expect to get just the protocol-intrinsic strings "highlighted" in the flow. Finally, to get a numerical outcome I sum up the total length of the matching nodes' labels on that path and divide by the flow length, yielding 1 for perfect overlap and 0 for no similarity. Figure 4.2 illustrates the process.

4.4 Evaluation

I implemented CSGs as an extension to the Bro IDS to make them accessible to a wide range of future network monitoring tasks and, developed in parallel, in a separate framework explicitly designed for testing classification performance. In the following sections I present an evaluation of CSGs investigating structural aspects, classification performance, and run-time behaviour.

4.4.1 Terminology

In this section, I use the term *session* to refer to all traffic exchanged between two endpoints, using the same quintuple of originator and responder IP address, originator and responder port numbers, and IP protocol (TCP or UDP). A session consists of two *flows*, one containing all packets in originator → responder direction while the other one comprises all packets in responder → originator direction. A TCP session thus contains all traffic belonging to a single TCP connection. A UDP session consists of all packets exchanged within a quintuple with packet inter-arrival times below 10s, the passing of which marks the end of the session. Given the lacking notion of a connection in UDP, a reasonable interval for separating individual "connections" must be used, and 10s has proven to be a reasonable value in practice.

4.4.2 Input Traffic

I used three different sets of real-world, full-packet network traffic traces to evaluate CSGs. The first set was collected at the uplink of the Computer

Laboratory of the University of Cambridge, UK, on 23 November 2003 over a period of 24 hours. This set will be referred to below as the Cambridge trace. The second set consists of traces collected from the uplink of UCSD's Computer Science & Engineering Department, ranging from 30 minutes to 2.5 hours, collected between 30 November 2005 and 7 February 2006. I will be referring to this set as the UCSD traces. The third set, UCSD-w, consists of a five-day capture of the wireless network at UCSD's Computer Science & Engineering Department, starting on 17 April 2006.

4.4.3 Graph Structure

The structure of a CSG (i.e., properties such as its size, the distribution of node frequencies, and path lengths) can influence its usefulness for capturing protocol-specific aspects. CSGs have four parameters: soft/hard maximum node limits, eviction weight threshold, and minimum string length. I suggest a soft/hard node limit of 200/500 nodes, a minimum weight threshold of 10%, and 4-byte minimum string length as reasonable default choices. To validate that these settings are sound, I used the Cambridge traces and selected 4 major TCP protocols (FTP, SMTP, HTTP, HTTPS) and 4 UDP ones (DNS, NTP, NetBIOS Nameservice, and SrvLoc) and for each of them collected 1000 sessions from destination services manually inspected to guarantee they were indeed running the intended protocol. In three separate runs with minimum string lengths of 2-4 bytes, 8 CSGs were loaded with each session's first message while I recorded node growth and usage. Figure 4.4 shows the number of nodes in each graph during the construction. The protocols exhibit fairly different growth behaviours, but all of them tolerate the 200-node soft limit. HTTP repeatedly pushes beyond the limit but never loses nodes at the eviction weight threshold. Figure 4.5 shows the frequency distribution of each CSG's nodes after 1000 insertions. In all CSGs except for the FTP one, at least 75% of the 200 nodes carry only a single path. The FTP CSG only grew to 11 nodes in the 2-byte minimum length run, explaining the cruder distribution. Minimum string length seems to matter little. Thus, my CSG settings seem tolerant enough not to hinder natural graph evolution.

Figures 4.6 and 4.7 show examples of real-world CSGs for protocols DNS, DHCP, HTTP, and SMTP after insertions of 1000 LCSs into each CSG. Darker nodes represent nodes carrying more paths. The visual representa-

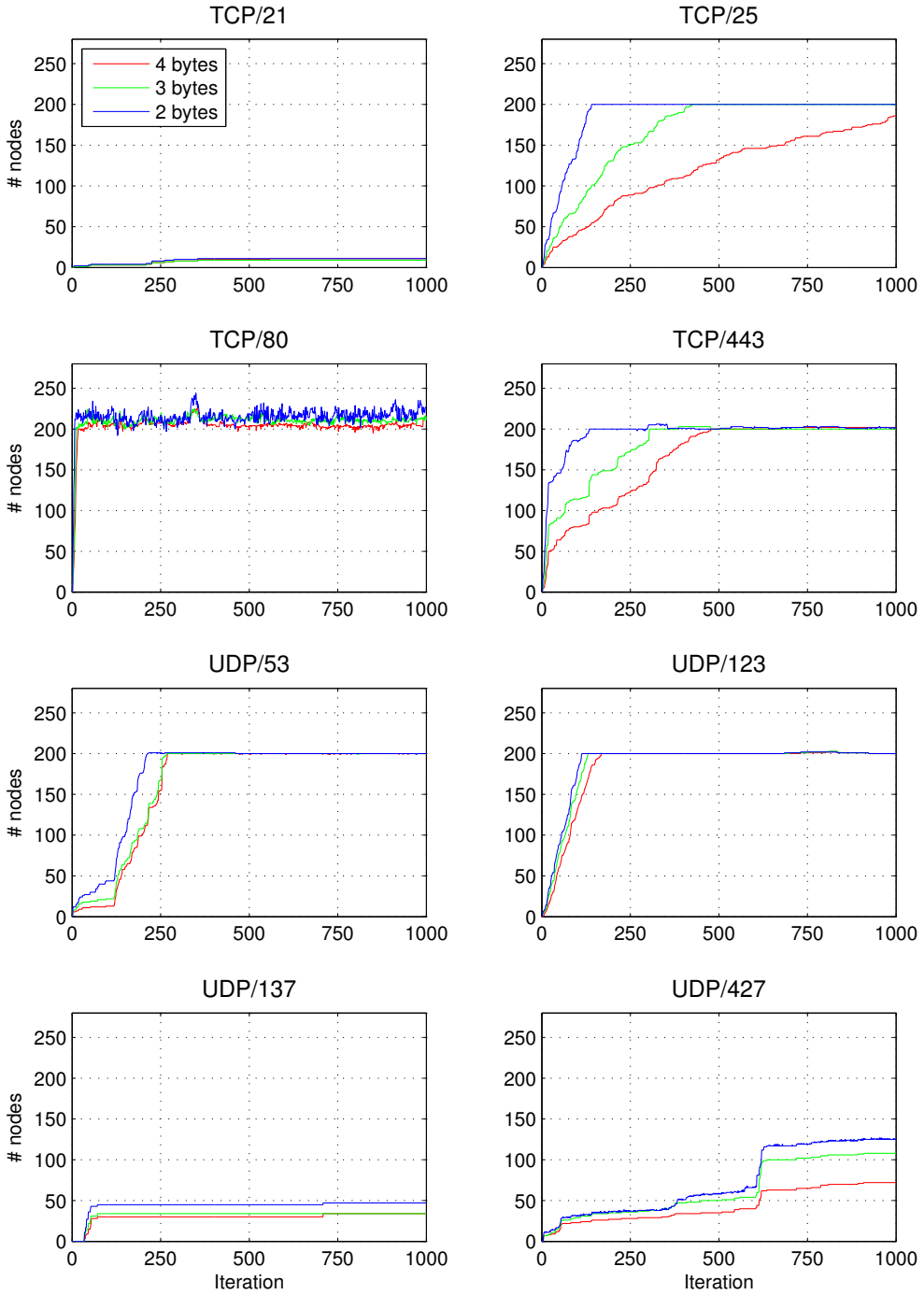


Figure 4.4. CSG node growth during insertion of 1000 sessions, for minimum string lengths of 2, 3, and 4 bytes, with a soft node limit of 200 nodes. Enforcement of the hard node limit of 500 nodes never becomes necessary.

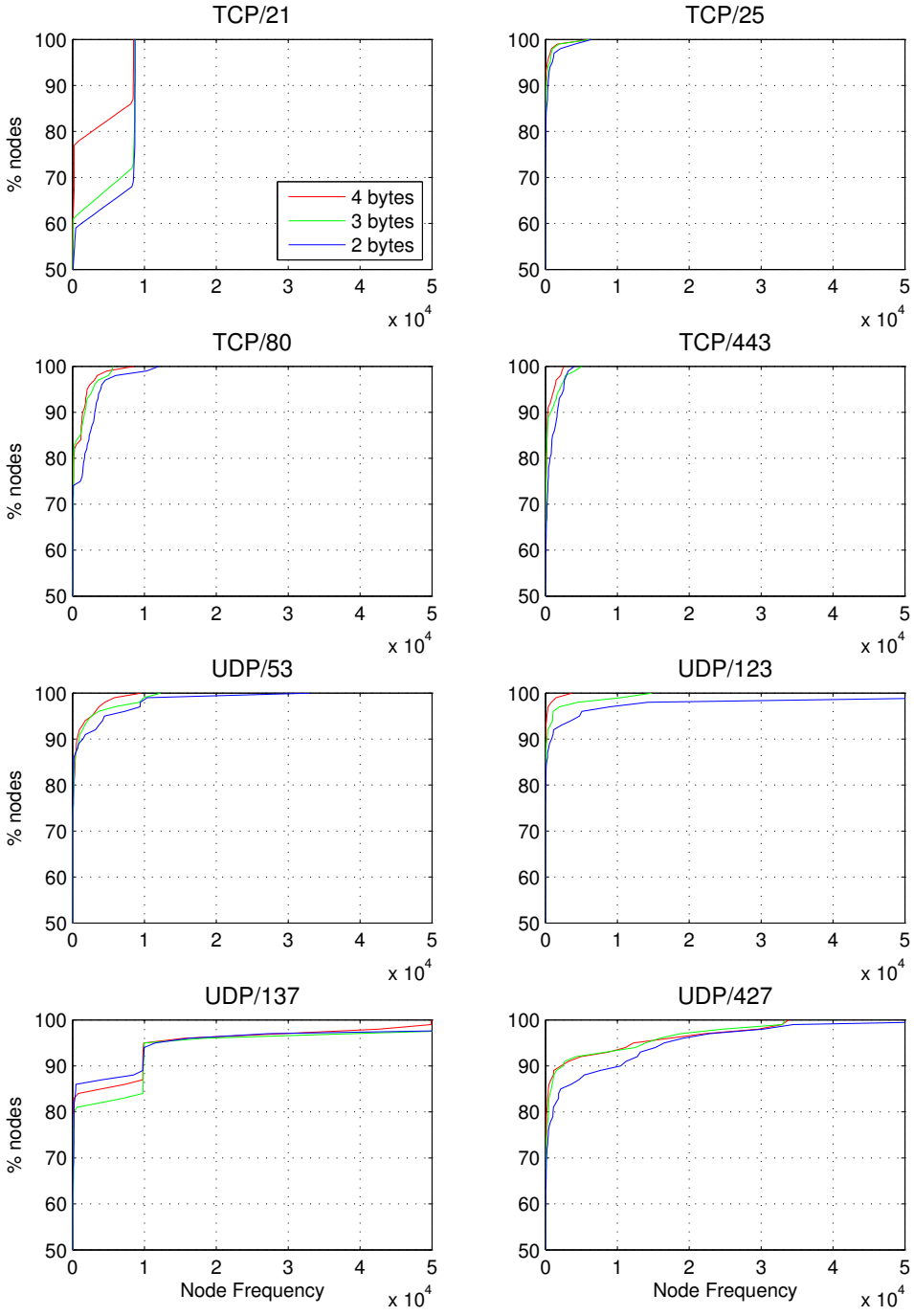


Figure 4.5. CSG node frequencies after 1000 insertions, for minimum string lengths of 2, 3, and 4 bytes.

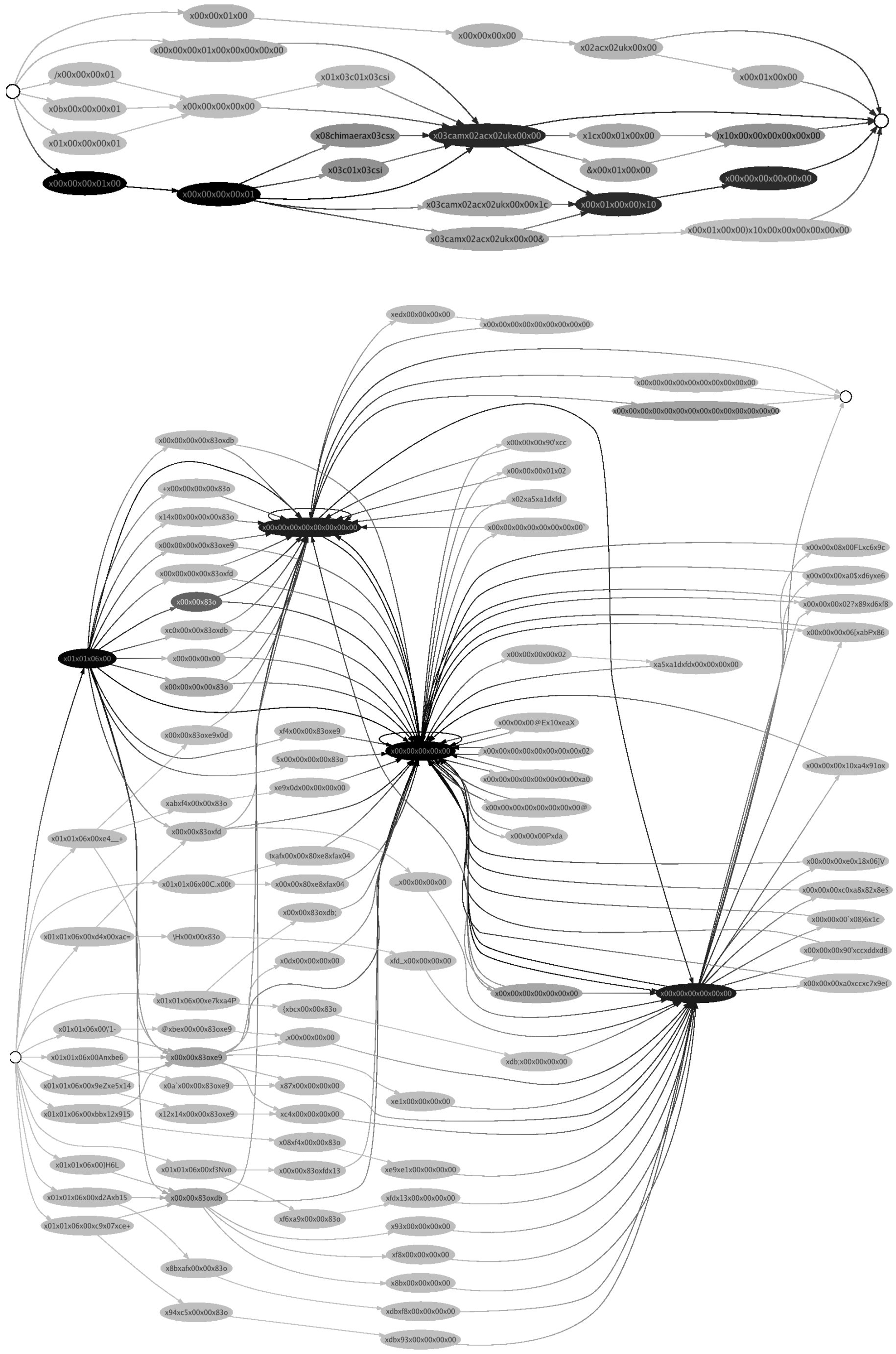


Figure 4.6. CSGs for UDP protocols DNS (top) and DHCP (bottom), flow prefix length 100 bytes, soft node limit 100 nodes. Heavier nodes are rendered in a darker background. Observe that only a small number of nodes are heavy.

Figure 4.7. CSGs for TCP protocols HTTP (top) and SMTP (bottom), flow prefix length 100 bytes, soft node limit 100 nodes. Heavier nodes are rendered in a darker background. Observe that only a small number of nodes are heavy.

tion mirrors the observation from Figure 4.5 that for all protocols, a small number of nodes carry the majority of CSG paths. There is also a distinctive structural difference between binary protocols such as DHCP, where alternative paths often have highly similar nodes, and text-based protocols such as SMTP, where the resulting overall structure is more complex.

4.4.4 Protocol Classification

The work I present in this section was done in collaboration with Justin Ma, Kirill Levchenko, Stefan Savage, and Geoff Voelker of the University of California at San Diego. I contributed to the classification framework itself, integrated CSGs into it, and performed parts of the evaluation.

A major application of CSGs is *protocol classification*: given a session's traffic, the goal is to determine the application-layer protocols present in the session's flows, *regardless* of the session's transport-layer port numbers. As pointed out in Section 4.2, port numbers as the traditional means of classification are becoming increasingly unreliable. To investigate how well CSGs are suited for the task, we implemented a framework for investigating multiple classification techniques. The details of the work are available in a technical report published at UCSD [99] and the corresponding paper [98].

The framework operates as follows. Starting from the assumption that all sessions destined to the same host and port form an equivalence class of application-layer protocol usage, equivalence classes are built for all sessions observed in the input traffic. Each such equivalence class's sessions are stored in a *cell*. For each equivalence class, each session's flow pair is collected, reassembled if necessary, and inserted into two CSGs using the method described in Section 4.3.1, one for each direction. In our experiments, we constrained ourselves to just the first 64 bytes of each flow and ignored any subsequent data. Flow reassembly thus often was not required. The CSGs of each pair of cells that have accumulated at least 500 sessions using the method presented in Section 4.3.2 are then compared in a pair-wise fashion and cells with greatest similarity are merged iteratively, as described in Section 4.3.3, in an agglomerative hierarchical clustering. As merging proceeds, the *merge threshold* increases monotonically until eventually all cells have been merged into a single "supercell."

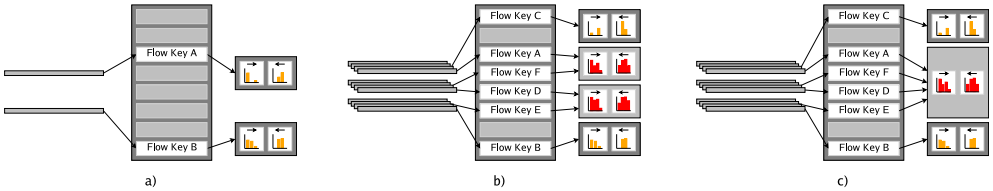


Figure 4.8. The protocol classification framework. (a) Flows are mapped to flow keys, stored in a hashtable. Each flow key points to a cell; the cells are only lightly loaded and have not yet been promoted. (b) More flows have been added, multiple flow keys now point to the same cells. The first cells have been promoted for merging. (c) Cells have begun merging.

Such a sequence of merges along with the merge thresholds at which they occur is illustrated in Figure 4.9. Along the sequence of merges, that constellation of cells is selected which comes closest to our goal of merging only those cells that contain the same application-layer protocol by finding the clustering that minimises the number of misclassifications in each cell. At this point, each cell is labelled with the application-layer protocol the majority of sessions in it exhibit. Figure 4.8 illustrates the cell build-up process. Once the cell configuration is obtained, we can use the framework to classify new sessions by scoring them against cells using the operation presented in Section 4.3.4 and classifying a session as carrying the application-layer protocol of the cell it most closely resembles.

We compared CSGs to two other content-based models of network protocols. I only summarise them here; the full description can be found in the paper [98]. Both models are specifically selected for the classification task and of lower input fidelity than CSGs. The first is a Markov process model which represents a protocol as a Markov chain with 256 nodes, one for each possible byte value. The transition probabilities of each Markov chain are derived from the totality of sessions present in a cell, separately for each direction, by adding up the occurrences of transitions among subsequent bytes. Sessions are scored by performing random walks over the cells' Markov chains, where the walks consist of a number of steps equal to the flow prefix length used. Note that this model completely ignores positional information about byte occurrences inside the flows, since the Markov chain transition probabilities sum up the byte occurrences in their entirety. The second model, which we termed product distributions, factors in positional information more strongly. Here, a 256-element his-

Markov process model

product distributions

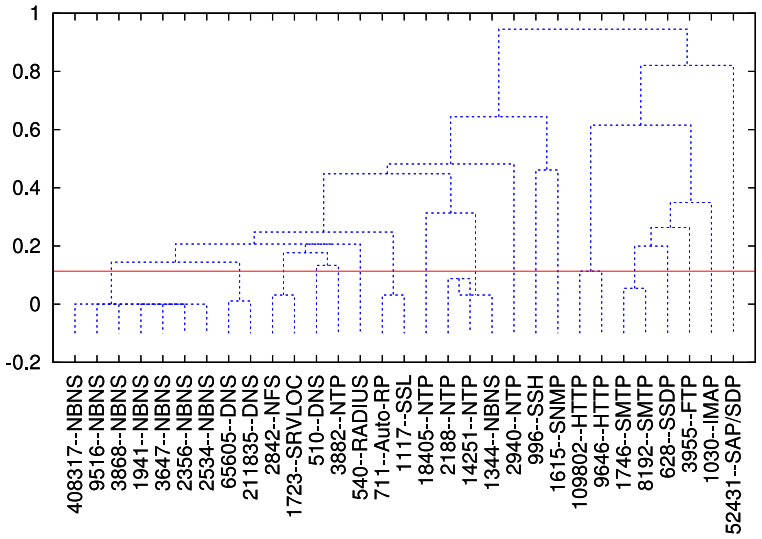


Figure 4.9. Iterative merging of cells while increasing the merge threshold. The horizontal line across the graph indicates the resulting merge threshold.

togram is maintained for every byte offset and each direction of a session's flows. Each histogram bin corresponds to a possible byte value and counts the number of occurrences of each byte value at the histogram's offset. As the model is trained, the histograms accumulate precise information about the frequency of individual bytes' occurrences at different offsets. In contrast to CSGs, no information is stored about consecutive strings occurring at different offsets.

In order to be able to evaluate classification accuracy, we required a classification oracle. We chose Ethereal 0.10.14 [2] for this purpose, despite the fact that it clearly is a weak oracle since it relies almost exclusively on well-known ports to label flows.[3] For our traffic sets, this weakness did not seem to matter. For better classification, the regular expression set from the L7 project [91] could be used.

Each of the trace sets was split into two halves, using the first half to train the models and the second for classification. The overall classification results are shown in Table 4.1. Product distributions achieve best accuracy, followed by CSGs, and Markov processes come last. The good perfor-

[2] `http://www.ethereal.com`

[3] Ethereal uses signature-like heuristic in a small subset of protocol analysers, for example for RPC.

	CAMBRIDGE		UCSD-W		UCSD	
	flows	novelty	flows	novelty	flows	novelty
	226,046	1.18%	403,752	0.51%	1,064,844	1.12%
	total	learned	total	learned	total	learned
Product	1.68%	0.50%	1.78%	1.28%	4.15%	3.03%
Markov	3.33%	2.15%	4.26%	3.75%	9.97%	8.85%
CSG	2.08%	0.90%	4.72%	4.21%	6.19%	5.06%

Table 4.1. Overall traffic statistics (top) and classification accuracies for the three content models (bottom), for the three trace sets. Novelty percentages indicate the fraction of protocols occurring in the test sets but not the training sets. The "total" columns show the overall misclassification encountered in the test trace sets, while the "learned" columns list the misclassifications among flows labelled with protocols that were present in the training sets.

mance of product distributions largely stems from the fact that they are able to fingerprint frequently occurring distributions of individual byte values at fixed offsets, as typically found in binary protocols. While the substring-based approach taken by CSGs is better at picking up freely movable strings and their sequencing, this does not translate into a decisive advantage in the classification task. It does however mean that CSGs are generally better at classifying text-based protocols than binary ones. Since text-based protocols however typically *also* have strings occurring at fixed offsets (such as the beginning of the flows, in particular), product distributions generally work well in those cases as well. The Markov process model cannot leverage any information about different offsets into the flows and suffers accordingly. These observations are confirmed by the detailed confusion ratios for CSGs shown in Table 4.2 and the overall accuracy ratios presented in Table 4.3: the larger contributors to the overall classification error are generally binary protocols. DNS, in particular, is a major culprit. Note that encrypted protocols are *not* generally hard to classify, assuming the connection setup is observable: the example of SSH shows that such scenarios are handled well in general, and CSGs are in fact performing better than both other models in this case. This would clearly no longer be the case if flow monitoring was cold-started in the middle of an already encrypted exchange. In this case, CSGs would struggle to find enough structure to build a model from, while the other models would likely build a useful model of the homogeneously distributed "line noise" of encrypted content.

Cambridge			UCSD-w			UCSD		
total	error	protocols	total	error	protocols	total	error	protocols
5983	0.696%	*FTP-DATA* → HTTP	15089	0.865%	ENIP → DNS	11379	0.585%	ISAKMP → DNS
1635	0.190%	*FTP-DATA* → SAP/SDP	9713	0.557%	DNS → NBNS	11265	0.580%	YPSERV → DHCP
1547	0.180%	SRVLOC → NFS	6358	0.364%	ENIP → POP	8391	0.432%	SSL → SMTP
1488	0.173%	NFS → DNS	5079	0.291%	MDNS → DNS	6928	0.356%	POP → SMTP
1145	0.133%	NTP → DNS	4212	0.241%	ENIP → MSNMS	5852	0.301%	CLDAP → LDAP
1000	0.116%	DNS → NBNS	3502	0.201%	ENIP → HTTP	5649	0.291%	Portmap → NFS
795	0.092%	Auto-RP → NFS	3028	0.174%	ISAKMP → DNS	4671	0.240%	*RADIUS* → HTTP
659	0.077%	NTP → NFS	2668	0.153%	ENIP → CLDAP	4571	0.235%	DNS → NBNS
383	0.045%	SNMP → SSL	2421	0.139%	NTP → RUDP	3564	0.183%	SSL → HTTP
337	0.039%	*FTP-DATA* → FTP	2175	0.125%	ENIP → SNMP	2736	0.141%	HTTP → SMTP
328	0.038%	*POP* → FTP	1879	0.108%	ENIP → IMAP	2412	0.124%	SNMP → DNS
251	0.029%	*FTP-DATA* → SSH	1867	0.107%	ENIP → AIM	2216	0.114%	HTTP → Socks
249	0.029%	SRVLOC → SSDP	1660	0.095%	NTP → NBNS	2009	0.103%	DNS → ISAKMP
193	0.022%	*Syslog* → HTTP	1601	0.092%	ENIP → CUPS	1792	0.092%	KRB5 → Slammer
192	0.022%	*Portmap* → NFS	1446	0.083%	ENIP → BOOTP	1786	0.092%	YPSERV → NFS
131	0.015%	*BROWSER* → NBNS	1338	0.077%	ENIP → NBNS	1732	0.089%	DNS → DHCP
130	0.015%	*Syslog* → SSH	1284	0.074%	ENIP → SMTP	1464	0.075%	NBSS → Portmap
94	0.011%	NTP → NBNS	941	0.054%	*MANOLITO* → NBNS	1311	0.067%	SSDP → HTTP
85	0.010%	*FTP-DATA* → SSDP	931	0.053%	RX → RUDP	1224	0.063%	NTP → DCERPC
83	0.010%	*FTP-DATA* → IMAP	798	0.046%	*UDPENCAP* → NTP	1140	0.059%	NTP → Messenger
79	0.009%	*FTP-DATA* → NFS	781	0.045%	SSL → RX	1129	0.058%	SSL → DNS
78	0.009%	*Syslog* → SAP/SDP	724	0.041%	NTP → BOOTP	922	0.047%	*XDMCP* → DNS
63	0.007%	*FTP-DATA* → NBNS	593	0.034%	ISAKMP → MDNS	914	0.047%	*RIPv1* → SRVLOC
60	0.007%	*HSRP* → RADIUS	580	0.033%	ENIP → STUN	913	0.047%	*FTP-DATA* → SMTP
58	0.007%	DNS → NTP	538	0.031%	*MANOLITO* → MDNS	876	0.045%	*RX* → NTP
56	0.007%	NTP → HTTP	524	0.030%	ISAKMP → NTP	853	0.044%	Syslog → HTTP
56	0.007%	FTP → SMTP	423	0.024%	*UDPENCAP* → DCERPC	845	0.043%	ISAKMP → SMB
43	0.005%	*FTP-DATA* → SMTP	349	0.020%	MDNS → BOOTP	845	0.043%	*ECHO* → HTTP
42	0.005%	NFS → NBNS	346	0.020%	*HSRP* → DNS	838	0.043%	*WHO* → HTTP
41	0.005%	NBNS → DNS	293	0.017%	*BitTorrent* → DNS	832	0.043%	RTSP → HTTP

Table 4.2. The top 30 protocol misclassifications for CSGs, for each of the three trace sets, sorted in descending order. Misclassified protocol flows are shown in both absolute numbers and percentages they contribute to the overall error. Italicised listings indicate novel protocols that were not in the training set.

			Product			Markov			CSG	
	protocol	%	err.%	prec.%	rec.%	err.%	prec.%	rec.%	err.%	prec.%	rec.%
Cambridge	DNS	26.28	0.09	99.94	99.78	0.61	97.89	99.97	0.45	98.82	99.52
	HTTP	12.24	0.07	100.00	99.99	0.09	100.00	99.98	0.74	99.91	99.99
	NBNS	44.89	0.35	100.00	99.25	0.40	99.82	99.31	0.17	99.71	99.99
	NTP	5.29	0.00	100.00	100.00	1.19	99.96	77.84	0.25	99.83	95.65
	SSH	0.22	0.14	68.39	100.00	1.10	17.39	100.00	0.05	99.22	100.00
UCSD-w	DNS	23.14	0.04	99.88	99.93	0.29	98.88	99.99	1.97	94.37	97.59
	HTTP	0.67	0.27	76.02	97.54	0.09	90.68	99.93	0.22	76.87	99.38
	NBNS	6.94	0.00	100.00	100.00	1.96	78.06	100.00	0.81	90.34	99.97
	NTP	0.57	0.01	99.95	99.72	0.51	100.00	11.29	0.40	86.65	48.76
	SSH	0.44	0.17	75.28	100.00	0.00	99.63	100.00	0.00	99.99	100.00
UCSD	DNS	54.78	0.26	99.90	99.95	1.90	97.13	99.98	1.43	98.47	99.15
	HTTP	9.17	0.38	97.46	99.62	0.33	97.21	99.72	1.21	95.14	97.19
	NBNS	7.03	0.01	100.00	99.81	1.25	85.66	99.81	0.33	96.04	99.45
	NTP	6.70	0.02	99.99	99.94	5.39	78.07	29.61	0.36	99.82	96.58
	SSH	0.08	0.08	68.81	81.82	0.09	0.00	0.00	0.03	95.40	82.01

Table 4.3. Total classification error, precision, and recall of selected protocols, for all the traffic models and trace sets. The percentage column following the protocols is the proportion of the protocol in the entire trace.

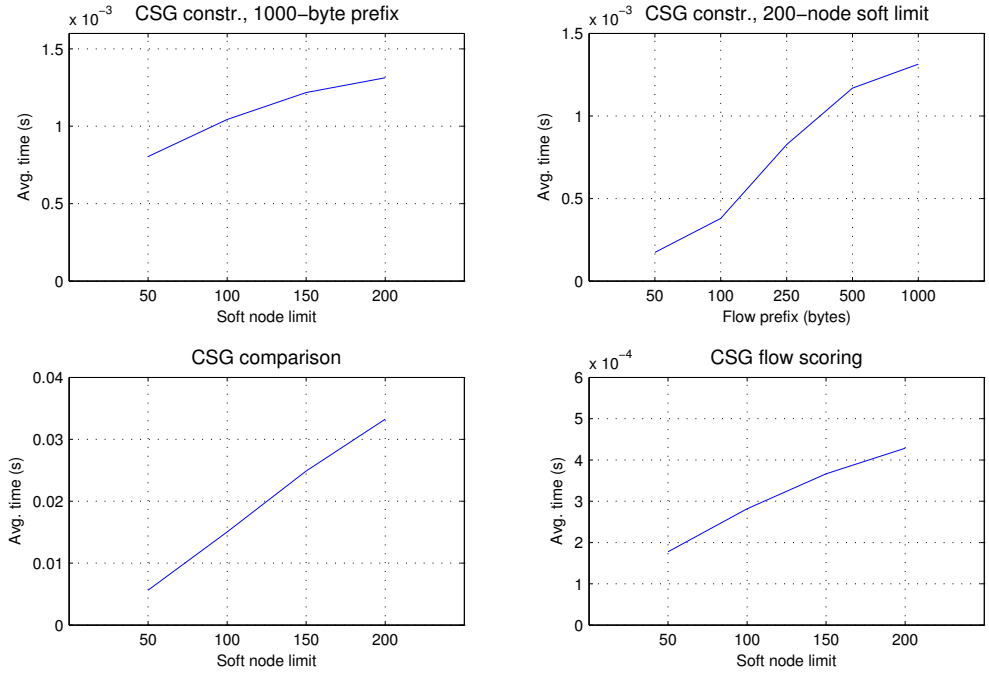


Figure 4.10. Performance of CSG construction with varying soft node limits and flow prefix sizes (top left and top right, respectively), comparison (bottom left), and flow scoring (bottom right). Error bars are omitted since do significant variation was noticeable.

4.4.5 Runtime Behaviour

To evaluate the runtime performance of the CSG operations, I first selected 1000 flows of a representative set of application layer protocols: DNS, DHCP, NTP, NetBios NS, SNMP, SrvLoc, and Syslog for UDP, and FTP's command channel, SSH, SMTP, HTTP, POP3, IMAP, and HTTPS for TCP. All experiments were made on an otherwise idle Pentium 4 running at 2.53GHz and 512MB of memory, the same machine used in Section 3.5.5.5.

To measure CSG creation performance, I then averaged the times needed to insert 1000 LCSs into individual CSGs per protocol. In separate runs I adjusted the soft node limit between 50 and 200 nodes while using a flow prefix of 1000 bytes, and the flow prefix size from 50 to 1000 bytes while using a soft node limit of 200 nodes. In the former experiment, my implementation of CSGs can insert LCSs at a rate of 1,243 to 761 per second, while in the latter the rate varies from 5,764 to likewise 761 per second.

The average insertion rate across the two experiments is 1,092 insertions per second. Note that these results do not include the time required to obtain the common substrings that were inserted. Recall Section 3.5.5.5 for the runtime requirements to obtain LCSs using Smith-Waterman and Jacobson-Vo, but also note that depending on the application, CSGs do not necessarily require common substrings as input but can also use entire strings, since content unique to individual flows will quickly be pushed out of the CSG.

Next, I measured the time needed to compare CSGs to each other by loading into memory all CSGs created during the construction experiment with the same soft node limit, performing pairwise comparisons among each of those sets 10 times, and computing the average time needed. The comparison rate ranges from 177 to 30 per second. These rates are significantly lower than those needed for CSG construction, but note that traffic model comparison is often less frequently required than LCS insertions or flow scoring, which I investigated next.

Again using the CSGs of varying soft node limits built during the CSG creation experiment, I created a random mixture of 1000 flows from all protocols in the dataset and measured for each application-layer protocol the average time needed to score those flows against each of the CSGs. Scoring performance ranges from 5,630 to 2,334 flows per second.

Figure 4.10 summarises these results.

4.5 Discussion

The evaluation shows that CSGs perform well when used as traffic classifiers. When compared to other traffic models, they score worse than product distributions, but better than Markov models. Not surprisingly, CSG's classification results are best when analysing text-based protocols with protocol-typical strings that are different from those of the other protocols, such as HTTP or IMAP. Classification of binary protocols is complicated by two factors. First, binary protocols do not necessarily have protocol-intrinsic tokens that naturally delimit semantic entities in the flows. This results in greater difficulty for sequence alignment algorithms to identify

definitive commonalities. Product distributions manage to overcome this hurdle due to their single-byte granularity. Second, several binary protocols exhibit presence of zero-byte strings, sometimes in close proximity (such as with DNS vs. NTP), which further complicates accurate differentiation.

On the other hand, product distributions are not without shortcomings: CSGs offer the unique benefit of providing protocol-intrinsic substrings in their entirety and with precise information about the whereabouts of their occurrence along with their frequencies. This makes them much easier to translate into content-based signatures that are immediately useful to present-day IDSs. CSG's main strength, the focus on common substrings, is also its main weakness: only substrings that were observed during training can later be used for classification. Binary protocols make the presence of such strings less certain, though the fact that I used a minimum string length of 4 bytes while achieving good classification results shows that this is not a fundamental hurdle.

The runtime performance of CSGs is good even though they are more complex than more statistical models, such as product distributions. The traffic volumes at which CSGs can operate at line speeds will strongly depend on how CSGs are used. In high-bandwidth environments, sampling could be used to selectively reduce the load on ports with high traffic volumes during model construction. Once the model is built, CSG comparison and flow scoring are read-only operations on the CSGs and can be parallelised with ease.

Over time, network traffic undergoes shifts in application-layer content as new applications are introduced, older ones are phased out, and existing ones update their protocol implementations. I have not investigated such "drift" issues and leave them for future work.

4.6 Related Work

The work I have presented in this chapter falls under the broad umbrella of traffic profiling, which has been researched quite thoroughly. It is often the precursor for anomaly detection, because detecting deviation from

the norm first requires a solid understanding of the norm. Much as in attack detection, traffic classification has been attempted with methods at varying depths of traffic inspection. I group the reviewed work roughly in increasing order of depth of analysis, i.e., going from lower to higher levels in the OSI network model.

Recent work by Karagiannis et al. has investigated transport-layer contact patterns among sets of hosts, trying to delineate emerging patterns for different classes of application-layer services (e.g., peer-to-peer applications vs. email) [79, 81]. Their approaches work well, but cannot distinguish among different protocols exhibiting similar contact patterns. Such patterns are orthogonal to CSGs and both approaches could be combined well.

Using a more statistical approach than Karagiannis et al. , Lakhina et al. [92] likewise employ transport-layer features for classification. Their goal is to identify anomalies, not necessarily of types known in advance, in large flow sets. They use entropy as the main feature, and obtain meaningful clusters for several well-known types of anomalies. Similar to our Cell framework, their approach supports fully unsupervised classification.

Several efforts have used manually generated signatures as the core engine of flow identification [51, 137, 80]. Manually generated signatures have two significant drawbacks: first, one has to know in advance what protocol one is actually looking for; second, manual signature generation is tedious and prone to errors on the sides of false positives or negatives. Statistical learning techniques can help with the latter problem, as has been demonstrated by Haffner et al. [69]. Not only can CSGs provide such signatures without manual intervention; they can also provide information about the relative frequencies with which elements of such signatures do occur in practise.

Moore and Papagiannaki [104] use a set of 9 classifiers operating at multiple levels of the network model and varying levels of complexity. They suggest a classification procedure using incrementally more classifiers, connecting them through causal reasoning. In contrast to our work, their approach is not fully automated, classifies at the granularity of 10 different classes of applications, and only manages to achieve or exceed our level of accuracy if 8 or 9 of their classifiers are used. Zuev and Moore [174]

employ network-level packet headers as features for supervised Bayesian classification. Using flows manually labelled as a baseline, they classify the packets into 10 general application classes and achieve accuracies ranging from 66% to 83%. The advantage of their approach is that access to flow-level content is not required; however, their accuracy is significantly worse than ours.

Zander et al. [171] suggest a framework for unsupervised protocol classification using Expectation Maximisation [49] to derive classes of traffic based on elementary statistical features such as packet inter-arrival times, packet length distribution, and flow size and duration. Their work focuses on the automated selection of good features for individual traces and currently gives no information on classification accuracy, whereas our framework focuses on fully automated classification of traffic given a fixed feature set.

A problem similar to the classification of flow content is that of classifying file types. Li et al. [93] use n-gram profiles for this purpose and achieve good accuracy; their work is similar to the product distributions in our traffic classification framework. Another application using a product-distribution-like is presented by Tang and Chen [153], who use byte distributions to fingerprint exploits in flow content.

Another general way of fingerprinting the normal is through specification and enforcement of adherence to such specification. Normalisation of network traffic [71, 163] consists of first formulating what constitutes compliant traffic, then detecting deviation from that profile, and finally deciding how and whether the deviation can be corrected without adversely affecting the end-to-end functionality of the traffic flows.

Finally, Cui et al. [48] likewise employ sequence analysis in a protocol-agnostic fashion, but use it model *global* commonality in application-layer sessions with the intent to replay protocol exchanges at one endpoint of the communication.

4.7 Summary

In this chapter I have illustrated the importance and difficulty of gaining accurate understanding of the application-layer protocols present in a network. The main contribution of the chapter is the introduction of a new model of network flows, called common substring graphs (CSGs). CSGs use sequence alignment to collect protocol-typical substrings and associate these substrings with the positions and frequency with which they occur in the input flows. CSGs are a *high-fidelity* content model: the content strings and their positional distributions remain fully accessible after model build-up, making CSGs useful in a wide range of applications. CSGs provide elementary operations such as the incremental construction of a content model from a set of input flows, pairwise comparison of CSGs to each other to measure similarity between two models, and the matching of individual flows against CSGs, highlighting the use of protocol-intrinsic content in the flow. As an example of a CSG application, I have investigated their suitability for classifying application-layer protocols. CSGs are able to classify individual application-layer protocols with an error rate ranging from 2.08% in the best to 6.19% in the worst case. When compared to product distributions and Markov process models, CSGs offer the best compromise between classification accuracy and detail of flow content.

5

Fingerprinting the Malicious

"Sergeant. Establish a recon post downstairs. Code Red. You know what to do."
— Woody in *Toy Story*.

5.1 Introduction

In the previous chapter I addressed the problem of extracting the structure of application-layer protocols in the absence of reliable transport-layer labelling. The techniques presented can be used to derive a baseline of what constitutes normal activity. In this chapter, I will narrow the focus to the identification of *malicious* activity in network traffic. I begin by showing ways to do so by contrasting malicious traffic against the normal in Section 5.2, and derive ways to capture the essence of such malice in two forms: a content-based one in Section 5.3, namely *automatically generated signatures* for the identification of attack exploits as they are attempted, and a more statistical one, *Packet Symmetry*, which can be used proactively to prevent volume-based attacks from entering the network core, in Section 5.4. I survey related work in Section 5.5, and conclude the chapter with a summary in Section 5.6.

5.2 Defining Malice

Network traffic is considered malicious when it violates a site's security policy, for example by gaining an attacker access to end hosts, or by rendering services offered by the site unreachable to its legitimate users. Operating specifically on malicious traffic is thus predicated on two require-

ments: first, one requires a means of specifying what constitutes malice; second, there must be a way to apply this specification to detect traffic matching the specification. The challenges are to derive a suitable specification and to turn it into a classifier that can detect traffic affected by the specification with suitable accuracy (recall Section 2.4.2).

In this chapter I focus on two kinds of malicious behaviour: (*i*) content-based exploitation of vulnerabilities in networked applications due to carefully crafted payload, and (*ii*) volume-based attacks such as aggressive scanning and denial of service.

5.2.1 Content-based Attacks

A large class of attacks on networked computers aims to exploit vulnerabilities in the software running on end systems. Feeding unexpected input to applications not processing such input in a robust fashion can cause these applications to crash or, as is more frequently attempted, coerce the software into executing code fed to it by the attacker. The last bit is crucial: in order for an attack to succeed, the offending content has to be carried to the victim machines over the network. Therefore, identification of those parts of a network flow that contain the exploit is one possibility of defining malice. Content-based traffic signatures are a major pillar of intrusion detection; the idea here is to express concisely the characteristics of the exploit as it is observable on the wire, and react according to a site's policy when a signature matches live traffic.

This approach to defining and detecting malice is not without problems:

- First, it requires that traffic is not encrypted, since encrypted traffic completely obscures the actual flow content. This has been known for a long time, but has not become as fundamental an obstacle to content-based detection as one would assume. The main reason is the fact that many major applications (such as the World Wide Web or electronic mail) do not *necessarily* require encryption. Strong counter-arguments to this line of thought are the emergence of new applications that heavily use encryption and could be used as exploit vectors (such as Skype [12, 16] and VoIP in general), and the possibility for attackers to employ encryption in their own communication

encryption as a threat

infrastructures such as botnets.

- Second, while there is a duality between vulnerabilities present in host software and the exploits that attack them, this is not necessarily a one-to-one relationship: it is frequently possible to encode the exploit in a number of different ways that still allow the attack to succeed. Such *polymorphism* of exploits is a threat to content-based signatures, since they significantly raise the bar of the accuracy required from a signature set — polymorphism drastically increases the chance of false negatives. It is worth noting however that there is no universal agreement on the exact extent of the polymorphism threat. While a high degree of polymorphism has been demonstrated for some vulnerabilities [44], the amount of invariant content an attacker can not work around depends strongly on the vulnerability. Polymorphism underlines the importance of avoiding *enumerating badness* [126]: engaging in an arms race with attackers to find a signature for every variant of an attack, instead of managing to capture the essence of the vulnerability in a smaller set of signatures remaining constant over time.

  vulnerability vs. exploits

  polymorphism

  enumerating badness

- Third, detection of attempted exploitation as it occurs is inherently *reactive* in nature. Clearly, it is preferable to protect a site's infrastructure proactively. Ideally, software would just be written with less vulnerabilities, but while this is a vast research field in itself [4], it is unlikely to become widespread any time soon. A fundamental hurdle is the fact that the economics of the software industry do not hold the creators of software liable for errors, leading to insufficient incentive to make software secure from the outset [134]. More incremental yet proactive measures do exist though; one example is fast and automated patch handling to fix vulnerabilities as soon as they are fixed by the vendors. This is becoming more and more of a basic requirement, since attackers are trying to derive the vulnerability from the patches, and the delay between published software updates and appearance of exploits for the corrected vulnerabilities is ever-decreasing [156, 157].

  reactive measures

Despite these problems, content-based attack detection remains one of the most important strategies when monitoring a network, and the work I

present in Section 5.3 aims to improve one of the most tedious aspects of content-based defence: identification of attack-crucial elements present in network traffic.

5.2.2 Volume-based Attacks

Volume-based attacks are orthogonal to payload-based ones: in contrast to the latter, the content of traffic comprising such an attack matters little, it is the presence of the attack traffic and the amount of it that is problematic. In volume-based attacks, malice is defined by the presence of enough unwanted traffic to crash the victim machines or to prevent legitimate users from reaching them. Since the content of such traffic matters little, it is just as hard to come up with precise attack signatures as with some content-based attacks, however the danger here is one of high false positive rates, causing substantial collateral damage by dropping legitimate clients' traffic along with the attacker's.

potential false positives

Compared to content-based attacks, the sources of volume-based ones are frequently much harder to identify since spoofing source addresses is feasible, given enough machines to spoof from. The advent of large-scale botnets with potentially hundreds of thousands of attacking machines has made this one of the most dominant threats to the Internet, and much work has been done to tackle it. While the majority of existing work in this space proposes reactive mechanisms that attempt to establish intricate and wide-spread filtering once a site detects that it is under attack, the work I present in Section 5.4 takes a more proactive stance: by making the detection of abusive volumes of traffic pervasive and placing it close to the sources of the attack streams, large-scale denial-of-service attacks are made drastically more difficult.

source address spoofing

botnet threat

5.3 Automated Signature Generation using Honeypots

In recent years, *honeypots* (briefly mentioned in Section 2.2), essentially decoy computer resources instrumented to monitor and log the activities of entities that probe, attack or compromise them [146], have become popu-

lar. Honeypots come in many shapes and sizes; examples include dummy items in a database, low-interaction network components such as preconfigured traffic sinks, or full-interaction hosts with real operating systems and services. Initially only used to observe manually the ways a broken-into machine is put to use, they are increasingly being leveraged as oracles for malice. This follows from the fact that activity on honeypots can be considered *suspicious* by definition, as they serve no purpose in benign interaction. Ideally, they should never see any traffic. Unfortunately, honeypot activity cannot automatically be considered *malicious*, since the Internet today carries a substantial amount of largely benign "background radiation" consisting of backscatter, misconfigurations, and a large array of broken traffic whose genesis is barely explicable (more on this in Section 5.5). Nevertheless, honeypots currently are among the best network-based implementations of a malice oracle available. The work I present in this section was among the first to recognise the potential of honeypots for automated analysis of malicious traffic and its ideas have been used and extended by a large body of recent work that I will discuss in detail in Section 5.5.

honeypot interaction levels

At present, the creation of exploit signatures for detection of content-based attacks is a tedious, manual process that requires detailed knowledge of both the vulnerability and the attack vectors it is supposed to capture. Simplistic signatures tend to generate large numbers of false positives, too specific ones cause false negatives. To overcome these issues, I have developed *Honeycomb*, a system that generates signatures for malicious network traffic automatically. The hypothesis is that by applying pattern-detection techniques and packet header conformance tests to traffic captured on honeypots, one can identify the elements in those flows characteristic to attacks, and express them in form of content-bases signatures. The purpose of the system was to find out whether such an approach can be made to work and highlight the relative difficulties involved. The system is an extension of `honeyd`, a popular low-interaction open-source honeypot. `honeyd` simulates hosts with individual networking *personalities*. It intercepts traffic sent to nonexistent hosts and uses the simulated systems to respond to this traffic. Each host's personality can be individually configured in terms of OS type (as far as detectable by common fingerprinting tools) and running network services (termed *subsystems*).

automated signature generation

My implementation spots patterns in traffic previously seen on the honeypot: parts of flows in the traffic are aligned and compared, and the resulting commonalities are one of the input streams for the signatures the system generates. My original implementation used a suffix-tree based LCR algorithm as introduced in Section 3.5.1 to spot similarities in the payloads. Recall that the suffix tree implementation allows computation of the LCR in linear time; among the several algorithms that have been proposed to build suitable suffix trees [167, 101, 154] I used my implementation of Ukkonen's algorithm as provided by the `libstree` library previously mentioned in Section 3.5.1.

Honeycomb's source code has been publically available from the outset[1] and the system remains the only automatic signature generator with available source code to date.

5.3.1 Architecture

I have added two new concepts to `honeyd`: a plugin infrastructure, and event callback hooks. The plugin infrastructure allows the development of extensions that remain logically separated from the `honeyd` codebase, while the event hooks provide a mechanism to integrate the plugins into the activities inside the honeypot. Event hooks allow a plugin to be informed when packets are received and sent, when data is passed to and received from the subsystems and to receive updates about `honeyd`'s connection state. Honeycomb is implemented as a `honeyd` plugin. Figure 5.1 illustrates the architecture.

Integrating the system into `honeyd` has several advantages over implementing a standalone *bump-in-the-wire* design from scratch:

- No duplication of effort: the system needs access to network traffic. For a standalone application, `libpcap`[100] would be an obvious choice. `honeyd` already does this: it inspects the network traffic using `libpcap` and passes the relevant packets to the network stacks of the simulated hosts and eventually to their configured subsystems. My approach is a minimum-effort solution that avoids performance hits by making use of packet data already transferred to userspace.

[1] See `http://www.cl.cam.ac.uk/~cpk25/honeycomb/` for details.

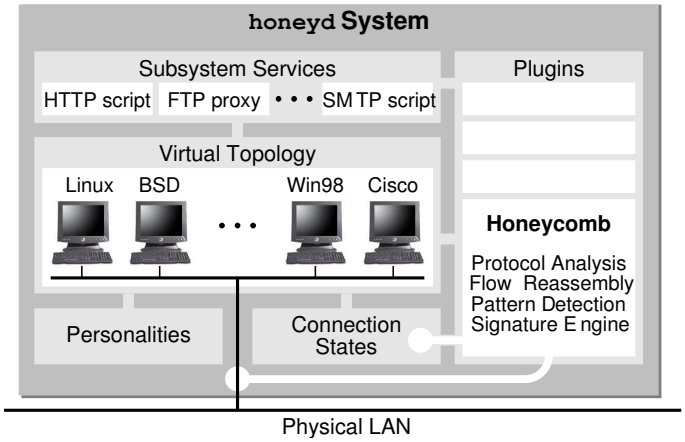


Figure 5.1. Honeycomb's architecture, illustrated as a typical `honeyd` setup. `honeyd` is simulating a number of different machines, each running a number of pre-configured services. The Honeycomb plugin has hooked itself into the wire to see in- and outgoing connections, and into `honeyd`'s connection state management.

- Sufficiently realistic response traffic: `honeyd` is not passively listening to traffic going in and out of the honeypot, rather, it actively *creates* the traffic coming out of it through the simulated network stacks and the configured subsystems. This creation of traffic is readily configurable and fully under the control of the user; an advantage when experimenting with signature generation compared to more faithful virtualisation environments.
- Avoidance of cold-start issues: a major advantage from the state management perspective lies in the fact that integrating Honeycomb into `honeyd` avoids desynchronisation from the current state of connections: when `honeyd` receives a packet that starts a new connection (whether in a legal fashion or not), Honeycomb *knows* that this starts the connection. The question whether it may have missed the beginning of the connection is a non-issue, in contrast to other systems that use the bump-in-the-wire approach[71, 125].

5.3.1.1 *Signature Creation Algorithm*

The philosophy behind the approach is to keep the system free of any knowledge specific to certain application layer protocols; Honeycomb's operation should be fully *protocol-agnostic*. Thus, each received packet

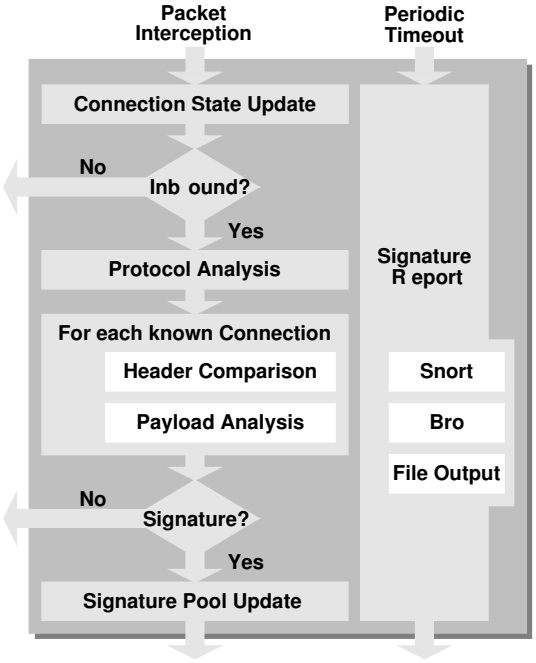


Figure 5.2. High-level overview of Honeycomb's signature creation algorithm.

causes Honeycomb to initiate the same sequence of activities:

- If there is existing connection state for the new packet, that state is updated, otherwise new state is created. At the same time, the timestamp of the new packet is used to potentially time out and expunge outdated connection state.
- If the packet is outbound, processing stops at this point.
- Honeycomb performs protocol header field analysis at the network and transport layers.
- For each stored connection state:
  - Honeycomb performs protocol header comparison in order to detect matching IP networks, initial TCP sequence numbers, etc.
  - If the connections have the same destination port, Honeycomb attempts pattern detection on the exchanged flow content.
- If no useful signature was created in the previous step, processing

stops. Otherwise, the signature is used to augment the *signature pool* as described in Section 5.3.1.5.

- Periodically, the signature pool is logged in a configurable manner, for example by appending the Bro representation of the signatures to a file on disk.

Figure 5.2 illustrates the algorithm. Each activity is explained in more detail in the following sections.

5.3.1.2 *Connection Tracking*

Honeycomb maintains state for a limited number of TCP and UDP connections, but has rather unique requirements concerning connection state-keeping. Since the aim is to generate signatures by comparing new traffic in the honeypot to flows seen previously, it cannot release all connection state immediately when a connection is terminated. Instead, Honeycomb only marks connections as terminated but keeps them around as long as possible, or until it can be sure that there is no benefit in storing them any longer.

Connections that have exchanged lots of information are potentially more valuable for detecting matches with new traffic. The system must prevent aggressive port scans from overflowing the connection hashtables which would cause the valuable connections to be dropped. Therefore, both UDP and TCP connections are stored in a two-stage fashion: Connections are at first stored in a "handshake" table and move to an "established" table when actual payload is exchanged. In this manner, high-rate connection attempts cannot cause the more valuable established-connection states to be dropped.

The system performs flow reassembly and message extraction as described in Section 3.3: for TCP, Honeycomb reassembles flows up to a configurable total maximum of bytes exchanged in the connection. It stores the reassembled stream as a list of exchanged messages up to a maximum allowed size, where a message is all the payload data that was transmitted in one direction without any payload (i.e., at most pure ACKs) going the other way. For example, a typical HTTP request is stored as two messages: one for the HTTP request and one for the HTTP reply. For UDP, messages are similarly created for all payload data going in one direction without

payload data going the other way. Figure 5.3 illustrates the idea.

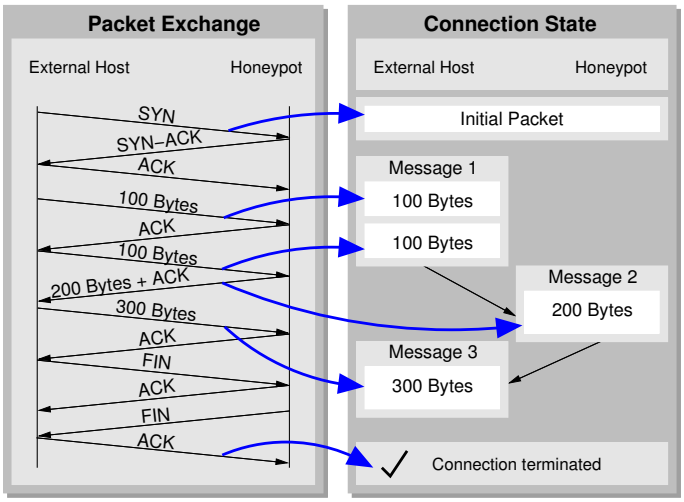


Figure 5.3. A TCP packet exchange (left) and the way Honeycomb traces the connection (right). The packet initiating the connection is copied separately; afterwards, two 100-Byte payloads are received and assembled as one message. 200 Bytes follow in response, forming a new message. This in turn is answered by another 300 Bytes, forming the final message. The successful completion of the TCP teardown triggers the labelling of the connection as "terminated".

5.3.1.3 *Protocol Analysis*

After updating connection state, Honeycomb creates an empty signature record for the flow and starts inspecting the packet. Each signature record has a unique identifier and stores discovered *facts* (i.e., characteristic properties) about the currently investigated traffic independently of any particular NIDS signature language. The signature record is then augmented continuously throughout the detection process, maintaining a count of the number of facts recorded[2].

Honeycomb performs protocol analysis at the network and transport layers for IP, TCP and UDP packet headers, using the header-walking technique previously used in traffic normalisation [71]. Instead of correcting detected anomalies, it records them in the signature, for example invalid

[2]The terms "signature record" and "signature" are used interchangeably here except for cases when I want to stress the difference between a signature record and a NIDS-specific signature string produced from the record.

IP fragmentation offsets or unusual TCP flag combinations. Note that for these checks, Honeycomb does not need to perform any comparison to previously seen packets. We refer to a signature at this point as the *analysis signature*.

Honeycomb then performs header comparison with each currently stored connection of the same type (TCP or UDP). If the stored connection has already moved to the second-level hashtable, Honeycomb tries to look up the corresponding message and uses the headers associated with that message. If no such message can be found, the next connection is investigated. If the connection is still in the first-level hashtable, the initial packet is used for the comparison.

If any overlaps are detected (e.g., matching IP identifiers or address ranges), the analysis signature is cloned and becomes specific to the currently compared flows. The discovered facts are then recorded in the new signature.

5.3.1.4 *Pattern Detection in Flow Content*

After protocol analysis, Honeycomb proceeds to the analysis of the reassembled flow content. Honeycomb applies the LCR algorithm to binary strings built out of the exchanged messages. It does this in two different ways, illustrated in Figures 5.4 and 5.5.

- Horizontal Detection: Assume that the number of messages in the current connection after the connection state update is n. Honeycomb then attempts pattern detection on the nth messages of all currently stored connections with the same destination port at the honeypot by applying the LCR algorithm to the payload strings directly.
- Vertical Detection: Honeycomb also concatenates incoming messages of an individual connection up to a configurable maximum number of bytes and feeds the concatenated messages of two different connections to the LCR algorithm. The point here is that horizontal detection will fail to extract meaningful messages from interactive sessions like Telnet and thus won't be able to detect meaningful flow commonalities, whereas vertical detection will still work. Furthermore, vertical detection is also guaranteed to *mask* directional TCP dynamics, since the concatenation effectively recovers the reassembled streams per direction.

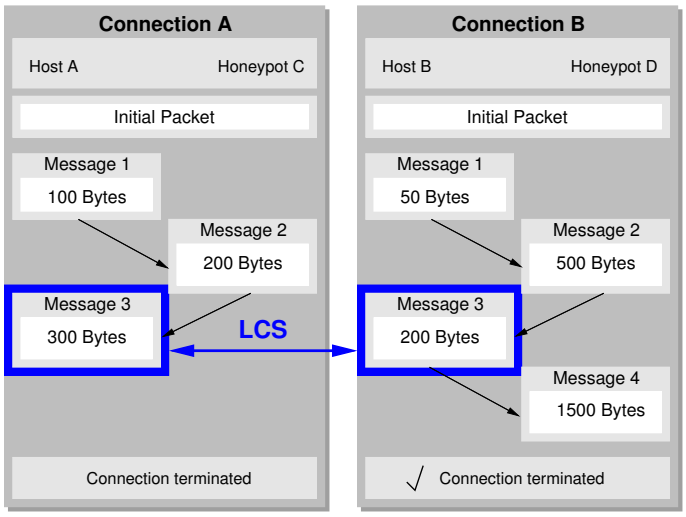


Figure 5.4. Horizontal pattern detection: two messages at the same depth into the stream are passed as input to the LCR algorithm for detection.

In either case, if a common substring is found that exceeds a configurable minimum length, the substring is added to the signature as a new payload byte pattern.

5.3.1.5 *Signature Lifecycle*

If the signature record contains no facts at this point, processing of the current packet ends. Otherwise, Honeycomb checks hows the signature can be used to improve the signature pool, which represents the recent history of detected signatures.

The signature pool is implemented as a queue with configurable maximum size; once more signatures are detected than can be stored in the pool, old ones are dropped. Dropped signatures are not lost, since the contents of the signature pool are reported in regular intervals (see Section 5.3.1.6).

Honeycomb tries to reduce the number of reported signatures as much as possible by performing *signature aggregation*. I have defined two relational operators for the generated signatures for this purpose:

- $sig_1 = sig_2$: signature identity. This operator evaluates to true when sig_1 and sig_2 match in all attributes except those which can be expressed as lists in resulting signatures (e.g., ephemeral source port

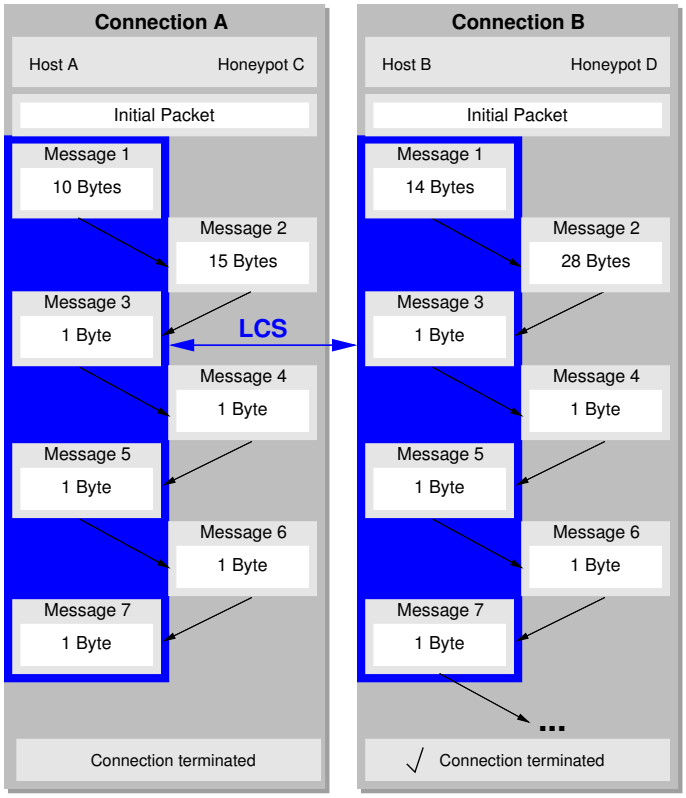


Figure 5.5. Vertical pattern detection: for both connections, several incoming messages are concatenated into one string and then passed as input to the LCR algorithm for detection.

numbers). An example would be a simple SYN portscan that is not IP source spoofed: the incoming packets share common source IP addresses and TCP SYN flags, but the destination ports vary.

- $sig_1 \subset sig_2$: signature sig_1 defines only a subset of sig_2's facts. This particularly includes any payload patterns detected by the LCR algorithm: A byte sequence b_1 is considered weaker than b_2 when b_1 is a substring of b_2.

If a new signature is a superset of an existing one, the new signature improves the old one, otherwise the new signature is added to the pool as a new entry.

5.3.1.6 *Signature Output*

The contents of the signature pool are periodically reported to an output module which implements the actual logging of the signature records. At the moment, there are modules that convert the signature records into Bro or pseudo-Snort format,[3] and a module that dumps the signature strings to a file.

The periodic reporting scheme is an easy way to make sure all signatures are reported while in the signature pool and also allows for tracking of the evolution of signature records through the signature identifier in a post-processing stage.

5.3.2 Evaluation

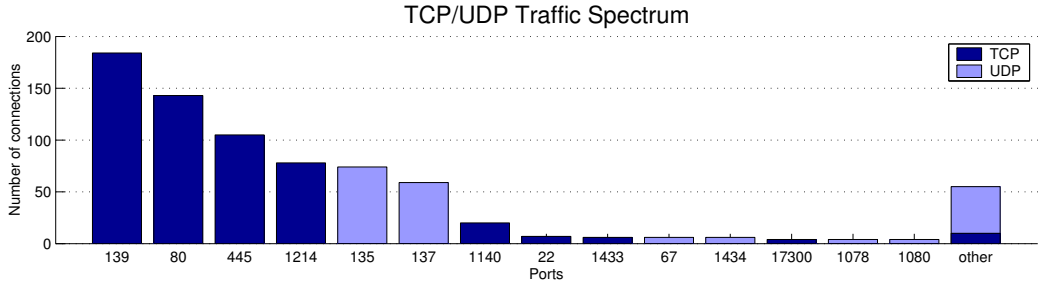


Figure 5.6. Distribution of TCP and UDP traffic destination ports in packets directed at the honeypot, as observed in the 24 hours.

The implementation consists of roughly 9000 lines of C code, with about 3000 lines for the `libstree` library. I tested system on an unfiltered cable modem connection in three consecutive sessions, covering a total period of three days. I was particularly interested in the traffic patterns and signatures created for a typical home-user connection, which can be assumed to be often only weakly protected, if at all. Furthermore, a larger honeynet that would potentially see higher traffic volumes was unavailable at the time.

[3]Honeycomb requires the ability to define a list of non-contiguous ports, and Snort's signature language currently does not permit this.

5.3.2.1 *Traffic Characteristics*

During the 24-hour period, `honeyd` captured 224 KB of traffic, comprising 557 TCP connections, 145 UDP connections and 27 ICMP pings. Figure 5.6 shows the distribution of the ports requested at the honeypot, in terms of numbers of connections.

5.3.2.2 *Signature Detection*

Honeycomb created 38 signatures for hosts that just probed common ports. 25 signatures were created containing flow content strings. These are relatively long; on average they contain 136 bytes. The longest strings are those describing worms: Honeycomb managed to create precise signatures for the Slammer and CodeRed II worms, see Figures 5.7 and 5.8. Note that these are the overlaps after 6 hits of Slammer and 3 CodeRed II hits, so they are *not* just accidental long matches between single pairs of flows, and are thus more reliable than a match found in only two flows. Also, Honeycomb did not report the typical HTTP GET string for CodeRed II; rather, it reported the longer string of bytes following afterwards.

```
alert udp any any -> 192.168.169.2/32 1434 (msg: "Honeycomb Fri Jul 18 11h46m33 2003 "; content: "|04 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 DC C9 B0|B|EB 0E 01 01 01 01 01 01 01|p|AE|B |01|p|AE|B|90 90 90 90 90 90 90 90|h
|DC C9 B0|B|B8 01 01 01 01|1|C9 B1 18|P|E2 FD|5 |01 01 01 05|P|89 E5|Qh.dllhel32hkernQhounthickChGetTf|B9|llQh32.dhws2_f
|B9|etQhsockf|B9|toQhsend|BE 18 10 AE|B|8D|E|D4|P|FF 16|P|8D|E|E0|P|8D|E|F0|P|FF 16|P|BE 10 10 AE|B|8B 1E 8B 03|=U
|8B EC|Qt|05 BE 1C 10 AE|B|FF 16 FF D0|1|C9|QQP|81 F1 03 01 04 9B 81 F1 01 01 01 01|Q|8D|E|CC|P|8B|E|C0|P|FF 16|j|11|
j|02|j|02 FF D0|P|8D|E|C4|P|8B|E|C0|P|FF 16 89 C6 09 DB 81 F3|<a|D9 FF 8B|E|B4 8D 0C|@|8D 14 88 C1 E2 04 01 C2 C1 E2 08|
)|C2 8D 04 90 01 D8 89|E|B4|j|10 8D|E|B0|P1|C9|Qf|81 Fl|x|01|Q|8D|E|03|P|8B|E|AC|P|FF D6 EB|"; )
```

Figure 5.7. Signature Honeycomb created for the Slammer Worm.

```
alert tcp 80.0.0.0/8 any -> 192.168.169.2/32 80 (msg: "Honeycomb Mon May  5 16h59m09 2003 "; flags: A; flow: established;
content: "u|08 81|˜0|9A 02 00 00 0F 84 C4 00 00 00 C7|F0|9A 02 00 00 E8 0A 00 00 00|CodeRedII|00 8B 1C|$|FF|U|D8|f|0B C0
0F 95 85|8|FE FF FF C7 85|P|FE FF FF 01 00 00 00|j|00 8D 85|P|FE FF FF|P|8D 85|8|FE FF FF|P|8B|E|08 FF|p|08 FF 90 84 00
00 00 80 BD|8|FE FF FF 01|thS|FF|U|D4 FF|U|EC 01|E|84|i|BD|T|FE FF FF|,|01 00 00 81 C7|,|01 00 00 E8 D2 04 00 00 F7 D0
0F AF C7 89|F4|8D|E|88|Pj|00 FF|u|08 E8 05 00 00 00 E9 01 FF FF FF|j|00|j|00 FF|U|F0|P|FF|U|D0|Ou|D2 E8|;|05 00 00|i|BD|
T|FE FF FF 00|\&|05 81 C7 00|\&|05|W|FF|U|E8|j|00|j|16 FF|U|8C|j|FF FF|U|E8 EB F9 8B|F4)E|84|jd|FF|U|E8 8D 85|<|FE FF FF|
P|FF|U|C0 0F B7 85|<|FE FF FF|=|88 88 00 00|s|CF 0F B7 85|>|FE FF FF 83 F8 0A|s|C3|f|C7 85|p|FF FF FF 02 00|f|C7 85|r
|FF FF FF 00|P|E8|d|04 00 00 89 9D|t|FF FF FF|j|00|j|01|j|02 FF|U|B8 83 F8 FF|t|F2 89|E|80|j|01|Th˜f|04 80 FF|u|80 FF|U
|A4|Yj|10 8D 85|p|FF FF FF|P|FF|u|80 FF|U|B0 BB 01 00 00 00 0B C0|tK3|DB FF|U|94|=3’|00 00|u?|C7 85|h|FF FF FF 0A 00 00
00 C7 85|l|FF FF FF 00 00 00 00 C7 85|’|FF FF FF 01 00 00 00 8B|E|80 89 85|d|FF FF FF 8D 85|h|FF FF FF|Pj|00 8D 85|’|FF
FF FF|Pj|00|j|01 FF|U|A0 93|j|00|Th˜f|04 80 FF|u|80 FF|U|A4|Y|83 FB 01|ul|E8 00 00 00 00|X-|D3 03 00 00|j|00|h|EA 0E 00
00|P|FF|u|80 FF|U|AC|=|EA 0E 00 00|u|11|j|00|j|01 8D 85|\|FE FF FF|P|FF|u|80 FF|U|A8 FF|u|80 FF|U|B4 E9 E7 FE FF FF BB
00 00 DF|w|81 C3 00 00 01 00 81 FB 00 00 00|xu|05 BB 00 00 F0 BF|’|E8 0E 00 00 00 8B|d$|08|dq|8F|"; )
```

Figure 5.8. Signature Honeycomb created for the CodeRed II Worm.

5.3.2.3 *Performance Overhead*

I measured the performance overhead involved when running Honeycomb compared to normal `honeyd` operation; the results are displayed in Figure 5.9.

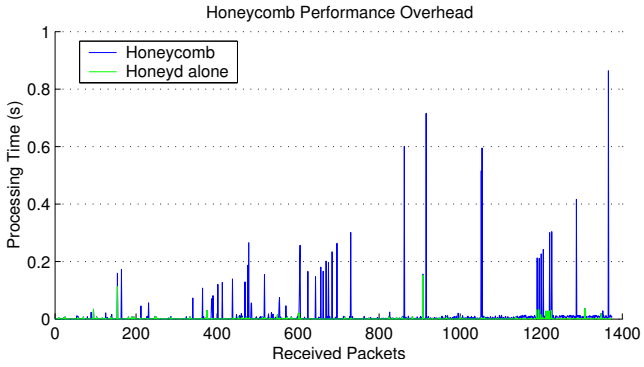


Figure 5.9. Performance overhead when running Honeycomb. The packet processing times are almost entirely dominated by Honeycomb, so the `honeyd` part is hardly visible.

5.3.3 Discussion

Honeycomb was one of the first if not *the* first system to use honeypots for automating the network-based analysis of attacks as they occur. Many lessons have been learned since its conception, both about the way Honeycomb does things as well as about the things Honeycomb did not yet address.

5.3.3.1 *Evasion of Signature Generation Algorithm*

The choice of LCR computation for signature generation is likely a bad one. Assuming sufficient freedom in flow content, all an attacker has to do to evade detection of valuable attack-relevant strings is to place a decoy string of length greater than the attack-relevant ones into the flows, and the LCR computation will not return the desired result. However, as pointed out in Section 3.6.2, LCS algorithms such as Smith-Waterman and Jacobson-Vo, while more appropriate, also need to be used with care. A solid fallback if the performance of Smith-Waterman is sufficient is the use of an ACS-computing variant of Smith-Waterman. It is guaranteed to highlight *all* common substrings between any pair of flows and thus feeds the maximum amount of and most consistent combination of common substrings into the signature generator.

Note however that given the high length of the common substrings reported by the LCR algorithm, it is quite unlikely that sequences of com-

mon substrings computed via some other means than LCR would have omitted the ones reported by LCR. At most, those algorithms would have reported additional common substrings, some of which may not be desirable, as I will discuss next.

5.3.3.2 *Leveraging Application-Layer Protocol Knowledge*

The protocol-agnostic approach employed by Honeycomb is clearly beneficial. Having to encode protocol knowledge would at the very least be tedious (if the protocol specification if known) and in the worst case be impossible (in case the workings of a protocol are unknown). However, the ability to use protocol-specific knowledge *without* understanding of a protocol's exact operation would be highly beneficial to solve the problem of mistaking highly frequent protocol-intrinsic strings for malicious content. The goal here is to automate the discovery of such substrings and treat them accordingly. Such *automated whitelisting* of protocol-intrinsic substring could very likely be provided by the CSGs as presented in Section 4.3 or, since the full flexibility of CSGs is likely not necessary, a simplified version thereof. Note that the goal is generally not going to be to drop such protocol-intrinsic strings altogether, but rather to recognise when signatures would be constructed that consist of *nothing but* such substrings.

automated substring whitelisting

5.3.3.3 *Signature Distillation*

Honeycomb has no signature postprocessing stage taking over once signatures are logged to the output file. Indeed, one of the most common complaints I have received by other users since Honeycomb's source code has been available is that it produces *too many* signatures. Solid *signature distillation*, that is, the on-line recognition of redundantly generated signatures and partial commonalities among sets of generated signatures is a component missing from all systems proposed in the literature to date. Some systems do not consider live operation at all and satisfy themselves with operating on fixed malicious and benign flow pools, though the the main research goal of such systems is the analysis of specific properties of the signature generators, such as false positive rates [94].

It is worth noting that to the best of my knowledge, no other system except Honeycomb has combined protocol header analysis with flow content analysis in the signature generation process. I believe that protocol

header analysis remains valuable since scripted attack tools often exhibit idiosyncrasies in the traffic they generate; these idiosyncrasies stand a good chance of being picked up by the header analysis.

5.3.3.4 *Performance*

The runtime performance evaluation of Honeycomb is preliminary. In my experiments with sequence alignment algorithms in general and flow pools in context of the CSG work in Chapter 4 I have found that it is rarely the runtime requirements of sequence alignment algorithms per se that cause performance bottlenecks, but rather the state-keeping context in which they are applied. For example, while it might be prohibitively slow to run a given algorithm on all pairwise combinations of a new flow message and the corresponding ones in a flow pool, it might very well be feasible to perform the computation on just those message pairs that have the greatest potential to yield substantial common substrings. It is a topic of future work to devise flow pooling and scanning strategies that cleverly use knowledge of the content of flows contained in a pool to reduce the amount of work required.

5.3.3.5 *Automatic Signature Enforcement*

The Holy Grail of automated signature generators is fast distribution and automated enforcement of generated signatures, for example to contain a worm epidemic in its early stages. Such a degree of automation has only been discussed theoretically to date, for example by Moore et al. [109] and Weaver and Paxson [165]. It is generally assumed that automated containment will be difficult to achieve, not only due to the tight timescales required, but also the ideal pervasiveness of the enforcement architecture and the issues of large-scale collaboration, trust issues, etc. It is worth pointing out that automatic signature creation for the purpose of containing an expanding epidemic is different from accumulating exploit-specific signatures in general: in the former case, generated signatures would serve as an aid in an emergency situation but may not be meant as a long-term solution, much like the interim relief character proposed by Wang et al. [161].

Nevertheless, an automated signature creator that captures the "Zeitgeist"

of current malicious activity on the Internet, in the spirit of dshield.org[4] would be highly beneficial to many institutions. On the other hand, the existence of an automated containment infrastructure would open up interesting new questions. For example, given such an infrastructure, the faster malware spreads, the more obvious its activity will be and thus the quicker it will be battled by the containment system. It might thus become more important for malware to operate *stealthily* to avoid early detection.

5.3.3.6 *Detection and Evasion of Honeypot Architectures*

It can be expected that in the foreseeable future only a comparatively small number of institutions will be running large-scale honeyfarms monitoring substantial ranges of the IP address space. This opens up the question of when or whether attackers are taking active steps to evade such honeynets [15]. While currently the likely truth is that attackers do not feel sufficiently threatened by the existing monitoring infrastructures, this might well change in the future. The problem is that big honeyfarms are not *agile*, that is, their address ranges cannot easily be changed. A potential alternative is to "outsource" the collection of unwanted traffic to large numbers of willing participants: by running a tunnelling daemon on such machines that forwards traffic arriving on participating end hosts on ports that are closed (e.g., via GRE), the breadth of vision of a honeyfarm could potentially be extended significantly. On the other hand, such a move causes incentive for the attackers to launch large-scale chaff attacks that feed garbage into the monitoring system. Furthermore, similarly to current malware being able to disable antivirus software upon machine compromise, the malware might learn to disable such tunnelling daemons. Hardware-based virtualisation techniques [37, 76] could solve this problem by sufficiently protecting the integrity of the tunnelling daemon.

Honeypots are also lacking agility in another sense: they can only serve information about exploits for which they serve the matching vulnerabilities. With most software packages receiving updates and patches on a regular basis, it can become a management challenge to provide relevant attack surfaces.

[4] `http://www.dshield.org`

5.4 Curtailing Malicious Traffic with Packet Symmetry

I now move on to an entirely different approach to fingerprinting malice, using the structure of network traffic at a much lower level in the network model. I here present the work I contributed to a collaboration with Andrew Warfield, Jon Crowcroft, and Steven Hand [86].

5.4.1 Packet Asymmetry as a Badness Oracle

The idea of this work is to leverage packet dynamics as a classifier for identifying badness: by monitoring the symmetry ratio of the number of outgoing *packets* to the number of received ones, one obtains a simple yet flexible detector that allows the gradual throttling of sources ranging from individual hosts to larger aggregates proportionally to the asymmetry exhibited by the traffic they generate. A high degree of packet symmetry embeds the notion of mutual consent within a protocol, allowing the receiver to implicitly throttle a sender simply by *not replying*. The next step is to *enforce* symmetry on network transmissions at the edge: a simple enforcement mechanism may be placed in NICs or access providers' line cards, to delay or drop packets that result in strongly asymmetric communications. This edge-near placement makes implementation easy, ensures a clear notion of packet provenance, and cannot be compromised by application or OS exploits on the end-host.

implicit receiver signalling

More formally, I define an *asymmetry metric* $\mathcal{S}$ based on the number of transmitted packets tx and received ones rx as follows:

$$\mathcal{S} = \log_e \left(\frac{tx+1}{rx+1} \right)$$

This metric produces negative values when rx outweighs tx, positive values when tx outweighs rx, and zero in the case of perfectly balanced traffic. The absolute value of $\mathcal{S}$ measures the magnitude of the asymmetry. This metric has been carefully chosen for analysis: it allows an unbiased means of evaluating traffic, centred around zero, and compresses wildly asymmetric traffic ratios into a tractable range. An initial concern was the question whether this metric would be sufficiently sensitive. Both mea-

surement and initial implementation have confirmed that it is indeed very useful to work with.

Given a network vantage point, the value of $\mathcal{S}$ may be calculated over traffic at some granularity (e.g. per-host, per-host-pair, per-flow) and over a window of time. One may then take action against traffic that exceeds some threshold value of $\mathcal{S}$. There are a number of design decisions in this approach that should be mentioned explicitly:

- Measure packets, not bytes. Rather than comparing bytes transmitted in each direction, we simply count packets. With no knowledge of the internals of the data being sent, packets are much more likely to indicate the message structure that exists within a given protocol. Moreover, the implicit signalling to receive more data may be as simple as a TCP ACK, for which byte counts are considerably less useful than packets.

- Measure and limit close to the transmitter. The outcome of the approach we advocate is that the policy of implicit signalling is enforced end-to-end: Receivers are responsible for generating sufficient backpressure on a channel to allow the transmitter to continue sending. Monitoring and enforcement, however, are performed within the network just outside the reach of the transmitting software (e.g. on a smart NIC). There are many reasons for this placement: First, we may clearly establish packet provenance, eliminating the need for traceback [142, 143, 168] or pushback [77, 120]. Second, we eliminate all potential damage done to interior links as well as the target endpoint. Third, we minimise the aggregate amount of state that must be tracked, allowing a simpler implementation. And finally, by mandating that placement be near, but not within the transmitter's software stack, we are robust against exploits which circumvent the OS.

- Delay, then drop. Unlike traditional IP congestion control, we opt to delay, rather than to drop packets. As asymmetry increases beyond a selected threshold, we introduce an increasing delay to the transmission of a queued packet. The intention is to be friendly to protocol congestion control approaches by more gently throttling transmitted packets. Where our approach is implemented in a smart NIC, queue-

ing may be completely deferred to the OS. In a non-local device, for instance in an ISP, we anticipate queueing some number of packets for delayed transmission, and then beginning to drop.

In the remainder of this section I analyse a large trace of Internet traffic to establish how the packet symmetry assumption applies in general. The Hotnets paper [86] presents a prototypical system showing how asymmetry restrictions can be enforced to curtail malicious senders.

5.4.2 Traffic Analysis

In order to determine what sorts of traffic might be considered 'well-behaved', I performed a traffic analysis on a 24 hour packet trace collected at a non-university research institution. The trace captured every packet on the full-duplex Gigabit Ethernet link which connected the institution to the Internet. The trace contains over 573 million packets to/from over 170,000 IP addresses and totalling over 250 gigabytes of data — see Moore and Papagiannaki [104] and our 2003 PAM paper [105] for more details on the trace characteristics and the monitoring infrastructure used.

I have examined the degree of symmetry present in the trace data at several granularities: all traffic from each source host; traffic between host pairs; and finally per-flow traffic. The aim of this analysis has been to determine to what degree our symmetry metric can be used to distinguish well-behaved traffic, and how much state it might be useful to maintain in order to achieve this.

5.4.2.1 *Host Packet Symmetry*

I first examined symmetry from the point of view of each of the 170,000 individual hosts in the trace. I calculated $\mathcal{S}$ for all packets relating to that host at a variety of time scales, from one second up to one day. The intention of this measurement was (*i*) to characterise the ranges of symmetries that are exhibited within the trace, and (*ii*) to determine the timescales at which it is appropriate to consider symmetry. The results are shown in Figure 5.10.

Regardless of the time-scale over which $\mathcal{S}$ is measured, the vast majority of hosts exhibit strongly symmetric traffic ($|\mathcal{S}| \leq 2.0$). The left-hand tails

depict hosts where a considerably larger number of packets were received than transmitted, while the right-hand tails show the opposite. The smallest time-scale (one second) most clearly separates symmetric and asymmetric hosts. Note that the plots for one hour and 24 hour windows completely overlap.

5.4.2.2 *Host-Pair Packet Symmetry*

Next I investigated the level of symmetry observed between the unique pairs of hosts in the trace. I carried this out for the approximately 320,000 pairs in which both source and destination send packets and use a time-scale of 1 minute to calculate $\mathcal{S}$. Figure 5.11 shows the cumulative distribution of $\mathcal{S}$ for all host pairs that exchanged packets in both directions during our observation period. Almost all host pairs maintain extremely strong symmetry in their communications ($|\mathcal{S}| \leq 1.0$), while very few are significantly skewed towards the receiving side (bottom 1%) or the transmitting side (top 3%). I also measured $\mathcal{S}$ for a further 6.8 million host pairs in which only the source sends any packets. This clearly undesired traffic demonstrated symmetry values ranging from 0.69–10.5, although with 99.9% less than 2.0, as is shown in Figure 5.12.

5.4.2.3 *Flow Symmetry*

To investigate the symmetry of traffic within individual flows, I chose to examine separately the sets of TCP and UDP flows within the trace. For increased precision, I calculated symmetry values every second. Figure 5.13 shows the cumulative distribution of the maximum value of $\mathcal{S}$ for the TCP flows in the trace. The use of acknowledgement packets in TCP imposes a degree of symmetry on all flows in the trace; virtually all TCP flows exhibit asymmetry ≤ 1.5 — a ratio of about 4.5 packets to one. Examining the outlying TCP flows reveals a small number of misbehaving (or at least irregularly behaving) flows. Considering UDP flows, Figure 5.14 shows a much broader range of symmetry values. Further examination of the outlying UDP flows reveals a great deal of misconfigured DNS traffic and a considerable number of malicious flows; all of these packets are clearly supposed to be subjected to throttling.

In both sets of flow measurements I examined the effect of ignoring packets at the beginning of the flow to reduce any transient asymmetry present

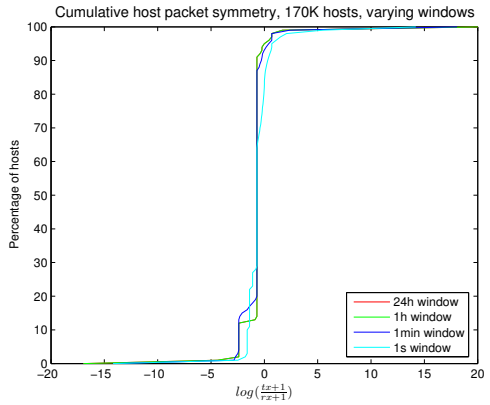


Figure 5.10. Host Symmetry

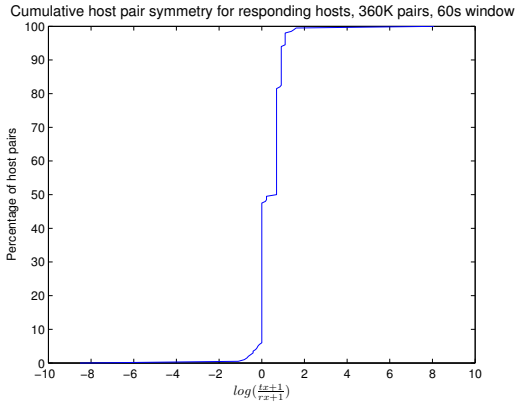


Figure 5.11. Host-Pair Symmetry ($rx > 0$)

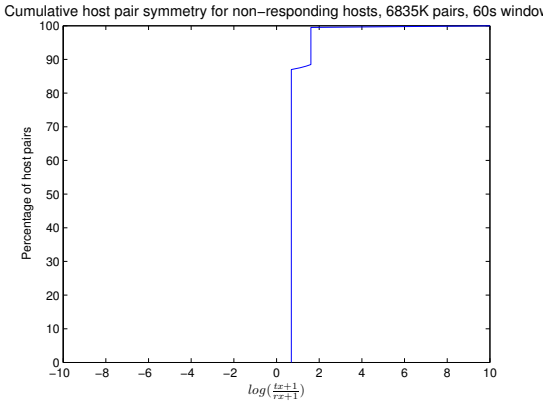


Figure 5.12. Host-Pair Symmetry ($rx = 0$)

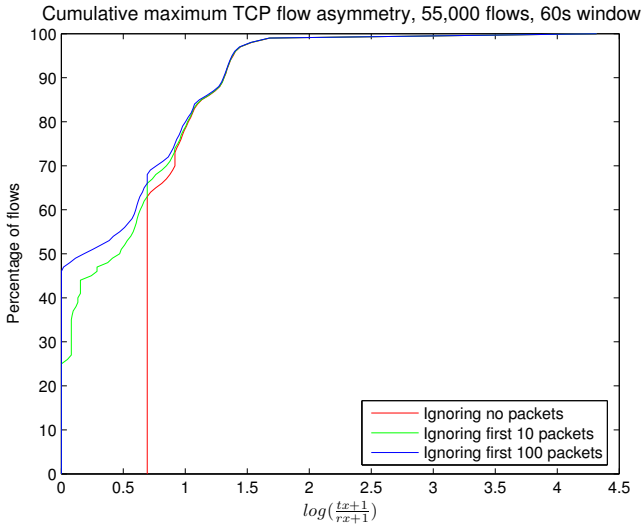


Figure 5.13. Maximum per-flow asymmetry (TCP flows of length > 100)

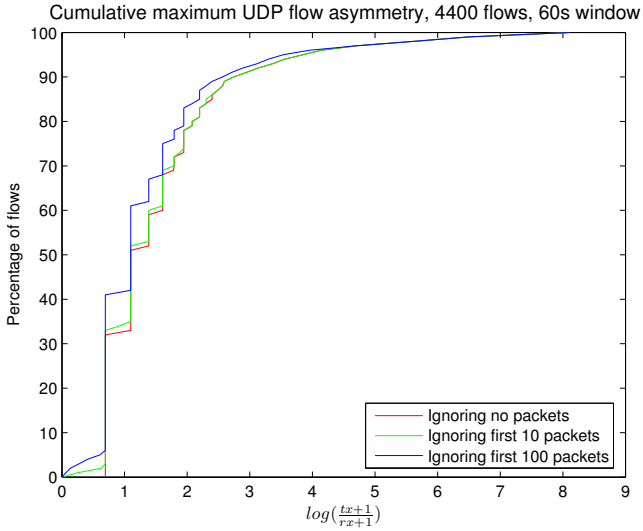


Figure 5.14. Maximum per-flow asymmetry (UDP flows of length > 100)

with low packet counts. Both of the flow-granularity CDFs use only flows from the trace with an excess of 100 packets, and demonstrate that a smoothing effect may be obtained in this manner.

In summary, the analysis shows that packet-level symmetry shows good promise as a classifier between well-behaved and malicious traffic. Furthermore, monitoring per-flow symmetry did not indicate fundamentally

different properties than did host-pair granularity, leading to the belief that the latter is a good starting point for future investigations.

5.4.3 Discussion

5.4.3.1 *Incentives for deployment*

In general, deployment of symmetry enforcement faces an asymmetric incentive problem somewhat reminiscent of the deployment of ingress filtering [62] or the extinction of open SMTP servers or recursive DNS servers. Note however that in many cases there is immediate benefit to sites employing a symmetry shaper. As one example, consider today's server farms, providing enormous CPU and bandwidth resources to essentially anyone, often leaving the site operators in the first line of responsibility in case of abuse. Here, deployment of symmetry enforcement is of immediate value.

5.4.3.2 *Evasion*

There is some potential for attackers to evade the mechanism by tricking it into incorrectly allowing sources to keep sending substantial amounts of undesired traffic. The degree to which this can be avoided depends on the amount of state being used and whether or not source addresses are assumed to be spoofable. In the following, assume host-pair symmetry tracking. The required amount of state should be manageable comparatively easily, particularly given deployment close to the edge, while allowing symmetry enforcement to be protocol-agnostic.

Without help from the outside, individual hosts can only dilute their asymmetric traffic by ensuring a large fraction of symmetric traffic. This is clearly a minor concern. A first distributed strategy is to "fly under the radar" individually, that is, to keep the abusive traffic to a small amount per node, but use substantial amounts of nodes for the attack, i.e., to leverage a botnet. Attackers could, for instance, use a pulsing attack where all nodes attempt to blast away as much traffic as possible, then fall quiet and just as the throttling mechanism permits new transmissions, blast away again. A suitably sensitive delay mechanism would only permit such blasts for a very brief period, as shown by the short time to disruption

of UDP floods in the paper [86].

Another strategy is collusion: during a DDoS attack, the attacker uses the nodes of a botnet to send spoofed cross-traffic amongst its members in order to fool the members' symmetry monitors into thinking that the victim is returning substantial amounts of packets. The feasibility of this approach depends on the following:

- Source address spoofing. As network ingress filtering becomes more pervasive, this will become increasingly hard. Note that the symmetry enforcement mechanism, if widely deployed, would be an ideal opportunity to enforce widespread deployment of ingress filtering.
- Randomisation of IP ID values by the victim. Without this feature, a significant number of colluding nodes have to be informed about the ID value, estimate the IP IDs of the forged packets, and do this quickly enough so forged packets are labelled accurately *and* arrive in a reasonable sequence.
- TTL estimation. Every bot needs to discover by itself the correct TTL value that matches the TTL of the actual victim's reply packets as they arrive at the attacking bot that is colluded with.

While better defence against spoofing-based collusion is conceivable, the IP ID and TTL combination should be difficult enough to overcome for the period until symmetry enforcement is deployed widely. Once that is the case, pervasive enforcement and ingress filtering allow the attacker at best fragile layered asymmetry exploitation to achieve limited amounts of gain in abusive traffic. Furthermore, the approach to reducing botnet effectiveness is aided by an increasing array of other mechanisms highly compatible with symmetry enforcement, such as rigorous checking of reverse path forwarding at the ingress point [10].

5.5 Related Work

The search for a reliable way to identify malicious packets on the Internet is almost as old as the Internet itself and has been iconified humorously by the suggestion of an Evil Bit that would, for some definition of evilness,

unambiguously flag individual packets as either benign or evil [14]. Much work has been done over the years to work towards an Evil Bit equivalent, and I now survey the subset relevant for the structural and statistical approaches presented in this chapter.

5.5.1 Honeypot Architectures

The idea of using honeypots for luring attackers into systems under close surveillance is rather old and predates the invention of the term "honeypot" to express this purpose by decades. Stoll's book [150], dating from 1986, reports on an elaborate hunt for attackers that involved honeypot equivalents. Cheswick's report on his evening with Berferd is another classic tale of honeypot use [36]. Provos introduced `honeyd` [124], the basis for Honeycomb and an excellent tool for experimenting with attacker interaction at medium levels of abstraction. Spitzner's book [146] surveys the honeypot landscape as of 2003. The Honeynet Project [90] aims to develop easily deployable honeypot infrastructures and increasingly focuses on distributed data management and analysis. The project also establishes much dominant terminology. Currently, honeypots serve as a primary tool in the investigation of botnet command and control channels [74, 42]. In the context of this line of work, one of Honeycomb's novelties was the *automation* of deriving conclusions from observations in honeypot environments.

A central component of honeypots is the monitoring of IP address space that is not in production use. Much work dedicates itself to the analysis of just the traffic found on such "dark" address ranges. The primary variable in the literature is the degree to which monitoring systems respond to traffic hitting dark address ranges. The most basic approach is to not respond at all and satisfy oneself with measuring characteristics of the traffic observed. Moore introduced the term "network telescope" for this class of monitors [106, 107], and Pang et al. [116] borrowed from physics the term "background radiation" for the traffic observed on such telescopes. Moore et al. [108] analyse part of the background radiation, namely the *backscatter* caused by victimised hosts as they respond to spoofed source packets, to infer denial-of-service activity. Only slightly more responsive, the Internet Motion Sensor [9] provides the minimum response traffic neces-

dark address space

network telescope

background radiation, backscatter

sary to enable bidirectional communication while focusing on distributed monitoring, while iSink [169] could be termed a misnomer since it does not just absorb traffic but instead provides more targeted yet canned responses for a selection of relevant protocols. Higher still on the interactivity meter rank large-scale *honeyfarms*, server farms consisting entirely of high-interaction honeypots, which focus on the data management of deeply virtualised environments monitoring IP address ranges of several hundreds of thousands of addresses [159].

honeyfarm

5.5.2 Automated Signature Generation

Since Honeycomb, much work on signature generators for malicious traffic has been done. None of these systems are operating publically or providing source code. It is worth noting that Honeycomb was not the first system proposed in the literature to attempt automated malware signature creation. To the best of my knowledge, Kephart and Arnold [82] created the first such system by exposing "goat files" — the file system equivalent of a honeypot — to viruses to study patterns emerging after repeated infections. However, to the best of my knowledge, Honeycomb was the first system to suggest the use of honeypots as a malice oracle for this purpose. Another early piece of work is SigSniffer [70], whose authors suggested the use of Bayesian inference to recommend probable signatures to an analyst, comparing traffic generated by known attack tools to normal one.

goat files

Singh et al. [140] proposed a system called EarlyBird for automated worm fingerprinting. The system uses Rabin fingerprints and manual whitelisting to identify frequent individual common strings without positional accuracy, focusing on the challenges of on-line operation of a single monitor. Simultaneously to EarlyBird, Kim and Karp [83] introduced Autograph. Both systems have a number of parallels. Like EarlyBird, Autograph uses Rabin fingerprints to identify prevalent content. Unlike EarlyBird, Autograph uses benign and malicious flow pools and scanning activity as a badness oracle. Similarly to Honeycomb, signature generation is initiated explicitly every n minutes. The authors also discuss a distributed component for sharing monitoring information. Interestingly, it communicates only badness oracle information and *not* generated signature components.

content sifting

With Polygraph [114], Kim and Karp improved the quality of generated

signatures. Instead of individual frequent substrings, LCSs are computed using Smith-Waterman. While the generated signatures make no use of positional information of the common substrings, the goal of the system is to produce signatures that can fingerprint the invariant bits of polymorphic exploits. Beyond subsequence-based signatures they also introduce Bayes signatures, which probabilistically classify flows according to their likelihood of being malicious given the token set they exhibit. The sample signatures the authors give illustrate the problem of Smith-Waterman identifying protocol-intrinsic signatures alongside exploit-specific ones. The authors propose manual whitelisting to overcome this hurdle.

Instead of trying to stay protocol-agnostic to enable uniform treatment of a wide range of traffic, Nemean [170] embraces the opposite and favours semantic awareness of the protocols it analyses. While this permits the attribution of different value to commonalities at different points in the flows, the obvious disadvantage of this approach is the fact that only supported protocols can be processed. The system is comparatively complex to the point of using machine learning techniques and heuristic search algorithms, partially due to the fact that it attempts to produce signatures at the session level as well. The Hamsa system by Li et al. [94] focuses on guaranteeing bounds on the numbers of false positives produced by generated signatures. The proposed system is otherwise highly similar to Polygraph. Hamsa assumes the existence of classifiers separating flows into benign and malicious flow pools, and bases its evaluation on a single set of flow pools with static content.

Tang and Chen [153] use byte product distributions similar to the ones described in Section 4.4.4 used for protocol classification. They let product distributions take the role of common substrings, arguing that the product distributions will better capture changes in polymorphic exploits. The method grows complex, employing both simulated annealing and an EM algorithm to compute the signature. Furthermore, since there is little reason why a polymorphic engine is restricted to mutations of similar byte distribution, it remains to be seen whether this approach is actually superior to common subsequences.

Some effort has been made to formalise the treatment of the roles played by different parts of network flows carrying exploits. Crandall et al. [44]

distinguish between the exploit vector itself, irrelevant control data, and the remaining payload. Beyond making the distinction between these parts explicit, there seems to be little added value in formalisation at this level, though it could help to establish common nomenclature.

Another line of work attempts host-based signature creation instead of the network-based approaches cited above. An immediate benefit of this approach is the irrelevance of the network-based presentation of an exploit. On the other hand, the signatures are less generally deployable and require equally host-based enforcement. The underlying idea of these approaches is *taint tracking*, i.e., the tracking of data read from the network through the execution of the monitored application [43, 113]. Liang and Sekar [95] attempt to generate vulnerability-specific signatures by analysing the memory images of corrupted C/C++ programs. Brumley et al. [20] generalise the approach and investigate the properties required of signatures to be vulnerability-specific more methodically.

taint tracking

5.5.3 Detection and Mitigation of Volume-based Attacks

A problem related to detection of volume-based attacks is the detection of ongoing changes in traffic volume caused not necessarily by denial-of-service attacks but possibly also other *heavy hitters* such as flash crowds, utilisation surges due to hardware failures, etc. A number of methods have been suggested for detecting such phenomena. One avenue inspired by work in the database community uses streaming algorithms, in which a data model is continuously updated incrementally by each new datapoint observed. [87] adapt a probabilistic summary and forecasting technique known as *sketches* [35] for this purpose. Schweller et al. [136] extend the approach to be reversible and allow the lookup of individual key values to identify culprits. Another approach is again the use of Bloom filters [89]. I previously mentioned them in context of detecting common content in 4. The benefit of these approaches is their suitability for non-edge deployments, however in contrast to packet symmetry they are not immediately useful for intervention. Section 5.5.1 already mentioned another strategy for detection, namely the detection of backscatter in network telescopes [108]. Gil and Poleto [67] described MULTOPS, a router design that used an approach similar to packet symmetry; their work is focused more

heavy hitters

streaming algorithms

sketches

on practical router data structures than on traffic analysis.

Volume-based attack mitigation and prevention has also undergone a considerable amount of research. Mitigation strategies revolve around identifying attack signatures, generally at the network level, and pushing filtering/tracing information back into the core, closer to the many attack sources [77, 120, 131, 144, 142, 143, 168]. These approaches are generally reactive and quite complex, and require updates to the Internet core. *Preventive* measures have focused on enabling the destination to control the sources' capability to communicate with the destination at all [5]. Packet symmetry is again a vastly simpler approach, though capabilities have the potential to address a wide range of attacks depending on the capability hand-out policies implemented. However, capabilities have been shown to be very hard to implement without rendering the capability-issuing channel susceptible to denial of service itself [7]. A radical idea posing many questions is the idea of asking legitimate clients to "compete" with the attackers [160].

traceback,pushback

capabilities

The work perhaps most closely related to packet-level symmetry is the D-WARD proposal [102] and the MANAnet Reverse Firewall [46]. Both propose throttling attack traffic close to the source, although they focus on byte- rather than packet-level metrics, and use more involved algorithms requiring additional state and computation. Moreover, the packet symmetry approach does not require access to payload content.

5.6 Summary

This chapter has introduced novel approaches for fingerprinting the two dominant kinds of malicious activity present on the Internet to date: malicious content of individual network flows on the one side, and abusive volumes of traffic as used in denial-of-service attacks on the other. I introduced the idea of using honeypots not just for satisfying the observer's curiosity but for automated analysis of attacks as they occur, and presented automated content-based signature generation as an example of such an application. The prototype I have developed operates at both the protocol header and payload content levels and generates such signatures

by applying sequence alignment algorithms as presented in Chapter 3 to pairs of network flows. My experiments show that the system can generate highly accurate fingerprints of previously unknown types of attacks in a fully automated fashion. I then changed the focus to packet-level analysis and volume-based attacks, and introduced the notion of packet symmetry as a way to fingerprint and enforce benign behaviour at the network edge. By preventing end systems from transmitting drastically more packets than they receive, a wide range of denial-of-service attacks can be prevented. I have introduced a simple metric to measure this symmetry and have demonstrated through trace-based analysis of real-world network traffic that such enforcement is highly unlikely to affect legitimate applications.

6

Conclusion

"That was not flying, that was falling in style."
— Woody in *Toy Story II*.

Nearing the end of the dissertation, I can now confirm the thesis stated in the Introduction:

> *Network traffic exhibits structural properties which, given suitable filtering and vantage points, permit fully automated derivation of fingerprints of previously unknown network applications and attacks. The generated fingerprints enable accurate detection as well as filtering of such network activity.*

In this dissertation I have examined structural properties of network traffic at two levels of abstraction: application-layer flow content, and packet-level transmission statistics. I discuss them in turn.

In Chapter 3 I discussed sequence alignment algorithms and their applicability to network traffic, including the consequences of the adversarial network security environment. The algorithms I presented extract commonalities among sets of network flows in a number of ways with different degrees of flexibility and performance. I introduced a novel variant of the Jacobson-Vo algorithm that allows flexible selection of LCSs, borrowing dynamic programming concepts from Smith-Waterman. In my experiments it outperformed Smith-Waterman by a factor of 33 on average and 58.5 in the best case. I then demonstrated the suitability of sequence alignment algorithms for fingerprinting the application-layer protocols in Chapter 4: I introduced Common Substring Graphs (CSGs) as a means of fingerprinting application-layer protocols and demonstrated their flexibility through the use in a fully unsupervised framework for application-layer protocol classification, where they provided the best balance between flexibility and accuracy when compared to two other proto-

col models. In Section 5.3 I leveraged an observation about traffic intent: by focusing the monitoring on machines attached to unused IP address space, so-called honeypots, the likelihood of observing malicious traffic is increased dramatically, and I have demonstrated in an experimental prototype the feasibility of extracting content-based malware signatures from such traffic without the need for human intervention.

In Section 5.4 I investigated a second structural property: the ratio of transmitted to received packets, packet symmetry, measured at the network edge. The underlying observation is that well-behaved applications do not send large amounts of packets without receiving any, turning packet asymmetry highly skewed toward to transmitting side into a fingerprint of malicious activity. Indeed, a large class of denial-of-service attacks, one of the greatest threats on the Internet today, operate by blasting traffic at a single destination from as many hosts as possible. By throttling such traffic proportionally to its packet-level asymmetry, such attacks are preventable. I proposed a symmetry metric to measure symmetry compliance of end systems and presented a measurement study confirming that well-behaved traffic is indeed highly symmetric at the packet level. In contrast to the content-based approaches presented earlier, the focus here is not on automated generation of such a fingerprint but on the feasibility of finding *universal* badness fingerprints.

I now conclude the dissertation by summarising potential avenues for future work and by discussing my work in the more general context of a classic design philosophy of the Internet, the end-to-end principle.

6.1 Future Work

Sequence analysis of network traffic in general and the investigation in an adversarial setting in particular is a young field. A number of heuristic alignment algorithms exist which favour speed over accuracy (such as BLAST [2] and FASTA [97]) whose usefulness in the network traffic context has not yet been investigated. On a more elementary level, substitution/alignment scoring schemes are highly flexible and their use in different application settings has not yet been investigated. For example, a scoring scheme that is aware of the keyboard layout could highlight ty-

pographical errors in domain names more generally than the more fixed approaches that have been used to date [162]. I briefly mentioned secondary applications of sequence alignment, such as phylogenetic trees, in Section 3.7.2 in the context of highlighting the relationships between different yet related implementations of the IRC protocol as command and control channels for botnets.

phylogenetic trees

Fingerprinting of both the normal as well as the malicious as presented in Chapters 4 and 5 uses flow pools. The dynamic properties of flows stored in such pools, such as variability of frequent content over time has not yet been investigated but could significantly affect the design of related systems such as selective sampling methods of live traffic. On a toolchain level, there exist no tools at present that use any of the suggested smart traffic classification schemes routinely, though work on this is underway in the IDS domain in Bro [57].

Automated signature generators still offer many avenues for future work. They have yet to prove their value in day-to-day operations. The dominant technical issues concern scalable signature lifecycle management, for example the robust handling of highly related signatures, or capturing the evolving quality of the generated signatures over time. Operational experience to date is lacking, and it is unclear whether all kinds of applications are equally suitable for signature generation. It remains an open question just how close one can get to the goal of *large-scale* automated enforcement of generated signatures.

Given the many ways in which one can express a traffic signature, distributing traffic fingerprinting information still poses many questions. In particular, work needs to be done to determine in what format, at what granularity, and what frequency fingerprinting information does need to be communicated to be generally usable. Incorporating different sites' policies regarding the use of received information and what information one is willing to share poses further challenges. Finally, it remains to be seen whether such a system can operate in an open, federated fashion or whether trust issues and information leakage [15] are issues too fundamental to permit this.

6.2 End-to-End Considerations

Historically, one of the most vigorously defended principles of the Internet architecture is the end-to-end principle [129]. This principle argues for a "dumb" network in which the lower protocol layers handle only the absolutely necessary tasks for enabling end-to-end communication, leaving complexity to a "smart" network edge whenever possible. It has been argued that this principle is essential to unhampered growth and innovation of the Internet [19].

Several elements of this dissertation undermine the end-to-end principle. Therefore, it behoves me to include a discussion of my work from that perspective. Reactive, autonomous filtering components such as automated signature and packet symmetry enforcement as presented in Chapter 5 can principally affect a wide range of network traffic, and the further they are deployed from the edge, the more they collide with the idea of a simple, transparent bit delivery service.

The concern is less of an issue for packet symmetry enforcement, since a central part of this idea is deployment at the very edge and we have yet to witness any application that truly *has* to use highly asymmetric traffic. Automated signature enforcement, however, could potentially occur anywhere in the network. Several arguments relativise the concerns. First, in-core signature enforcement should not be considered an always-on mechanism but rather an emergency response to a critical situation, much in the way Wang et al. [161] proposed Shield not as a solution to the insecure software problem but an interim responsive means to prevent damage until the proper fix has been applied. Second, signature enforcement is not fundamentally different from previously proposed in-core filtering techniques such as pushback [77, 120] (albeit occurring at a semantically higher level), or ISP-based filtering of unsolicited email. Moreover, and most fundamentally, widespread deployment of technology addressing present-day needs already constitutes a substantial de-facto erosion of the principle. Examples of this erosion have frequently grown out of considerations of network security, since relying on end hosts for security-critical functionality implies capable, benign cooperation that is no longer a given. Examples of such end-to-end violations include:

- NATs: network address translation has become pervasive in the IPv4 world, addressing two problems: firstly, mapping a local host population to a typically smaller set of externally visible IP addresses, and secondly, host protection from unsolicited external access, since a NAT cannot be traversed from the outside without third-party negotiation or prior establishment of state in the NAT. NATs break the end-to-end principle since they break unique global addressing and reachability from within the network.
- Firewalls: typically configured statically, firewalls break the end-to-end principle because end hosts have no unambiguous way of learning of the firewall configuration. Some firewalls might return an ICMP port unreachable message, while others might silently drop connection attempts. Nevertheless, today firewalls are deployed pervasively because they serve a clear network security purpose: they prevent undesired traffic from entering a network.
- Content filtering: at the application layer, content filtering is pervasive. Even novice users typically know they need to install malware scanners on their machines for better protection from malicious agents on the network. Proxy servers can suppress content while leaving the network flows otherwise unaffected. Reactive IDSs, typically termed intrusion prevention systems (IPSs), may similarly filter network activity. Often the user remains uninformed about such filtering activity.

I do not mean to justify the erosion of the end-to-end principle by the fact that it has happened as a consequence of addressing the more immediate problems the Internet is facing. While openness, global connectivity, and a generally "well-lit" Internet are preferable to an Internet of walls and minefields, I do believe that in cases such as the ones listed above, we have to make a decision: we can either leave all non-elementary function out of the core and suffer from the malicious reality on today's Internet, or we can violate the end-to-end principle in certain aspects as long as the overall well-being of the Internet is enhanced. The challenge is to find a precise enough yet general and enforceable definition of malicious behaviour. The techniques presented in this dissertation are one step toward a better classifiability of end-host behaviour along this dimension.

A

Code

A.1 Bro Policy for Message Extraction

The policy given in this section implementes message extraction as described in Section 3.3. It is included in Bro version 1.1.51 and newer.

adu.bro

```
1  @load conn-id
2
3  module adu;
4
5  # Generated events:
6  #
7  # - adu_tx(c: connection, a: adu_state) reports an ADU seen from
8  #   c's originator to its responder.
9  #
10 # - adu_rx(c: connection, a: adu_state) reports an ADU seen from
11 #   c's responder to the originator.
12 #
13 # - adu_done(c: connection) indicates that no more ADUs will be seen
14 #   on connection c. This is useful to know in case your statekeeping
15 #   relies on event connection_state_remove(), which is also used by
16 #   adu.bro.
17
18 # ---- Input configuration -- which ports to look at ----------------
19
20 redef tcp_content_deliver_all_orig = T;
21 redef tcp_content_deliver_all_resp = T;
22 redef udp_content_deliver_all_orig = T;
23 redef udp_content_deliver_all_resp = T;
24
25 export {
26
27 # ---- Constants ----------------------------------------------------
28
29         # The maximum depth in bytes up to which we follow a flow.
30         # This is counting bytes seen in both directions.
31         const adu_conn_max_depth     = 100000 &redef;
32
33         # The maximum message depth that we report.
34         const adu_max_depth          = 3 &redef;
35
```

```
36          # The maximum message size in bytes that we report.
37          const adu_max_size          = 1000 &redef;
38
39          # Whether ADUs are reported beyond content gaps.
40          const adu_gaps_ok           = F &redef;
41
42  # ---- Types ------------------------------------------------------------
43
44          # adu_state records contain the latest ADU and additional flags
45          # showing message direction, depth in the flow, etc.
46          type adu_state: record {
47                  adu: string     &default = ""; # The current ADU.
48                  depth_tx: count &default = 1;  # Msg count (>= 1), orig->resp.
49                  depth_rx: count &default = 1;  # Msg count (>= 1), resp->orig.
50                  seen_tx: count  &default = 0;  # TCP: seqno tracking.
51                  seen_rx: count  &default = 0;  # TCP: seqno tracking.
52                  size: count     &default = 0;  # Size of connection, in bytes.
53                  is_orig: bool   &default = F;  # Whether ADU is orig->resp.
54                  ignore: bool    &default = F;  # Ignore future activity.
55          };
56
57          # Tell the ADU policy that you do not wish to receive further
58          # adu_tx/adu_rx events for a given connection. Other policies
59          # may continue to process the connection.
60          #
61          global adu_skip_further_processing: function(cid: conn_id);
62  }
63
64
65  # ---- Globals ----------------------------------------------------------
66
67  # A global table that tracks each flow's messages.
68  global adu_conns: table[conn_id] of adu_state;
69
70  # ---- Functions --------------------------------------------------------
71
72  function adu_skip_further_processing(cid: conn_id)
73          {
74          if ( cid !in adu_conns )
75                  return;
76
77          adu_conns[cid]$ignore = T;
78          }
79
80  function flow_contents(c: connection, is_orig: bool,
81                         seq: count, contents: string)
82          {
83          local astate: adu_state;
84
85          # Ensure we track the given connection.
86          if ( c$id !in adu_conns )
87                  adu_conns[c$id] = astate;
88          else
89                  astate = adu_conns[c$id];
90
91          # Forget it if we've been asked to ignore.
92          #
93          if ( astate$ignore == T )
94                  return;
95
96          # Don't report if flow is too big.
97          #
```

```
98          if ( astate$size >= adu_conn_max_depth )
99                  return;
100
101         # If we have an assembled message, we may now have something
102         # to report.
103         if ( |astate$adu| > 0 )
104                 {
105                 # If application-layer data flow is switching
106                 # from resp->orig to orig->resp, report the assembled
107                 # message as a received ADU.
108                 if ( is_orig && ! astate$is_orig )
109                         {
110                         event adu_rx(c, copy(astate));
111                         astate$adu = "";
112
113                         if ( ++astate$depth_rx > adu_max_depth )
114                                 skip_further_processing(c$id);
115                         }
116
117                 # If application-layer data flow is switching
118                 # from orig->resp to resp->orig, report the assembled
119                 # message as a transmitted ADU.
120                 #
121                 if ( !is_orig && astate$is_orig )
122                         {
123                         event adu_tx(c, copy(astate));
124                         astate$adu = "";
125
126                         if ( ++astate$depth_tx > adu_max_depth )
127                                 skip_further_processing(c$id);
128                         }
129                 }
130
131         # Check for content gaps. If we identify one, only continue
132         # if user allowed it.
133         #
134         if ( !adu_gaps_ok && seq > 0 )
135                 {
136                 if ( is_orig )
137                         {
138                         if ( seq > astate$seen_tx + 1 )
139                                 return;
140                         else
141                                 astate$seen_tx += |contents|;
142                         }
143                 else
144                         {
145                         if ( seq > astate$seen_rx + 1 )
146                                 return;
147                         else
148                                 astate$seen_rx += |contents|;
149                         }
150                 }
151
152         # Append the contents to the end of the currently
153         # assembled message, if the message hasn't already
154         # reached the maximum size.
155         #
156         if ( |astate$adu| < adu_max_size )
157                 {
158                 astate$adu += contents;
159
```

```
160                 # As a precaution, clip the string to the maximum
161                 # size. A loong content string with astate$adu just
162                 # below its maximum allowed size could exceed that
163                 # limit by a lot.
164                 str_clip(astate$adu, adu_max_size);
165                 }
166
167
168         # Note that this counter is bumped up even if we have
169         # exceeded the maximum size of an individual message.
170         #
171         astate$size += |contents|;
172
173         astate$is_orig = is_orig;
174         }
175
176 # ---- Event Handlers ----------------------------------------------------
177
178 event tcp_contents(c: connection, is_orig: bool,
179                    seq: count, contents: string)
180         {
181         flow_contents(c, is_orig, seq, contents);
182         }
183
184 event udp_contents(u: connection, is_orig: bool, contents: string)
185         {
186         flow_contents(u, is_orig, 0, contents);
187         }
188
189 event connection_state_remove(c: connection)
190         {
191         if ( c$id !in adu_conns )
192                 return;
193
194         local astate = adu_conns[c$id];
195
196         # Forget it if we've been asked to ignore.
197         #
198         if ( astate$ignore == T )
199                 return;
200
201         # Report the remaining data now, if any.
202         #
203         if ( |astate$adu| > 0 ) {
204                 if ( astate$is_orig )
205                         {
206                         if ( astate$depth_tx <= adu_max_depth )
207                                 event adu_tx(c, copy(astate));
208                         }
209                 else
210                         {
211                         if ( astate$depth_rx <= adu_max_depth )
212                                 event adu_rx(c, copy(astate));
213                         }
214         }
215
216         delete adu_conns[c$id];
217         event adu_done(c);
218 }
```

A.2 Improved Jacobson-Vo Algorithm

The following is my implementation of the improved Jacobson-Vo algorithm presented in Section 3.5.5. It will be included in Bro 1.2. Inclusion of standard headers and declarations of the relevant Bro data types are not shown; likewise, straightforward parts of the code are omitted for brevity.

jacobson-vo.h

```
1  #define JV_ALPHABET_SIZE 256
2
3  class JV_Indices {
4  public:
5    typedef list<short> Ind;
6    typedef Ind::iterator IndIt;
7    typedef Ind::const_iterator IndCIt;
8
9    JV_Indices()
10   {
11     for (int i = 0; i < JV_ALPHABET_SIZE; i++)
12       _indices[i]._ind = new Ind();
13
14     _first = 0;
15     memset(_iterations, 0, sizeof(int) * JV_ALPHABET_SIZE);
16   }
17
18   ~JV_Indices()
19   {
20     for (int i = 0; i < JV_ALPHABET_SIZE; i++)
21       delete _indices[i]._ind;
22   }
23
24   void AddUsage(u_char c, int iteration)
25   {
26     IndLink* ind = &_indices[c];
27
28     if (_iterations[c] != iteration)
29       {
30         _iterations[c] = iteration;
31         ind->_usage = 0;
32         ind->_ind->clear();
33       }
34
35     if (ind->_usage == 0)
36       {
37         ind->_next = _first;
38         _first = ind;
39       }
40
41     ind->_usage++;
42   }
43
44   void AddIndex(u_char c, short index, int iteration)
45   {
46     if (_iterations[c] == iteration)
47       _indices[c]._ind->push_front(index);
48   }
49
```

```
50    int GetPiLength()
51    {
52      int len = 0;
53
54      for (IndLink* ind = _first; ind; ind = ind->_next)
55        len += ind->_usage * ind->_ind->size();
56
57      _first = 0;
58      return len;
59    }
60
61    const Ind* GetIndices(u_char c) { return _indices[c]._ind; }
62
63  private:
64
65    struct IndLink {
66      Ind* _ind;
67      int _usage;
68      IndLink* _next;
69    };
70
71    IndLink* _first;
72    IndLink _indices[JV_ALPHABET_SIZE];
73    int  _iterations[JV_ALPHABET_SIZE];
74  };
75
76  static JV_Indices jv_indices;
77
78  struct Node {
79    short _s1_idx;   // s1 index of this node
80    short _s2_idx;   // s2 index of this node
81
82    short _len;      // the node's running substring length
83    short _str_len;  // length of full substring this node extends
84    int _score;      // the node's running score.
85    int _str_score;  // the node's score if following from start of string
86
87    Node* _down;     // next node further down in column
88    Node* _skip;     // next node with smaller s2 index down in column
89    Node* _prev;     // previous node in best alignment
90    Node* _sostr;    // start of string: the location at which we might
91                     // have to adjust the prev pointer.
92  };
93
94  #define NODE_PQ_HEAP_SIZE 2000000
95
96  // A priority implementing a binary max-heap for the
97  // score of a node. Node pointers can be updated anywhere
98  // in the heap in O(1).
99  //
100 class JV_NodePQ {
101 public:
102   JV_NodePQ() : _next(1), _min(INT_MAX), _max(0), _shift(0)
103   { memset(_map, 0, sizeof(int) * NODE_PQ_HEAP_SIZE); }
104
105   void Push(Node* node)
106   {
107     int mapping = _map[node->_score - _shift];
108
109     // If we have an entry for that score, we just adjust
110     // the pointer to the new node and are done.
111     //
```

```
112       if (mapping)
113         {
114           _heap[mapping] = node;
115           return;
116         }
117
118       // Otherwise, we have to add the node to the heap,
119       // and ensure the heap property.
120       //
121       _heap[_next] = node;
122       _map[node->_score - _shift] = _next;
123       BubbleUp(_next);
124
125       _next++;
126
127       if (node->_score < _min)
128         _min = node->_score;
129       if (node->_score > _max)
130         _max = node->_score;
131     }
132
133     Node* Top() { return _next > 1 ? _heap[1] : 0; }
134
135     void Delete(Node *node)
136     {
137       int slot = _map[node->_score - _shift];
138
139       if (_heap[slot] != node)
140           return;
141
142       Swap(slot, _next - 1);
143       _next--;
144       BubbleDown(slot);
145       _map[node->_score - _shift] = 0;
146     }
147
148     int Size() { return _next - 1; }
149
150     void Reset()
151     {
152       Node** ptr = _heap + 1;
153
154       for (int i = 1; i < _next; i++, ptr++)
155         _map[(*ptr)->_score - _shift] = 0;
156
157       _shift = (_min != INT_MAX ? _min : 0);
158       _max = 0; _min = INT_MAX; _next = 1;
159     }
160
161     void FullReset() { _shift = 0; _max = _next = 0; _min = INT_MAX; _next = 1; }
162
163   private:
164
165     inline int Parent(int slot) const { return slot >> 1; }
166     inline int LeftChild(int slot) const { return slot << 1; }
167     inline int RightChild(int slot) const { return (slot << 1) + 1; }
168
169     void Swap(int slot1, int slot2); // Omitted for brevity.
170     void BubbleUp(int slot);         // Omitted for brevity.
171     void BubbleDown(int slot);       // Omitted for brevity.
172
173     int _next;  // next slot currently available in heap array:
```

```
174     int _min;   // smallest slot entered since last reset.
175     int _max;   // largest slot entered since last reset.
176     int _shift; // how far to shift new entries, given minimum
177
178     static Node* _heap[];
179     static int _map[];
180   };
181
182   Node* JV_NodePQ::_heap[NODE_PQ_HEAP_SIZE];
183   int JV_NodePQ::_map[NODE_PQ_HEAP_SIZE];
184
185   JV_NodePQ pq;
186
187   class JV_CoverMatrix {
188   public:
189
190     struct NSet {
191       Node* _bottom; // bottom node in column
192       Node* _top;    // top node in column
193       Node* _class;  // first node in column with current s2 index.
194     };
195
196     struct NWin {
197       Node* _lo;
198       Node* _hi;
199       Node* _ext;
200       Node* _max;
201     };
202
203     JV_CoverMatrix(const BroString *s1, const BroString *s2, int pi_length)
204       : _s1(s1), _s2(s2), _last_col(0), _best_score(0), _best_node(0)
205     {
206       int max_cols = min(s1->Len(),s2->Len());
207
208       // We need one node per Pi element.
209       _nodes = new Node[pi_length];
210
211       // We have at most as many columns as the shorter string is long,
212       // since the LCS can't exceed that length.
213       _nset = new NSet[max_cols];
214
215       memset(_nodes, 0, sizeof(Node) * pi_length);
216       memset(_nset, 0, sizeof(NSet) * max_cols);
217
218       _node = _nodes;
219     }
220
221     ~JV_CoverMatrix()
222     {
223       delete[] _nodes;
224       delete[] _nset;
225     }
226
227    void SetNode(NSet* set, short s2_idx, short s1_idx)
228     {
229       _node->_s1_idx = s1_idx;
230       _node->_s2_idx = s2_idx;
231
232       if (! set->_bottom)
233         {
234           set->_class = _node;
235           set->_top = _node;
```

```
236         }
237       else
238         {
239           if (_node->_s2_idx < set->_class->_s2_idx)
240             {
241               set->_class->_skip = _node;
242               set->_class = _node;
243             }
244
245           set->_bottom->_down = _node;
246         }
247
248       set->_bottom = _node;
249
250       // Adjust the object's node pointer so we use next one next time.
251       _node++;
252     }
253
254     void SetBackpointers()
255     {
256       NSet *set_p = _nset + _last_col;
257       NSet *set = set_p - 1;
258
259       pq.FullReset();
260
261       // In case of an LCS of only a single character, the two-column
262       // parallel scan strategy below won't work and will be skipped.
263       // To get a best node regardless, we just pick the first one in
264       // the last column (which will also be the only column).
265       //
266       _best_node = set_p->_top;
267       _best_score = 0;
268
269       for (int i = _last_col - 1; i >= 0; i--, set--, set_p--)
270         {
271           Node* node;
272           NWin win;
273
274           win._lo = win._hi = win._max = win._ext = set_p->_top;
275           pq.Push(win._lo);
276
277           node = set->_top;
278
279           // Skip down from the start of the column as far
280           // as possible.
281           //
282           while (node->_s2_idx >= win._hi->_s2_idx)
283             node = node->_skip;
284
285           for (set->_top = node; node; node = node->_down)
286             {
287               // Adjust the window, obtaining current maximum.
288               //
289               if (! AdjustWin(node, &win, (i == _last_col - 1)))
290                 break;
291
292               // Now see how this affects our previous-pointer.
293               //
294               node->_prev = win._max;
295               node->_score = node->_str_score = win._max->_score + 1;
296               node->_sostr = node;
297
```

```
298             if (win._ext->_s1_idx - 1 == node->_s1_idx &&
299                 win._ext->_s2_idx - 1 == node->_s2_idx)
300               {
301                 int score = win._ext->_score + win._ext->_len + 1;
302                 node->_str_len = win._ext->_str_len + 1;
303                 node->_str_score = win._ext->_str_score + node->_str_len;
304
305                 // We abuse the skip pointer to point to the previous
306                 // node of the string this node is extending. Skip is only
307                 // needed to find the beginning of the region in this column
308                 // that requires analysis and is no longer needed at this point.
309                 //
310                 node->_skip = win._ext;
311
312                 // Check if the running score improves on the current
313                 // maximum via extending a substring.
314                 //
315                 // Equal here is very important for minimising gaps,
316                 // since we want to prefer extension of an existing
317                 // substring over inserting a gap, even though those
318                 // might have identical score.
319                 //
320                 if (score >= node->_score)
321                   {
322                     node->_len = win._ext->_len + 1;
323                     node->_prev = win._ext;
324                     node->_score = score;
325                     node->_sostr = win._ext->_sostr;
326                   }
327
328                 // If the previous string has chopped off a bit of the
329                 // end of the current string, the full substring's score
330                 // might have surpassed the running one. Compare, and if
331                 // it's better, adjust accordingly.
332                 //
333                 if (node->_str_score > node->_score)
334                   {
335                     node->_prev = win._ext;
336                     node->_sostr->_prev = node->_sostr->_skip;
337                     node->_score = node->_str_score;
338                   }
339               }
340
341             if (node->_score > _best_score)
342               {
343                 _best_score = node->_score;
344                 _best_node = node;
345               }
346           }
347
348         pq.Reset();
349       }
350   }
351
352   // Finds a sequence (i.e., row in the matrix) to add pi_idx
353   // (a value in the Pi sequence) to, using binary search to
354   // maintain O(r log n). We can add an index as long as it's
355   // no larger than the last element in a row.
356   //
357   NSet* FindSequence(int pi_idx, int& upper)
358   {
359     int lower = 0;
```

```
360       int middle;
361       NSet* set = &_nset[upper];
362
363       if (upper != 0 &&
364           set->_bottom->_s2_idx >= pi_idx &&
365           (set-1)->_bottom->_s2_idx < pi_idx)
366         return set;
367
368       while (upper != lower)
369         {
370           middle = (lower + upper) >> 1;
371           set = &_nset[middle];
372
373           if (set->_bottom->_s2_idx < pi_idx)
374             {
375               lower = middle + 1;
376               set++;
377               continue;
378             }
379
380           upper = middle;
381         }
382
383       // If we found the last row and we cannot add pi_idx
384       // to the sequence because pi_idx is too big, we need
385       // to start a new row.
386       //
387       if (upper == _last_col && set->_bottom && set->_bottom->_s2_idx < pi_idx)
388         {
389           set++; upper++; _last_col++;
390         }
391
392       return set;
393     }
394
395     void CollectLCS(SWParams& params, SW_LCS& lcs)
396     {
397       Node *node = _best_node;
398
399       lcs.clear();
400
401       if (! node)
402         return;
403
404       short start_s1 = node->_s1_idx;
405       short start_s2 = node->_s2_idx;
406       short end_s1 = node->_s1_idx;
407       short end_s2 = node->_s2_idx;
408       unsigned short len;
409
410       while (node)
411         {
412           short s1 = node->_s1_idx;
413           short s2 = node->_s2_idx;
414
415           if (s1 > end_s1 + 1 || s2 > end_s2 + 1)
416             {
417               // Check whether we have a gap and if so whether
418               // it's of at least the required minimum string length.
419               // If so, add an alignment to result.
420               //
421               len = end_s1 - start_s1 + 1;
```

```
422
423             if (len >= params._min_strlen)
424               {
425                 BroSubstring *str = _s1->GetSubstring(start_s1, len);
426                 str->AddAlignment(_s1, start_s1);
427                 str->AddAlignment(_s2, start_s2);
428                 lcs.push_back(str);
429               }
430
431             start_s1 = s1;
432             start_s2 = s2;
433           }
434
435         end_s1 = s1;
436         end_s2 = s2;
437         node = node->_prev;
438       }
439
440     // Finish the last substring:
441     //
442     len = end_s1 - start_s1 + 1;
443
444     if (len >= params._min_strlen)
445       {
446         BroSubstring *str = _s1->GetSubstring(start_s1, len);
447         str->AddAlignment(_s1, start_s1);
448         str->AddAlignment(_s2, start_s2);
449         lcs.push_back(str);
450       }
451   }
452
453   int NumCols() { return _last_col; }
454
455 private:
456
457   bool AdjustWin(Node* guide, NWin* win, bool last_col)
458   {
459     Node* n;
460
461     // Adjust high boundary, potentially moving all pointers downward.
462     //
463     n = win->_hi->_down;
464
465     while (win->_hi->_s1_idx <= guide->_s1_idx)
466       {
467         if (win->_lo == win->_hi)
468           win->_lo = 0;
469         if (win->_ext == win->_hi)
470           win->_ext = n;
471
472         pq.Delete(win->_hi);
473         win->_hi = n;
474
475         if (! n || (!last_col && n && ! n->_prev))
476           return false;
477
478         n = n->_down;
479       }
480
481     if (! win->_lo)
482       {
483         win->_lo = win->_hi;
```

```
484           pq.Push(win->_hi);
485         }
486
487       // Adjust low boundary, potentially updating the maximum.
488       //
489       n = win->_lo->_down;
490
491       while (n && n->_s2_idx > guide->_s2_idx)
492         {
493           if (! last_col && ! n->_prev)
494             break;
495
496           pq.Push(n);
497           win->_lo = n;
498           n = n->_down;
499         }
500
501       win->_max = pq.Top();
502
503       // Adjust extension boundary, if feasible.
504       //
505       // After pushing down the top end of the window, the extension's
506       // s1 index already is smallest possible bit larger than guide's.
507       // Now keep scanning while the s2 index is too big.
508       //
509       n = win->_ext->_down;
510
511       while (n && n->_s2_idx > guide->_s2_idx && n->_s1_idx == win->_ext->_s1_idx)
512         {
513           if (! last_col && ! n->_prev)
514             break;
515
516           win->_ext = n;
517           n = n->_down;
518         }
519
520       return true;
521     }
522
523     // Input strings -- s1 is aligned along rows, s2 along columns.
524     //
525     const BroString* _s1;
526     const BroString* _s2;
527
528     Node *_nodes;
529     Node *_node;       // current node among _nodes
530     NSet *_nset;
531     int _last_col;     // current last column
532
533     int   _best_score;
534     Node *_best_node; // Overall best node
535   };
536
537   void jacobson_vo(const BroString* s1, const BroString* s2,
538                    SWParams& params, SW_LCS& result)
539   {
540     byte_vec s1_bytes = s1->Bytes();
541     byte_vec s1_bytes_end = s1_bytes + s1->Len();
542     byte_vec s2_bytes = s2->Bytes();
543     byte_vec s2_bytes_end = s2_bytes + s2->Len();
544     int i, num_rows = 0, pi_length = 0;
545     byte_vec bv;
```

```
546     static int iteration = 0;
547
548     iteration++;
549
550     // Pi build-up.
551     //
552     for ( bv = s1_bytes; bv < s1_bytes_end; bv++ )
553       jv_indices.AddUsage(*bv, iteration);
554     for ( i = 0, bv = s2_bytes; bv < s2_bytes_end; i++, bv++ )
555       jv_indices.AddIndex(*bv, i, iteration);
556
557     pi_length = jv_indices.GetPiLength();
558     JV_CoverMatrix cover(s1, s2, pi_length);
559
560     for ( i = 0, bv = s1_bytes; bv < s1_bytes_end; i++, bv++ )
561       {
562         const JV_Indices::Ind* ind = jv_indices.GetIndices(*bv);
563         int last_col = cover.NumCols();
564
565         // Greedy cover generation over Pi.
566         for ( JV_Indices::IndCIt it = ind->begin(); it != ind->end(); ++it )
567           {
568             int s1_idx = i;
569             int s2_idx = *it;
570             JV_CoverMatrix::NSet* set = cover.FindSequence(s2_idx, last_col);
571             cover.SetNode(set, s2_idx, s1_idx);
572           }
573       }
574
575     cover.SetBackpointers();
576     cover.CollectLCS(params, result);
577   }
```

Bibliography

Each entry in this bibliography includes in parentheses the page numbers on which it was referenced.

[1] Christopher Alberts and Audrey Dorofee. *Managing Information Security Risks: The OCTAVE (SM) Approach*. Addison-Wesley, July 2002. (Page 7.)

[2] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *J. Mol. Biol*, 215(3):403–410, 1990. (Page 119.)

[3] James P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., Ford Washington, PA, April 1980. (Page 8.)

[4] R.J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc. New York, NY, USA, 2001. ISBN 0471389226. (Pages 18, 87.)

[5] T. Anderson, T. Roscoe, and D. Wetherall. Preventing internet denial-of-service with capabilities, 2003. (Page 116.)

[6] William A. Arbaugh, William L. Fithen, and John McHugh. Windows of vulnerability: A case study analysis. *Computer*, 33(12):52–58, 2000. (Page 12.)

[7] K. Argyraki and D. Cheriton. Network Capabilities: The Good, the Bad and the Ugly. *Proc of 4th ACM Workshop on Hot Topics in Networks*, 2005. (Page 116.)

[8] Stefan Axelsson. Intrusion Detection Systems: A Survey and Taxonomy. Technical Report 99-15, Depart. of Computer Engineering, Chalmers University, March 2000. (Page 13.)

[9] M. Bailey, E. Cooke, F. Jahanian, J. Nazario, and D. Watson. The Internet motion sensor: A distributed blackhole monitoring system.

Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS05), San Diego, CA, Feb, 2005. (Page 112.)

[10] F. Baker and P. Savola. Ingress Filtering for Multihomed Networks. RFC 3704, IETF, March 2004. URL `http://www.ietf.org/rfc/rfc3704.txt`. (Page 111.)

[11] Paul Baran. On Distributed Communication Networks. *IEEE Transactions on Communications*, 12:1–9, Mar 1964. (Page 7.)

[12] S.A. Baset and H. Schulzrinne. An analysis of the Skype peer-to-peer internet telephony protocol. Technical report, Columbia University, New York, NY, 2004. (Pages 59, 86.)

[13] Marshall Beddoe. Protocol informatics. `http://www.baselineresearch.net/PI`, 2005. (Page 55.)

[14] S. Bellovin. The Security Flag in the IPv4 Header. RFC 3514, IETF, April 2003. (Page 112.)

[15] J. Bethencourt, J. Franklin, and M. Vernon. Mapping Internet Sensors With Probe Response Attacks. *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, August*, 2005. (Pages 103, 120.)

[16] Philippe Biondi and Fabrice Desclaux. Silver Needle in the Skype. In *BlackHat Europe*, March 2006. (Page 86.)

[17] DNS Providers Blacklist. `http://www.dnsbl.org/`. (Page 14.)

[18] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. ISSN 0001-0782. (Page 56.)

[19] M. Blumenthal and D. Clark. Rethinking the Design of the Internet: the End-to-End Arguments vs. the Brave New World. *ACM Transactions on Internet Technology*, 1(1), August 2001. (Page 121.)

[20] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy, Oakland, CA, USA*, pages 2–16, May 2006. (Page 115.)

[21] J. Brunner. *Shockwave Rider*. Del Rey, 1984. (Page 9.)

[22] Deutscher Bundestag. *Gesetz über den Datenschutz bei Telediensten*. Bundesministerium der Justiz, July 1997. BGBl. I S. 1870, 1871. (Page 18.)

[23] Christian Burks, James W. Fickett, Walter B. Goad, Minoru Kanehisa, Frances I. Lewitter, Wayne P. Rindone, C. David Swindell, Chang-

Shung Tung, and Howard S. Bilofsky. The genbank nucleic acid sequence database. *Comput. Appl. Biosci.*, 1(4):225–233, 1985. doi: 10.1093/bioinformatics/1.4.225. (Page 27.)

[24] Internet Storm Center. Mcafee 4715 dat false positive deletion reports follow-up. `http://isc.sans.org/diary.php?storyid=1184`, March 2006. (Page 16.)

[25] V.C. Cerf and R.E. Kahn. A Protocol for Packet Network Interconnections. *IEEE Transactions on Communications*, COM-22(5), May 1974. (Page 8.)

[26] CERT. Advisory CA-1991-04 Social Engineering. `http://www.cert.org/advisories/CA-1991-04.html`, April 1991. (Page 10.)

[27] CERT. Advisory CA-1992-03 Internet Intruder Activity. `http://www.cert.org/advisories/CA-1992-03.html`, February 1992. (Page 9.)

[28] CERT. Advisory CA-1992-14 Altered System Binaries Incident. `http://www.cert.org/advisories/CA-1992-14.html`, June 1992. (Page 9.)

[29] CERT. Advisory CA-1994-01 Ongoing Network Monitoring Attacks. `http://www.cert.org/advisories/CA-1994-01.html`, February 1994. (Page 10.)

[30] CERT. Advisory CA-1996-01 UDP Port Denial-of-Service Attack. `http://www.cert.org/advisories/CA-1996-01.html`, February 1996. (Page 10.)

[31] CERT. Advisory CA-1996-26 Denial-of-Service Attack via ping. `http://www.cert.org/advisories/CA-1996-26.html`, December 1996. (Page 10.)

[32] CERT. Advisory CA-1997-28 IP Denial-of-Service Attacks. `http://www.cert.org/advisories/CA-1997-28.html`, December 1997. (Page 10.)

[33] CERT. Advisory CA-1998-01 Smurf IP Denial-of-Service Attacks. `http://www.cert.org/advisories/CA-1997-28.html`, January 1998. (Page 10.)

[34] CERT. Advisory CA-1999-04 Melissa Macro Virus. `http://www.cert.org/advisories/CA-1999-04.html`, March 1999. (Page 10.)

[35] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004. (Page 115.)

[36] William R. Cheswick. An evening with Berferd, in which a cracker is lured, endured, and studied. In *Proceedings of the 1992 Winter USENIX Conference*, 1992. (Page 112.)

[37] Chris Clark, Wenke Lee, David Schimmel, Didier Contis, Mohamed Kone, and Ashley Thomas. A hardware platform for network intrusion detection and prevention. In *Proceedings of The 3rd Workshop on Network Processors and Applications (NP3)*, Madrid, Spain, February 2004. (Page 103.)

[38] Fred Cohen. Computer viruses : Theory and experiments. *Computers and Security*, 6:22–34, 1987. (Page 8.)

[39] U.S. Congress. *Freedom of Information Act*. U.S. Department of State, 2002. 5 U.S.C. §552. (Page 18.)

[40] U.S. Congress. *Health Insurance Portability and Accountability Act*. U.S. Department of Health & Human Services, 1996. (Page 18.)

[41] U.S. Congress. *Children's Online Privacy Protection Act*. U.S. Federal Trade Commission, 1998. (Page 18.)

[42] Evan Cooke, Farnam Jahanian, and Danny McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Proceedings of SRUTI'05: Steps to Reducing Unwanted Traffic on the Internet Workshop*, pages 39–44, June 2005. (Pages 11, 112.)

[43] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 133–147, 2005. (Page 115.)

[44] Jedidiah R. Crandall, Shyhtsun Felix Wu, and Frederic T. Chong. Experiences using minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *DIMVA*, pages 32–50, 2005. (Pages 87, 114.)

[45] S. Crosby and D. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium*. USENIX, August 2003. (Page 50.)

[46] CS3, Inc. MANAnet Reverse Firewall: Fighting DDoS attacks at their origins. `http://www.cs3-inc.com/ps_rfw.html`. (Page 116.)

[47] FM Cuenca-Acuna, C. Peery, RP Martin, and TD Nguyen. PlanetP: using gossiping to build content addressable peer-to-peer information sharing communities. *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*, pages 236–246, 2003. (Page 56.)

[48] W. Cui, V. Paxson, N. Weaver, and R. H. Katz. Protocol-independent adaptive replay of application dialog. In *13th Annual Network and Distributed System Security Symposium (NDSS), San Diego, USA*, February 2006. (Pages 55, 83.)

[49] AP Dempster, NM Laird, and DB Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977. (Page 83.)

[50] Dorothy Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, February 1987. (Page 9.)

[51] Christian Dewes, Arne Wichmann, and Anja Feldmann. An analysis of Internet chat systems. In *Proc. of the Second Internet Measurement Workshop (IMW)*, Nov 2002. (Page 82.)

[52] S. Dharmapurikar and V. Paxson. Robust TCP stream reassembly in the presence of adversaries. In *Procedings of the 14th USENIX Security Symposium*. USENIX, August 2005. (Page 51.)

[53] S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor. Longest prefix matching using Bloom filters. *Proceedings of 2003 ACM SIGCOMM Conference*, pages 201–212, 2003. (Page 56.)

[54] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, and J.W. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1):52–61, 2004. (Page 56.)

[55] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. *Proceedings of the 13th USENIX Security Symposium*, 2, 2004. (Page 55.)

[56] H. Dreger, C. Kreibich, R. Sommer, and V. Paxson. Enhancing the accuracy of network-based intrusion detection with host-based context. In *Proceedings of the conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), Vienna, Austria*, July 2005. (Page 51.)

[57] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic application-layer protocol analysis for network intrusion detection. In *Procedings of the 15th USENIX Security Symposium*. USENIX, August 2006. (Page 120.)

[58] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Operational experiences with high-volume network intrusion detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 2–11, October 2004. (Pages 22, 25, 51.)

[59] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998. (Page 28.)

[60] Richard Durbin, Sean Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis*. Cambridge University Press, April 1998. (Pages 26, 34.)

[61] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *ACM SIGCOMM Computer Communication Review, Proceedings of the 2002 SIGCOMM Conference*, volume 32, pages 323–336, 2002. (Page 56.)

[62] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827, IETF, May 2000. (Page 110.)

[63] National Center for Biotechnology Information. GenBank Statistics. `http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html`, March 2006. (Page 27.)

[64] Fritz Froehlich and Allen Kent. *The Froehlich/Kent Encyclopedia of Telecommunications*, volume 10, pages 231–255. Marcel Dekker, March 1995. (Page 7.)

[65] David Geer. Malicious bots threaten network security. *Computer*, 38 (1):18–20, 2005. (Page 11.)

[66] D. Gerrold. *When Harlie Was One*. Ballantine Books, 1975. (Page 9.)

[67] T.M. Gil and M. Poleto. MULTOPS: a datastructure for bandwidth attack detection. In *Proceedings of the 10th Usenix Security Symposium*, Aug 2001. (Page 115.)

[68] Dan Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997. (Pages 26, 29, 30, 33, 34, 50, 66.)

[69] Patrick Haffner, Subhabrata Sen, Oliber Spatscheck, and Dongmei Wang. ACAS: Automated construction of application signatures. In *Proc. of the ACM SIGCOMM Workshop on Mining Network Data*, August 2005. (Page 82.)

[70] Hong Han, Xian Liang Lu, Jun Lu, Chen Bo, and Ren Li Yong. Data mining aided signature discovery in network-based intrusion detection system. *SIGOPS Oper. Syst. Rev.*, 36(4):7–13, 2002. ISSN 0163-5980. (Page 113.)

[71] Mark Handley, Christian Kreibich, and Vern Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 9th USENIX Security Symposium*, August 2001. (Pages 12, 51, 83, 91, 94.)

[72] L.T. Heberlein, G.V. Dias, K.N. Levitt, B. Mukherjee, J. Wood, and D. Wolber. A network security monitor. In *IEEE Computer Society Symposium on Research in Security in Privacy*, pages 296–304, Oakland, CA, USA, May 1990. (Page 9.)

[73] Paul G. Higgs and Teresa K. Attwood. *Bioinformatics and molecular evolution*. Blackwell Science Ltd, 2005. ISBN 1-4051-0683-2. (Page 28.)

[74] Thorsten Holz and Georg Wicherski. Know your enemy: Tracking botnets. `http://www.honeynet.org/papers/bots/`, March 2005. (Page 112.)

[75] IANA. TCP and UDP port numbers. `http://www.iana.org/assignments/port-numbers`. (Page 59.)

[76] *Intel vPro: Built-in Manageability and Improved Security for Desktop PCs*. Intel Corp., 2006. `ftp://download.intel.com/vpro/pdfs/vpro_wp.pdf`. (Page 103.)

[77] J. Ioannidis and S.M. Bellovin. Implementing pushback: Router-based defense against DDoS attacks. *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS 2002)*, 2002. (Pages 105, 116, 121.)

[78] G. Jacobson and K. P. Vo. Heaviest increasing/common subsequence problems. In *Proc. of the 3rd Symposium on Combinatorial Pattern Matching*, volume 644, pages 52–65. Springer LNCS, 1992. (Page 34.)

[79] Thomas Karagiannis, Andre Broido, Michalis Faloutsos, and KC Claffy. Transport layer identification of p2p traffic. In *Proc. of the Second Internet Measurement Workshop (IMW)*, Nov 2002. (Pages 55, 82.)

[80] Thomas Karagiannis, Andre Broido, Nevil Brownlee, KC Claffy, and Michalis Faloutsos. Is P2P dying or just hiding? In *IEEE Globecom 2004 - Global Internet and Next Generation Networks, Dallas/Texas, USA*, Nov, 2004. IEEE. (Pages 55, 82.)

[81] Thomas Karagiannis, Dina Papagiannaki, and Michalis Faloutsos. BLINC: Multilevel traffic classification in the dark. In *Proceedings of the 2005 ACM SIGCOMM Conference*, oct 2005. (Pages 55, 82.)

[82] J.O. Kephart and W.C. Arnold. Automatic Extraction of Computer Virus Signatures. *4th Virus Bulletin International Conference*, pages 178–184, 1994. (Page 113.)

[83] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th Usenix Security Symposium, San Diego, CA*, 2004. (Pages 56, 113.)

[84] Stefan Kornexl, Vern Paxson, Holger Dreger, Anja Feldmann, and Robin Sommer. Building a time machine for efficient recording and retrieval of high-volume network traffic. In *Proceedings of the 2005 Internet Measurement Conference, Berkeley, CA, USA*, pages 267–272. USENIX, October 2005. (Page 27.)

[85] Christian Kreibich and Robin Sommer. Policy-controlled event management for distributed intrusion detection. In *Proceedings of the 4th International Workshop on Distributed Event-Based Systems (DEBS'05)*, June 2005. (Page 14.)

[86] Christian Kreibich, Andrew Warfield, Jon Crowcroft, Steven Hand, and Ian Pratt. Using packet symmetry to curtail malicious traffic. In *Proceedings of the Fourth Workshop on Hot Topics in Networks (HotNets-IV), College Park/Maryland, USA*, November 2005. (Pages 104, 106, 111.)

[87] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: methods, evaluation, and applications. *Proceedings of the 2003 ACM SIGCOMM Conference on Internet Measurement*, pages 234–247, 2003. (Pages 56, 115.)

[88] J. Kubiatowicz, C. Wells, B. Zhao, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, et al. OceanStore: an architecture for global-scale persistent storage. *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201, 2000. (Page 56.)

[89] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. Space-code bloom filter for efficient per-flow traffic measurement. *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, 3:1762–1773, 2004. (Page 115.)

[90] George Kurtz, Bruce Schneier, Alfred Huger, Martin Roesch,

and Jennifer Granick. The Honeynet Project. `http://project.honeynet.org`, 1999. (Page 112.)

[91] L7 Application-layer Filtering. L7 application-layer filtering. `http://l7-filter.sourceforge.net`. (Page 75.)

[92] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. *Proceedings of the 2005 ACM SIGCOMM Conference*, pages 217–228, 2005. (Page 82.)

[93] W.J. Li, K. Wang, SJ Stolfo, and B. Herzog. Fileprints: identifying file types by n-gram analysis. In *Proceedings of the Systems, Man and Cybernetics (SMC) Information Assurance Workshop, 2005. Proceedings from the Sixth Annual IEEE*, pages 64–71, 2005. (Page 83.)

[94] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and Brian Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy, Oakland, CA, USA*, May 2006. (Pages 30, 101, 114.)

[95] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. *Proceedings of the 12th ACM conference on Computer and communications security*, pages 213–222, 2005. (Page 115.)

[96] J.C.R. Licklider. Man-Computer Symbiosis. *IRE Transactions on Human Factors in Electronics*, HFE-1:4–11, Mar 1960. (Page 8.)

[97] DJ Lipman and WR Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435, 1985. (Page 119.)

[98] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. Voelker. Unexpected means of identifying protocols. In *Proceedings of the 2005 Internet Measurement Conference*, October 2006. (Pages 73, 74.)

[99] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. Voelker. Automatic protocol inference: Unexpected means of identifying protocols. Technical Report CS2006-0850, University of California, San Diego, February 2006. (Page 73.)

[100] S. McCanne, C. Leres, and V. Jacobson. tcpdump/libpcap. `http://www.tcpdump.org/`, 1994. (Page 90.)

[101] Edward. M. McCreight. A space-economical suffix-tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976. (Pages 30, 90.)

[102] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the source. In

Proceedings of 10th IEEE International Conference on Network Protocols, Nov 2002. (Page 116.)

[103] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, volume 21, pages 39–51, 1987. (Page 9.)

[104] Andrew Moore and Dina Papagiannaki. Toward the accurate identification of network applications. In *Proc. of the Passive and Active Measurement Workshop*, March 2005. (Pages 82, 106.)

[105] Andrew Moore, James Hall, Christian Kreibich, Euan Harris, and Ian Pratt. Architecture of a network monitor. In *Passive and Active Measurement Workshop Proceedings*, pages 77–86, La Jolla, California, April 2003. (Page 106.)

[106] D. Moore. Network Telescopes: Observing Small or Distant Security Events. *Invited presentation at the 11th Usenix Security Symp.(SEC 02), Aug*, 2002. (Page 112.)

[107] D. Moore, C. Shannon, G. Voelker, and S. Savage. Network telescopes. Technical report, Technical Report CS2004-0795, CSE Department, UCSD, July 2004. (Page 112.)

[108] D. Moore, C. Shannon, D.J. Brown, G.M. Voelker, and S. Savage. Inferring Internet denial-of-service activity. *ACM Transactions on Computer Systems (TOCS)*, 24(2):115–139, 2006. (Pages 112, 115.)

[109] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *INFOCOM*, 2003. (Pages 14, 102.)

[110] S.J. Murdoch and G. Danezis. Low-Cost Traffic Analysis of Tor. *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 183–195, 2005. (Page 55.)

[111] Computer History Museum. Internet History. `http://www.computerhistory.org/exhibits/internet_history/`, 2004. (Page 7.)

[112] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970. (Page 29.)

[113] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *Proceedings of the 12th Annual Network and Distributed Sys-*

tem Security Symposium (NDSS05), San Diego, CA, Feb, 2005. (Page 115.)

[114] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proc. 2005 IEEE Symposium on Security and Privacy*, pages 226–241, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2339-0. doi: http://dx.doi.org/10.1109/SP.2005.15. (Pages 55, 113.)

[115] Tim Oliver, Bertil Schmidt, and Douglas Maskell. Hyper customized processors for bio-sequence database scanning on FPGAs. In *FPGA '05: Proc. of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 229–237, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-029-9. doi: http://doi.acm.org/10.1145/1046192.1046222. (Page 33.)

[116] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of Internet background radiation. *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, pages 27–40, 2004. (Page 112.)

[117] UK Parliament. *Regulation of Investigatory Powers Act*. The Stationery Office, Ltd., 2000. ISBN 0-10-542300-9. (Page 17.)

[118] UK Parliament. *Data Protection Act*. The Stationery Office, Ltd., 1998. ISBN 0-10-542998-8. (Page 17.)

[119] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23-24): 2435–2463, 1998. (Pages 10, 45.)

[120] T. Peng, C. Leckie, and K. Ramamohanarao. Defending against distributed denial of service attack using selective pushback. *Proceedings of the Ninth IEEE International Conference on Telecommunications (ICT 2002)*, 2002. (Pages 105, 116, 121.)

[121] Ian Peter. History of the Internet. `http://www.nethistory.info`, 2004. (Page 7.)

[122] P. Pevzner and M. Waterman. Matrix longest common subsequence problem, duality and Hilbert bases. In *Proc. of the 3rd Symposium on Combinatorial Pattern Matching*, volume 644, pages 79–89. Springer LNCS, 1992. (Page 34.)

[123] The Spamhaus Project. `http://www.spamhaus.org/`. (Page 14.)

[124] Niels Provos. Honeyd - a virtual honeypot daemon. In *10th DFN-CERT Workshop, Hamburg, Germany*, February 2003. (Page 112.)

[125] T. H. Ptacek and T. N. Newsham. Insertion, evasion and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., 1998. (Pages 12, 14, 51, 91.)

[126] Marcus Ranum. The six dumbest ideas in computer security. `http://www.ranum.com/security/computer_security/editorials/dumb/`, September 2005. (Page 87.)

[127] AMA Research. Electronic monitoring & surveillance survey. Technical report, American Management Association, 2005. (Page 17.)

[128] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th Conference on Systems Administration*, pages 229–238, 1999. (Page 10.)

[129] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov 1984. (Page 121.)

[130] A. Sanfeliu and K. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-13(3):353–362, 1981. (Page 65.)

[131] Stefan Savage, David Wetherall, Anna R. Karlin, and Tom Anderson. Practical network support for IP traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference*, pages 295–306, 2000. (Page 116.)

[132] S.E. Schechter. *Computer Security Strength & Risk: A Quantitative Approach*. PhD thesis, Harvard University, 2004. (Page 11.)

[133] Bruce Schneier. *Secrets and Lies*, pages 318–333. John Wiley and Sons, New York, 2000. (Page 7.)

[134] Bruce Schneier. *Beyond Fear: Thinking Sensibly about Security in an Uncertain World*. Copernicus Books, 2003. (Page 87.)

[135] David V. Schuehler and John Lockwood. TCP-Splitter: A TCP/Ip flow monitor in reconfigurable hardware. In *10th Symposium on High Performance Interconnects (HotI'02)*. IEEE Computer Society, August 2002. (Page 25.)

[136] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reversible sketches for efficient and accurate change detection over network data streams. *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, pages 207–212, 2004. (Page 115.)

[137] Subhabrata Sen, Oliver Spatscheck, and Dongmei Want. Accurate, scalable in-network identification of P2P traffic using application

signatures. In *Proc. of the 13th International World Wide Web Conference*, may 2004. (Pages 59, 82.)

[138] Steven L. Shaffer and Alan R. Simon. *Network Security*. Academic Press Professional, Inc., San Diego, CA, USA, 1994. ISBN 0-12-638010-4. (Page 6.)

[139] Umesh Shankar and Vern Paxson. Active mapping: Resisting NIDS evasion without altering traffic. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 41–59. IEEE, 2003. (Pages 12, 51.)

[140] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation*, Dec 2004. (Pages 56, 113.)

[141] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147, 1981. (Page 29.)

[142] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based IP traceback. In *ACM SIGCOMM Computer Communication Review, Proceedings of the 2001 SIGCOMM Conference*, volume 34, pages 3–14, 2001. (Pages 56, 105, 116.)

[143] A. C. Snoeren, C. Partridge, C. E. Jones L.A. Sanchez, F. Tchakountio, B. Schwartz, S. T. Kent, and W. T. Strayer. Single-packet IP traceback. *IEEE/ACM Transactions on Networking (TON)*, 10:721–734, Dec. 2002. (Pages 105, 116.)

[144] D.X. Song and A. Perrig. Advanced and authenticated marking schemes for IP traceback. *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 2:878–886, 2001. (Page 116.)

[145] Eugene Spafford. The Internet worm: Crisis and aftermath. *Communications of the ACM*, 32(6):678–687, June 1989. (Page 9.)

[146] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, 2003. URL `http://www.tracking-hackers.com/book/`. (Pages 11, 19, 88, 112.)

[147] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, August 2002. (Page 11.)

[148] Bruce Sterling. *The Hacker Crackdown: Law And Disorder On The Electronic Frontier*. Bantam, 1993. Also online at `http://stuff.mit.edu/hacker/hacker.html`. (Page 8.)

[149] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960, IETF, October 2000. (Page 22.)

[150] Clifford Stoll. *The Cuckoo's Egg*. Addison-Wesley, 1986. (Pages 9, 112.)

[151] Symantec. Support: Cannot connect to AOL after running LiveUpdate on march 15, 2006. `http://service1.symantec.com/SUPPORT/sharedtech.nsf/docid/2006031520164313`, March 2006. (Page 16.)

[152] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley/Symantec Press, 2005. (Page 9.)

[153] Yong Tang and Shigang Chen. Defending against Internet worms: A signature-based approach. In *Proc. of IEEE INFOCOM 05, Miami, Florida, USA*. IEEE, May 2005. (Pages 83, 114.)

[154] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, (14):249–260, 1995. (Pages 30, 90.)

[155] Carnegie Mellon University. DARPA Establishes Computer Emergency Response Team. Press release. `http://www.cert.org/about/1988press-rel.html`, Dec 1988. (Page 9.)

[156] M. Vojnovic and A. Ganesh. On the effectiveness of automatic patching. *Proceedings of the Third Workshop on Rapid Malcode ACM-SIGSAC (WORM)*, 2005. (Page 87.)

[157] M. Vojnović and A. Ganesh. On the race of worms, alerts and patches. Technical report, Technical Report TR-2005-13, Microsoft Research, February 2005. (Page 87.)

[158] G. Voss, A. Schröder, W. Müller-Wittig, and B. Schmidt. Using graphics hardware to accelerate biological sequence analysis. In *Proc. of IEEE Tencon*, Melbourne, Australia, 2005. (Page 33.)

[159] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A.C. Snoeren, G.M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. *Operating Systems Review*, 39: 148–162, 2005. (Page 113.)

[160] M. Walfish, H. Balakrishnan, D. Karger, and S. Shenker. DoS: Fighting fire with fire. *Proc of 4th ACM Workshop on Hot Topics in Networks*, 2005. (Page 116.)

[161] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 ACM SIGCOMM Conference*, volume 34, pages 193–204. ACM Press, October 2004. (Pages 102, 121.)

[162] Y.M. Wang, D. Beck, J. Wang, C. Verbowski, and B. Daniels. Strider Typo-Patrol: Discovery and analysis of systematic typo-squatting. In *Proceedings of the 2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, July 2006. (Page 120.)

[163] David Watson, Matthew Smart, G. Robert Malan, and Farnam Jahanian. Protocol scrubbing: Network security through transparent flow modification. *IEEE/ACM Transactions on Networking*, 12(2):261–73, April 2004. (Pages 51, 83.)

[164] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A taxonomy of computer worms. *Proceedings of the 2003 ACM workshop on Rapid Malcode*, pages 11–18, 2003. (Page 11.)

[165] Nicholas Weaver and Vern Paxson. A worst-case worm. In *Proceedings of the third Annual Workshop on Economics and Information Security (WEIS04)*, May 2004. (Pages 11, 102.)

[166] Nicholas Weaver, Stuart Staniford, and Vern Paxson. Very fast containment of scanning worms. In *Proceedings of the 13th Usenix Security Symposium, San Diego, CA*, pages 29–44, 2004. (Page 14.)

[167] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973. (Pages 30, 90.)

[168] A. Yaar, A. Perrig, and D. Song. Pi: A path identification mechanism to defend against ddos attacks. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 1–15, 2003. (Pages 105, 116.)

[169] V. Yegneswaran, P. Barford, and D. Plonka. On the design and use of Internet sinks for network abuse monitoring. *Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID), Sept*, 2004. (Page 113.)

[170] V. Yegneswaran, J.T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *Proceedings of the 14th Usenix Security Symposium*, 2005. (Page 114.)

[171] Sebastian Zander, Thuy Nguyen, and Grenville Armitage. Self-learning IP traffic classification based on statistical flow characteris-

tics. In *Proc. of the 6th Passive and Active Network Measurement Workshop*, March 2005. (Page 83.)

[172] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. On the characteristics and origins of Internet flow rates. In *Proceedings of the 2002 ACM SIGCOMM Conference*, pages 309–322, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-570-X. (Page 27.)

[173] Hubert Zimmermann. OSI reference model: The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980. (Page 21.)

[174] Denis Zuev and Andrew Moore. Traffic classification using a statistical approach. In *Proc. of the Passive and Active Measurement Workshop*, March 2005. (Page 82.)