THE UNIVERSITY OF CALGARY

Enhancing Firewalls: Conveying User and Application Identification to Network Firewalls

by

Reinderd Gordon Nathan deGraaf

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

MAY, 2007

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled “Enhancing Firewalls: Conveying User and Application Identification to Network Firewalls” submitted by Reinderd Gordon Nathan deGraaf in partial fulfillment of the requirements for the degree of Master of Science.

Dr. Michael John Jacobson, Jr., Supervisor, Department of Computer Science.	Dr. Zongpeng Li, Internal examiner Department of Computer Science.
Dr. John Aycock, Co-Supervisor, Department of Computer Science.	Dr. Behrouz Far, External examiner Department of Electrical & Computer Engineering.

Date

Abstract

Firewalls are used to protect networks from malicious traffic from the outside and limit the flow of information from inside protected networks to the outside world. Most firewalls filter traffic based on network addresses and packet contents. Unfortunately, one major goal of firewalling, that of limiting the *users* and *programs* that can communicate, is not well served by such designs: it is difficult to accurately map network addresses and packet contents to user and program names.

Firewalls can solve the problem of securely mapping user names to addresses when filtering inbound traffic from untrusted networks through the use of *covert authentication systems* such as *port knocking* and *single packet authorization*. Egress firewalls can identify users and programs on trusted networks through the use of application-filters. In this thesis, I survey the current state of both types of systems, describe their weaknesses, and introduce techniques to alleviate some of these weaknesses.

Acknowledgements

First, I must thank my supervisors, John Aycock and Michael Jacobson, for guiding me through my degree and putting up with my frequent topic changes. Without their patience and support, I never would have finished this.

Also, my experimental analysis of UDP packet dropping and out-of-order delivery would not have been possible without the assistance of André Baron, Bryan Kadzban and Patrick Lucas, who volunteered home computers as targets for my packet generator.

Finally, I must thank my mother, for donating a few days of her time to help me edit this thesis, and for having beaten me over the head with a grammar stick throughout grade school until I could write decently.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Security hole
Open, you are insecure
Closed, you are no threat
– Shidoshi, `http://www.ranum.com`

In ancient times, towns and villages were based around market-places, where goods from many sources could be traded freely. Over time, as towns grew into cities and gathered wealth, barbarians grew envious of the city-dwellers. In response to this threat, cities erected defensive walls to protect against outsiders. However, as the cities were still dependent on trade, the walls needed to have many gates to allow passage in and out of the cities; guards monitored who entered and exited and attempted to keep the barbarians out.

So it is with the Internet. When first created, it was designed to foster sharing and collaboration. True to this goal, it was built to be as open as possible, with few to no restrictions. Later, as threats grew, network administrators deployed firewalls, which restrict the network traffic allowed to enter and leave local networks, while still allowing "legitimate" traffic to pass.

Unfortunately, discriminating between "legitimate" and "illegitimate" traffic is not easy. The best practice is to allow only traffic that is explicitly recognized as legitimate while blocking everything else, but this is easier said than done. Factors to take into account when examining traffic include sources, destinations, the users and programs that sent or will receive the traffic, the information being exchanged, the format of the information being exchanged, the time of day, the volume of traffic that has been sent by the source, and others; while not all of these are necessarily appropriate under all circumstances, others that are important are frequently ignored due to lack of information or the difficulty of checking. Also, no defensive measure is perfect: walls can be scaled with ladders or battered down by trebuchets, and security software can be disabled or bypassed by exploiting software or configuration vulnerabilities. For this reason, security (of both cities and computers) depends on the principle of *defense in depth*: the principle that security comes in layers, where the defeat of one layer doesn't leave everything vulnerable and that attackers must bypass multiple layers to reach anything important.

1.1 Problems with Existing Firewall Technology

Most modern firewalls are designed to filter packets or validate protocol semantics, and they are very good at this. However, behind the packets and protocol messages visible to firewalls are users and programs; firewalls have little knowledge of these and consequently aren't very good at filtering based on the users and programs responsible for network traffic. This can be viewed as two separate problems: that of filtering by user on ingress, and that of filtering by user and program on egress.

1. Firewalls can easily limit what services can be reached from outside. However, it may also be necessary to limit which users can connect to those services. A common assumption, made by many modern firewalls, is that trusted users only connect from small sets of trusted hosts with specific addresses; they implement user filtering by blocking incoming packets with source addresses not in these sets. Unfortunately, the source addresses on incoming packets tell little about the user who sent them; malicious users can spoof trusted hosts, and trusted users can connect from untrusted hosts. Since many trusted hosts may have dynamic (DHCP-assigned) IP addresses, opening a firewall to one trusted host

may require opening it to thousands of IP addresses, making it easier for an attacker to find an address to spoof or a machine with a trusted address to hijack. Adjusting the set of trusted IP addresses typically involves either manual reconfiguration by a firewall administrator or connecting to some world-accessible authentication service, which itself may be vulnerable to attack.

2. Although users usually can be accurately linked to IP addresses within a local network, it can be difficult to limit the services with which those users are allowed to communicate. Firewalls generally attempt to filter outbound traffic by restricting the ports to which users may connect: for example, disallowing outbound connections to anything except TCP ports 80 (HTTP), 443 (HTTPS), and 20 and 21 (FTP). Unfortunately, this isn't particularly effective: non-standard services may be running on these allowed ports. Whereas application-layer firewalls can easily filter traffic that doesn't match the expected protocol for a port, it is much more difficult to detect disallowed applications that tunnel traffic through standard protocols on standard ports. For instance, tunnelling various protocols through port 80, normally used for unencrypted WWW traffic, has become quite common [Alb04, BP04], and encryption renders most application-layer filters useless. Also, standard protocols can run on standard ports and still be used for unauthorized purposes. Restricting network access to only authorized local users and programs has the potential to alleviate these problems, but information about the users and applications that generated or will receive network traffic is usually only available at the source or destination hosts themselves, and isn't necessarily reliable.

1.2 Contributions of this thesis

This thesis introduces and describes methods for addressing both of these problems. The first can be addressed by using *covert authentication systems*, systems that allow users to authenticate without making their presence easy for attackers to detect, to allow legitimate users to inform ingress firewalls of their current network addresses and request that subsequent connections be accepted. Two such systems used today are *port knocking* and *single packet authorization* (*SPA*); I survey existing designs for both and highlight their strengths and weaknesses. Of particular concern are their weaknesses: both are frequently implemented with insecure authentication systems, do not authenticate servers to clients, fail in the presence of network address translation, are susceptible to denial-of-service attacks, and do not logically associate authentication exchanges with the network connections that they enable. Port knocking in particular is highly vulnerable to packet loss and reordering. With these flaws in mind, I then propose techniques that can be used to improve on existing port knocking and SPA systems. Challenge-response authentication provides both cryptographically secure authentication and a method to authentication servers to clients; I propose port knocking and SPA designs using challenge-response authentication and show that the overhead imposed by such a system is not unreasonable under most circumstances. I present experimental analysis quantifying the degree of packet loss and re-ordering in packet streams typical of port knocking, and describe and compare several techniques for ensuring that messages transmitted by port knocking can be properly reassembled on delivery, regardless of the degree of reordering. I also propose several novel designs for port knocking systems and discuss their strengths and weaknesses compared to existing systems. Finally, I present and discuss several methods for creating logical associations between authentication exchanges and subsequent connections. This material expands and improves on my previously published work [dAJ05].

The second problem can be addressed by extending the capabilities of application-filtering firewalls. Existing systems generally only provide application filtering on host firewalls; I discuss the advantages of extending network firewalls to also provide application filtering and mechanisms for communicating user and program information to network firewalls. Many designs for application filtering do not detect malicious programs masquerading as legitimate ones under certain circumstances; I suggest ways of preventing this through the use of integrity shells, but point out that solving this problem requires that the operating systems of hosts using application filtering must be trusted by the filtering system. Also, application filtering is ineffective against interpreted programs and those running inside virtual machines, unless the interpreters and virtual environments are also trusted. Finally, existing network firewalls do not always reliably detect when one process uses another as a client, potentially allowing malicious or untrusted programs to make unauthorized network connections. To solve this problem, I present an algorithm for tracking inter-process communication that can identify most such attempts.

Chapter 2 provides background information on networks and security. In Chapter 3, I present my survey of port knocking and single packet authorization, while Chapter 4 presents improvements to port knocking and SPA. Chapter 5 contains my proposed enhancements to application filters. Finally, Chapter 6 gives my conclusions and suggestions for further work.

Chapter 2

Background

The wire protocol guys don't worry about security because that's really a network protocol problem. The network protocol guys don't worry about it because, really, it's an application problem. The application guys don't worry about it because, after all, they can just use the IP address and trust the network.
– Marcus J. Ranum

As background for the ideas presented in the following chapters, this chapter presents a general overview of networking, cryptography, and relevant offensive and defensive computer security technologies.

2.1 Introduction to Networking

The Internet was designed in the 1960s, '70s and '80s as a robust communication system between diverse local networks. Its design is based on a stack of five protocol layers:

1. **Physical** – responsible for encoding and decoding signals over a transmission medium, such as a wire, optical fiber, radio frequency, or avian carrier;

2. **Data link** – responsible for communication between hosts on a physical network segment;

3. **Network** – responsible for global addressing and routing packets between physical network segments;

4. **Transport** – responsible for communication between processes, and optionally, reliable connections;

5. **Application** – responsible for encoding and decoding information in formats understood by applications, as well as providing any necessary services not provided by lower layers.

Each protocol layer provides services to layers above it; information being sent is passed down from the application layer to the physical layer, with each level performing transformations appropriate to that layer, before the physical layer handles the actual work of transmission. When information is received, each protocol layer undoes its transformations and passes information back up the stack, until it reaches the application layer. This design is similar to the OSI network stack model [Tan96], which mandates seven layers and a somewhat different breakdown of responsibilities.

The infrastructure of the Internet consists of a number of *routers*, interconnected computers whose function is to forward information from its source to its intended destination; the process of finding such a path and forwarding information along it is known as *routing*. Computers and other devices that communicate over the Internet are known as *hosts*.

2.1.1 Network and Transport Protocols

The standard network protocol on the Internet is known as the *Internet Protocol, version 4* (*IPv4*) [Pos81c]. (In this thesis, the abbreviation “IP”, for “Internet Protocol”, always refers to IPv4; the next-generation Internet protocol, *IPv6* will not be discussed.) IP provides an unreliable datagram service: it breaks the information that it transmits into *packets* (also known as *datagrams*) and routes each to its destination independently, but provides no guarantees that packets are properly delivered. Packets may be dropped, duplicated, delayed, re-ordered, or corrupted in transit; error messages (in the form of ICMP packets [Pos81b]) may or may not be returned if delivery fails. IP packets contain a header of 20 to 60 bytes and a payload of 8 to 65,516 bytes of data; packet sizes are selected based on the properties of lower protocol layers. IP headers contain metadata that enables packets to be routed to their destinations, such as the addresses of packets' sources and destinations. IP addresses are 32-bit numbers and are not necessarily unique; routers necessarily have more than one address, and some techniques allow hosts to share IP addresses.

The Internet doesn't *strictly* follow the OSI stack model: various protocols exist that allow IP packets to be encapsulated inside protocols that run on top of IP. This encapsulation of protocols is known as *tunnelling*. For instance, IPsec (a security architecture for IP) [KS05] can create encrypted tunnels for IP and other network protocols on top of IP, whereas GRE ("Generic Routing Encapsulation") [FLH+00] can create plain-text tunnels. More information on tunnelling is available in [CBR03].

A number of transport layers exist, each providing different services. ICMP ("Internet Control Message Protocol") is used for delivering many types of error messages from the network and transport layers as well as performing a variety of administrative functions [Pos81b]. UDP ("User Datagram Protocol") provides an unreliable datagram service between applications [Pos80]; the only important feature that it adds to IP is application addressing. TCP ("Transmission Control Protocol"), the most common transport protocol on the Internet, provides reliable bidirectional streams between applications [Pos81a]. Other transport protocols exist, providing other services.

TCP breaks streams down into *segments* which are then encapsulated into IP packets. Like IP packets, TCP segments contain both headers and payload data. TCP assigns sequence numbers to every byte of payload data sent and requires that every byte be acknowledged; if any segments are lost or corrupted, either they will be resent or an error will be detected. Duplicate and out-of-order packets can also be detected and corrected using the sequence numbers. TCP headers contain application-layer addresses, data and acknowledgement sequence numbers, and control flags, among other fields. Segments used to initiate connections have the SYN ("synchronize") flag set; those used to finalize connections have the FIN flag set. The ACK flag indicates that a segment is acknowledging data received from the other end of the connection. Opening a TCP connection requires that both endpoints exchange *initial sequence numbers (ISNs)* using an algorithm known as the *three-way handshake*, shown in Figure 2.1. The client starts the handshake by choosing an ISN and sending it to the server in a segment with the SYN flag set. If the server chooses to accept the connection, it responds with its own ISN in a segment carrying the SYN flag; it also signals that it received the client's ISN by setting its acknowledgement number to the client's ISN plus one and setting the ACK flag. The client then acknowledges receipt of the server's ISN with an ACK segment carrying the server's ISN plus one in its acknowledgement number field. After completing this exchange, a TCP connection is established and both the client and server can send and receive data.

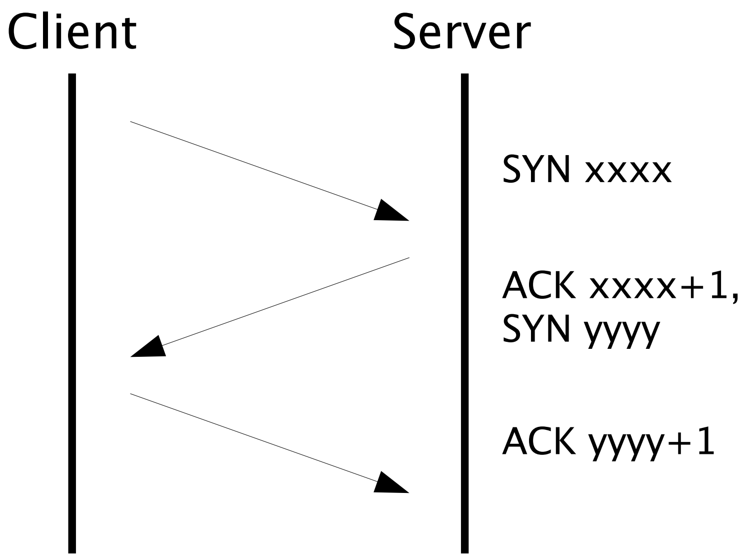


Figure 2.1: TCP connection establishment handshake

Both TCP and UDP use *ports* for addressing applications. A port is simply a 16-bit integer. Any application can register itself (*bind*) to any unused port to send or receive data, although some common services customarily receive connections (*listen*) on particular ports. For instance, SSH ("Secure Shell", used for secure remote logins, file transfer, and tunnelling other protocols) servers typically listen on port 22/TCP, HTTP (used for retrieving documents from web servers) uses port 80/TCP, and DNS ("Domain Name System", responsible for mapping easy-to-remember names to numeric IP addresses) servers listen on port 53/UDP. Unix-based operating systems typically allow only privileged applications (those with superuser or administrative privileges) to use TCP and UDP ports with numbers below 1024.

On most operating systems, applications use the *socket* interface [LFJ$^+$86] to send and receive messages. Unprivileged applications are typically limited to using TCP and UDP; headers, with the exception of addressing information, are automatically generated by the operating system, as are TCP connection establishment and finalization messages. Privileged applications can use the *raw socket* interface to generate arbitrary packets [LFJ$^+$86].

Neither IPv4, TCP, nor UDP provide any authentication or privacy services; any host can generate any packet and any host observing packets at the network layer can read the packets' payloads. If these services are important, then they can be provided by application-layer protocols, TCP extensions such as TLS [DR06], IP extensions such as IPsec [KS05], or the next-generation Internet Protocol, IPv6 [DH98].

2.1.2 Network Application Architectures

Applications that communicate over networks usually follow the *client-server* model: client programs, which may interface with users, connect to servers and request resources. For instance, web browsers connect to servers and request documents, and

SSH clients connect to servers and request login shells. *Peer-to-peer (p2p)* applications operate as both clients and servers, allowing them to connect to each other. *Proxy servers* act as intermediaries between clients and servers; clients connect to proxies, which then connect to servers on their behalf and forward data between the two connections. Such systems are often used for firewalling (as in circuit and application gateways; see Section 2.4.1), caching, and load balancing.

2.1.3 Vulnerabilities in Internet Protocols

When the core Internet protocols were designed, the Internet was a research network with a small user community who knew and trusted each other. As a result, security was not a high priority. Many protocols have useful features that easily lend themselves to malicious use; many even have outright design flaws that weaken security to no tangible benefit. For example,

- Hosts can assert any source IP address they want on packets that they generate (*IP spoofing*). Routers often have no way of determining if packets' source addresses are correct and are not required to block invalid packets even when they can be detected.

- Especially in early TCP implementations, the initial sequence numbers used for new connections were predictable; in conjunction with IP spoofing, this allowed attackers to establish TCP sessions with remote hosts while using spoofed IP addresses [Mor85].

- Many of the protocols used to distribute routing information on the Internet and map physical addresses to IP addresses on local networks are not secure, allowing attackers to intercept traffic destined to other hosts [Bel89, dVdVI98].

- IP packets can be fragmented at (almost) arbitrary boundaries, allowing attackers to split attack signatures between multiple packets. Furthermore, IP fragments and TCP segments are allowed to overlap; it is up to the receiver to decide how to re-assemble them. As a result, firewalls and IDSs that re-assemble streams before checking them may miss attacks against hosts that use different re-assembly strategies [PN98].

Only in the late 1980s did the effects of the insecurity of the Internet start to become well known and, even then, few people worried about it; even well-publicized, high-impact events such as the Morris Worm outbreak [Spa89] did little to sway users. Since then, as the Internet has experienced rapid growth, corrections to many of its fundamental flaws have been few and far between, mostly consisting of patches designed to reduce the risk of an attack with minimal disruption to users.

2.2 Introduction to Cryptography

Modern cryptography seeks to provide the services of authentication, data integrity, confidentiality and non-repudiation [MvOV96, p. 4] through the use of well-known algorithms whose security relies solely on the knowledge of relatively small cryptographic keys. Generally, the strengths of related cryptographic algorithms within a class of algorithms are proportional to the sizes of their keys; an algorithm is considered to be strong if it is computationally infeasible to determine anything about the key, given full knowledge of the plaintext, ciphertext, and algorithm; this is a principle known as *Kerckhoffs' law* [Ker83]. For instance, identifying properly-chosen keys for most modern ciphers is estimated to require thousands of years, even if all the computational power in the world was applied to the problem [Sch06b]. This section provides a brief overview of some of the building blocks of modern cryptography; more information can be found in [MvOV96, Sch06b, Sta03].

2.2.1 Security Services

Cryptographic algorithms are generally used to provide the following five services [IT91, Sta03]:

Authentication The assurance that the identity an entity is as claimed.

Access control The prevention of unauthorized use of a resource.

Data confidentiality The protection of data against unauthorized disclosure.

Data integrity The assurance that data is received or retrieved exactly as it was sent or stored.

Non-repudiation Protection against an entity performing an action and later denying that it did so.

2.2.2 Symmetric Ciphers

Symmetric ciphers use keys that are shared between communicating parties to encrypt and decrypt information. Either the encryption and decryption keys are the same, or they are easily calculated from each other. Symmetric ciphers are typically used for ensuring data confidentiality; they attempt to ensure that ciphertexts cannot be decrypted without knowledge of the key used [Sta03, p. 25]. They tend to be relatively fast and can be implemented in hardware.

Block ciphers are symmetric ciphers that process data in fixed-size blocks, typically 64 or 128 bits in length [MvOV96, ch. 7]. Block ciphers can be used in a variety of modes, such as *electronic codebook* (*ECB*), *counter* (*CTR*) and *cipher block chaining* (*CBC*), differing primarily in how they handle messages longer than a single block.

Most modern block ciphers are based on combinations of permutations and substitutions. The most common modern block cipher is AES, also known as Rijndael [Sta03, ch. 5].

Stream ciphers process data one bit at a time, usually by XORing the data with a pseudorandom key stream [Sta03, p. 192]. Key streams can be produced using permutation generators, linear feedback shift registers, block ciphers, or even hash functions [MvOV96, ch. 6]. A common example of a stream cipher is RC4 [Sta03, p. 194].

Neither form of symmetric cipher typically ensures data integrity or message authentication. Any possible string (in the case of block ciphers, of a multiple of the block size) is typically a valid ciphertext and will decrypt to something.

2.2.3 Cryptographic Hash Functions

Cryptographic hash functions are functions that map arbitrary-length inputs to fixed-length outputs (usually 128 to 512 bits), with the following properties [MvOV96, p. 323]:

1. **Pre-image resistance**: Given a hash h, it is computationally infeasible to find any message m such that $H(m) = h$.

2. **2nd pre-image resistance**: Given a message m, it is computationally infeasible to find another message n such that $H(m) = H(n)$.

3. **Collision resistance**: It is computationally infeasible to find any two messages m, n such that $H(m) = H(n)$.

Although they do not use keys, many hash functions are similar in structure to block ciphers. Hash functions are used for verifying data integrity; a hash on a given message is verified by re-computing it and ensuring that the two hashes match. Common hash functions include SHA-1 and MD5 [Sta03, ch. 12].

A common misconception is that hash functions are a form of encryption, or that hash functions alone provide authentication. Since hash functions are necessarily many-to-one, there is no way to uniquely "decrypt" a message given its hash; the pre-image resistance property of hash functions makes it computationally infeasible to even try. Since there is no secret key involved, anyone can generate a valid hash for any given message. Hashes can only be used for message authentication if they are protected from modification in transit.

2.2.4 Message Authentication Codes

Message authentication codes (MACs) are essentially keyed hash functions and provide both data integrity and authentication services [Sta03, p. 324]: only someone in possession of both a message and the secret key can generate a valid MAC. A MAC algorithm can be constructed from a block cipher in CBC mode by encrypting messages and discarding all but the last ciphertext block [MvOV96, p. 353], or from a hash function using the HMAC algorithm [Sta03, p. 372]. Like hashes, MACs are verified by re-computing and comparing.

2.2.5 Asymmetric Ciphers

Asymmetric ciphers (also known as *public key ciphers*) [MvOV96, ch. 8] use separate keys for encryption and decryption. Moreover, it is computationally infeasible to determine a decryption key given an encryption key. This allows encryption (*public*) keys to be published freely; only decryption (*private*) keys must be kept secret. Anyone can encrypt a message with another party's public key; only that party can decrypt it.

Asymmetric ciphers are usually based on mathematical problems that are believed to be intractable, such as factoring the products of large primes or computing discrete logarithms in various groups. They tend to be slow, so they are frequently used only to encrypt session keys for symmetric ciphers [Sch06b, p. 33]; the messages themselves are encrypted using symmetric ciphers and secret session keys. Two common asymmetric ciphers are RSA [MvOV96, p. 285] and ECIES. [Cer00].

2.2.6 Digital Signatures

While MACs provide integrity and authentication, they depend on symmetric keys and don't provide non-repudiation. In a *digital signature* scheme, messages signatures are generated using senders' private keys; any party can use a sender's public key to verify the signature on a message. Unlike asymmetric ciphers, digital signatures do not provide confidentiality, but they do provide message integrity, authentication and non-repudiation. In order to improve execution performance (the algorithms used in asymmetric cryptography tend to be slow) and prevent existential forgery attacks (in which an attacker can generate a valid signature for a message without having control over the message's contents) [MvOV96, p. 432], digital signatures are typically computed over cryptographic hashes of the original messages. Verification of a signature then requires applying the public key to the signature to recover the sender's hash and recomputing the hash on the message; if the two hashes match, then the message must have been sent by the owner of the public key and has not been altered.

Digital signature algorithms are often constructed from asymmetric ciphers. For instance, the RSA cipher can be used to generate digital signatures by "encrypting" message hashes with the sender's private key; the hash can be recovered by "decrypting" the signature with the sender's public key [MvOV96, p. 433]. Other algorithms designed specifically for digital signatures include DSA [MvOV96, p. 451] and ECDSA

[Cer00].

2.2.7 Key Exchange Algorithms

Both symmetric ciphers and MACs require that all participating parties know a secret key, whereas asymmetric ciphers and digital signatures require that all parties know each others' public keys. Securely distributing these keys is a difficult problem, requiring that all parties are able to guarantee that they received the correct keys and that they were not modified in transit.

The easiest way to solve the problem (for symmetric keys) is for one party to generate the key and distribute it to all other parties in person [Sta03, p. 212], but this is not always practical. If all parties already share a key, then the existing key can be used to create a secure channel to transmit the new one [Sta03, p. 211]; however, if the existing key has been compromised, then so will the new one. When a pre-shared key is available a common strategy is to use it as a *master key* and use it only for distribution of *session keys* [Sta03, p. 213]; the loss of a session key does not compromise the master, and the infrequent use of the master reduces the risk of it being compromised. Other strategies for key distribution rely on a trusted third party, with whom all parties already share keys. The third party generates the new key, establishes secure channels with all parties, and uses the secure channels to distribute the new shared key [Sta03, p. 211]. Similar techniques can be used for distributing public keys.

Instead of one party choosing a symmetric key and passing it to the other, the *Diffie-Hellman key exchange* algorithm [MvOV96, p. 515] (also known as *exponential key exchange*) can be used to allow both parties to contribute to the shared key. Diffie-Hellman is based on the believed intractability of computing discrete logarithms in multiplicative groups of integers modulo primes (with an equivalent existing for elliptic curves); it is computationally infeasible to calculate the shared secret given the messages exchanged between the two parties. However, to prevent man-in-the-middle attacks (see the next section), the messages exchanged must be authenticated, requiring that the two parties share a master key or have authenticated public keys for each other.

2.2.8 Attacks Against Cryptographic Algorithms

All cryptographic algorithms, regardless of any proofs of security that they may have, are insecure if used improperly. The following are some classes of attacks against improperly-used cryptographic algorithms:

Brute-force attacks

When information about the plaintext for a given ciphertext is known, any cipher can be attacked by simply testing all possible keys until the ciphertext decrypts to the desired plaintext. For a cipher employing n-bit keys, such an attack will require that, on average, 2^{n-1} possible keys be tested (i.e., the attack is exponential in the key size). Similarly, pre-images or collisions against n-bit hash functions can be found by testing an average of 2^{n-1} inputs [Sta03, p. 335]. Most cryptographic algorithms are generally only considered secure if the fastest known attack is not significantly faster than brute force [Sch06b, p. 152]. Faster attacks are known against certain classes of algorithms, including many asymmetric ciphers; these must use larger key sizes to compensate [Sch06b, ch. 7]. For example, RSA requires 3072-bit keys to achieve the same level of security as 128-bit AES [BBB^{+}06].

Birthday attacks

A hash function with an output of n bits is generally considered secure if there is no way to find a pre-image or 2nd pre-image that is significantly faster than brute force. However, due to the *birthday paradox* [Sta03, p. 341], collisions can be found in $O(2^{n/2})$ steps, so hash functions expected to have k bits of security against collisions must have outputs of at least $2k$ bits. Significant progress has been made recently in finding collisions in many common hash functions (such as MD5 [Kli06, WFLY04] and SHA-1 [Sch05, WYY05]) in *fewer* than $2^{n/2}$ operations, but, to date, none of these attacks apply to finding pre-images.

Known-plaintext attacks

Known-plaintext attacks are attacks against ciphers in cases where both the plaintext and ciphertext are known [Sta03, p. 28]. Modern ciphers are designed to resist known-plaintext attacks, but knowing the plaintext, or at least part of it, makes it easier to conduct brute-force searches against poorly-chosen keys. Many encrypted messages will have predictable contents at predictable locations; for instance, encrypted email messages may end with predictable signatures, and encrypted IP packets contain predictable header fields. Also, some types of ciphers allow predictable changes to be made to plaintexts by making certain changes to ciphertexts; this is known as *malleability* [DDN91]. For example, in stream ciphers which combine key streams with plaintexts using XOR operations (*additive* stream ciphers), particular bits can be changed in plaintexts by flipping the corresponding ciphertext bits; using this technique, arbitrary changes can be made to portions of plaintexts for which the original plaintext is known.

Replay attacks

Many cryptographic protocols, particularly those used for authentication, depend on the *timeliness* of messages: the property that a message received was generated for the purpose of being used at this time. If no mechanism to ensure timeliness is used, messages may be accepted that were captured by attackers from earlier exchanges and replayed [MvOV96, p. 417], possibly giving some benefit to the attacker.

Reflection attacks

In certain authentication protocols involving symmetric keys, it is possible for an attacker to fraudulently convince a target that it is a trusted party by conducting two simultaneous authentication exchanges, one as a client and the other as a server. By forwarding messages received from the target in one exchange back to the server in the other (i.e., reflecting them), it may be possible to trick the target into generating valid authentication tokens for the attacker [MvOV96, p. 417].

Interleaving attacks

By simultaneously attempting to authenticate to two targets and impersonating the other to each, some poorly-designed authentication protocols can allow an attacker to convince one target that it is the other by relaying messages between them [MvOV96, p. 417].

Man-in-the-middle (MiTM) attacks

In protocols employing unauthenticated messages, an attacker may be able to trick two parties into communicating with it instead of with each other; it forwards appropriate messages between the victims to prevent them from discovering the intruder [Sch06b, p. 48]. A classic MiTM attack against the Diffie-Hellman key exchange protocol involves an attacker carrying out Diffie-Hellman exchanges with both targets, who believe that they are communicating with each other. The result is that the attacker shares session keys with both targets and is able to decrypt and read all messages sent by either before re-encrypting and forwarding them to their intended destination [Sta03, p. 505]. Attacks of this type are sometimes also known as *middle-person* or *intruder-in-the-middle* attacks.

2.3 Attacks and Offensive Technologies

The barbarians at the gates of ancient cities came from many tribes and carried many different weapons. Likewise, attackers against computer systems take many forms and use many different strategies. This section describes some common attack strategies, including port scans, 0-day exploits, worms, and denial-of-service attacks.

2.3.1 Port Scans

Before an attacker, be it person or program, can launch an attack against something, it needs to gather information about its target. Possibly the single most important information about a target computer, from an attacker's perspective, is what services are running. The easiest way to gather this information is by attempting to connect to all ports suspected of running services of interest, in what is known as a *port scan* [Fyo97].

Attackers with a particular target in mind will often scan many or even all ports on the target, in order to identify all running services and gather as much information as possible. Scanning multiple ports on a single target in this manner is known as *vertical scanning* [SHM02]; the unqualified term "port scan" generally refers to vertical scans. Other attackers looking for a particular service, not caring about what host is running it, will scan a single port over a large number of hosts; this is known as *horizontal scanning* [SHM02] or *network scanning* [MMB05].

Port scans can take many forms [dVCIdV99, Fyo97]. The simplest form of scan against TCP ports is the *connect scan*, in which the attacker attempts to open a TCP connection to each port. If the connection attempt succeeds, then the attacker knows that something is listening there and can either attempt to determine what it is or simply make a guess based on the port number; if the connection attempt fails, then the attacker typically assumes that no service is using that port. Connect scans are not particularly stealthy, since a successful connection establishment will be noticed by the application listening on the target, which may log the event. However, there is no need to fully open a TCP connection to determine if something is listening; a TCP stack is required to send a SYN-ACK packet in response to a SYN to any open port and will either send an ICMP error or nothing at all in response to a SYN packet sent to a closed port. This takes place at the kernel level on the target system; the application is never informed of connections that are never fully opened. *SYN scans* take advantage of this by scanning with SYN packets only; the scanner may send RST packets to tear down the half-open connections that it creates. Other types of TCP scans include *ACK scans*, *FIN scans*, *Xmas scans* (using packets with many flags set) and *null scans* (using packets with no TCP flags set); these are used to penetrate certain firewalls that interfere with other types of scans or to gather information about firewall rule sets. UDP scanning is somewhat more difficult: ICMP errors in response to a packet sent to a UDP port imply that it is closed, UDP responses suggest that it is open, and no response at all can mean anything.

2.3.2 0-Day Exploits

When the “good guys” learn of a new vulnerability in a computer system, the typical response is to try to fix it and to learn how to detect exploitation attempts. Usually, vulnerabilities are caused by defects in software implementations, which are ideally corrected by patches written and made available by the vendors or maintainers of the vulnerable software within a few hours to a few weeks. If no patch is immediately available, then instructions on how to prevent or lessen the impact of a successful exploitation are frequently published; these may involve disabling certain functionality or even disabling vulnerable services entirely.

When a new vulnerability is discovered, the practice of *responsible disclosure* is to first privately inform the vendor or maintainer of the vulnerable software of the existence of the flaw and wait until a patch or work-around is available before publishing details of the flaw. Unfortunately, the “bad guys” rarely follow this practice: new vulnerabilities are often kept secret or circulated privately among the “hacker” community, not becoming known to the security community until long after automated tools to exploit them are available. Newly discovered vulnerabilities to which the security community has not yet had the chance to react are known as *0-day vulnerabilities*; methods to exploit them are known as *0-day exploits*.

Defending against 0-day exploits is no easy task: since their existence is not known to defenders, no reactive measures can be taken. The only option is proactive security: using multiple layers of security so that a breach in one exposes little (a practice known as *defense in depth*) and aggressively auditing potentially vulnerable systems to attempt to detect flaws before the bad guys do. Unfortunately, defense in depth is not always easily implemented and many protocols and software systems are too complex to effectively audit: many flaws in common software have gone undiscovered for years. Nevertheless, short of disconnecting systems entirely, these are the only defenses available against 0-day attacks.

2.3.3 Worms and Malware

Malware is a generic term for any form of malicious software. Malware may take many forms, the best known of which is the *virus*. Two forms of malware particularly relevant to this thesis are *worms* and *rootkits*.

A *worm* is a program that replicates itself to new hosts across networks by scanning for targets and infecting them [Ayc06]. Worms usually gain access to computer systems by exploiting user errors or software vulnerabilities or misconfigurations, and once inside, continue scanning for new targets. This allows worms to spread exponentially. The `Sapphire` worm of 2003 infected 90% of all vulnerable hosts within 10 minutes [MPS+03] of release; theoretically, some worms could infect nearly all available targets in under a second [SMPW04]. Worms are usually non-specific as to what hosts they infect; they simply use horizontal port scanning to locate vulnerable hosts. On hosts that they infect, many worms install additional software, such as rootkits, backdoor software that allows the worms' creators to issue commands to infected computers, or spyware that attempts to collect passwords and credit card numbers. Worms are often written to exploit 0-day software vulnerabilities. This, combined with worms' rapid propagation, makes reactive security measures largely useless against worms.

Rootkits [HB06] are software that is used to hide the presence or activity of other software. These can take the form of modified userspace programs or kernel drivers that modify the behaviour of other drivers under certain circumstances. Common uses are to hide certain files from directory listings or certain processes from process listings; these files and processes typically belong to other forms of malware. In general, rootkits can be used to modify just about any form of operating system behaviour.

2.3.4 Denial-of-Service Attacks

A *denial-of-service (DoS) attack* is simply any attack that results in the degradation of the quality of service of a targeted system, limiting or denying its use to legitimate users. DoS attacks can range from the unintentional [wik07] to the criminal [Vij04], and from single-packet logic attacks [Ken97] to massive resource-consumption attacks [Gib05]. Logic attacks leading to denial-of-service conditions are usually caused by software bugs and are easily correctable; more interesting (and more difficult to defend against) are the resource-consumption attacks.

Resource-consumption attacks can attempt to saturate network links, burn CPU time, fill available memory, or use up other scarce resources. If the target host has sufficiently limited resources, then a single host may be able to effect a resource-consumption attack, but more often, several hosts need to cooperate to bring down a target, in what is known as a *distributed denial-of-service (DDoS) attack* [LRST00]. A common type of resource-consumption attack is the *SYN-flood* [dri96], in which attackers attempt to fill TCP data structures on targets with half-open connections, preventing them from accepting any new legitimate connections. In order to prevent targets from blocking traffic from attackers, SYN floods normally employ random spoofed source IP addresses. Since targets will attempt to open TCP connections with all of these apparent sources, a side effect of SYN floods is SYN-ACK packets being sent all over the Internet; this is known as *backscatter* [MSB+06].

2.4 Firewalls and Defensive Technologies

Like the defenses of ancient cities, defensive network technologies are primarily based on keeping intruders out and identifying them when they manage to gain entrance. *Firewalls* are the walls themselves - they restrict the traffic that is allowed to enter and leave a protected area. *Intrusion Detection Systems* (*IDSs*) and related technologies take a more active role in attempting to detect and identify attacks that are being attempted or have succeeded, making them more like guards. Other technologies, such as *Network Address Translators* (*NATs*) and *Virtual Private Networks* (*VPNs*), also have roles in securing networks.

2.4.1 Firewalls

Bellovin and Cheswick [BC94] defined the following three design goals for firewalls:

1. All traffic from inside to outside, and vice-versa, must pass through the firewall.

2. Only authorized traffic, as defined by a local security policy, will be allowed to pass.
3. The firewall itself is immune to penetration.

Two architectural designs are possible to satisfy the first requirement: either firewalls are situated on all network paths in and out of protected networks (*network* or *perimeter firewalls*), or firewalls are situated on the protected hosts themselves (*host firewalls*). Local security policy differs from place to place. Firewalls are most frequently deployed as proactive security mechanisms that limit the traffic that is allowed to enter the protected host or network in order to block attacks: this is known as *ingress filtering*. Firewalls can also be used for *egress filtering*: restricting outbound traffic in order to limit information leaks, the spread of malware, and the outside resources that can be accessed. In general, firewalls can be used to enforce a number of types of access-control policies on both inbound and outbound traffic:

1. what services are available,
2. who may access those services,
3. from where those services may be accessed,
4. when those services may be accessed, and
5. how those services may be used.

There are many different types of firewalls, differing in how they filter traffic and what information is available to them. The properties of each are summarized in the following sections; more information is available in [CBR03].

Packet filters

The simplest type of firewall is the packet filter [CBR03, p. 176], which examines packets one at a time at the network layer and decides to accept or reject them based on available information. Packet filters typically have no knowledge of application-layer protocols, but can make decisions based on link-, network- and transport-layer protocol headers and the network links on which packets arrive. Some packet filters are also capable of making decisions based on the current time, the volume of data transferred to or from a host within a time frame, the number of packets matching a rule within a time frame, and other conditions not directly related to the packets themselves. Common uses of packet filters include

- preventing hosts outside the local network from connecting to an internal server,
- blocking packets with obviously spoofed source addresses (such as internal addresses on packets on an external network interface),
- blocking tiny IP fragments, packets using source routing, directed broadcasts, and other suspicious packets, and
- preventing employees from using peer-to-peer file-sharing software.

Packet filters are simple to implement and require few resources; they are found on many network devices (bridges, routers, etc.) and older operating systems (such as Linux 2.2's *ipchains* [Rus00]). Packet filters are good at limiting what services are available from where and when, but are poor at tracking connections and have little knowledge of who is using network services or how they are being used. For instance, TCP clients typically connect from high-numbered ports, whereas servers listen on low-numbered ports. If a firewall wants to allow users to connect to external web servers, then it must allow outbound TCP traffic to port 80. However, responses from these web servers must also be allowed through the firewall. Simply allowing all packets originating at port 80 through is not adequate; anyone could run a program on port 80 and bypass the firewall entirely. The usual solution is to allow all packets with the ACK flag coming from port 80 and destined to high-numbered ports, but this will still allow some packets through the firewall that are not part of any legitimate connection. Since packet filters have no knowledge of application-layer data, they can't prevent hapless users from downloading viruses from a web pages, nor can they prevent malicious insiders from sending trade secrets to third parties. They also have no knowledge of who or what is sending the packets they see. Packet filters can only identify users and programs by IP addresses and ports: they can't distinguish between worms and web browsers. Newer operating systems and firewall appliances generally use more elaborate forms of firewalls that address some of these problems.

Some packet filters attempt to process application-layer data as well, in order to identify the application protocol being used or to detect and reject malformed or malicious application-layer messages. This is known as *deep packet inspection* [Dub03]. Unless deep packet filters perform some re-assembly of packets, they can be confused by pathological fragmentation. They are often considered to be a form of Intrusion Prevention System (see Section 2.4.2).

Stateful packet filters

As opposed to simple packet filters, which consider packets independently, *stateful* (also known as *dynamic*) packet filters [CBR03, p. 188] consider packets to be parts of connections. Connections can be accepted or denied; packets associated with accepted connections are generally accepted with little further inspection. Stateful packet filters are also capable of the sorts of analysis performed by simple packet filters: connections may be filtered based on source, destination or any other packet characteristics, and packets may be blocked even if they belong to valid connections.

How packets are associated with connections differs based on the transport protocol being used. TCP connections are easily tracked; a partial implementation of the TCP state machine is enough to determine when connections open and close. More advanced stateful firewalls may track TCP window sizes and other aspects of TCP to more accurately determine if a packet belongs to a given connection. UDP doesn't define "connections" on its own, so UDP connections are usually tracked using intervals between packets; connections are opened for any UDP packet traveling between a pair of addresses and are closed when a timeout is reached without any more packets between the addresses being observed. ICMP error messages are usually treated as part of the TCP or UDP connections associated with the packets that triggered them; other types of ICMP messages (such as "pings") may be treated as connections of their own. Other protocols' connection states are tracked in manners appropriate to those protocols. Since the underlying network protocol (IPv4) is fundamentally unreliable, connection-closing messages from TCP and other stateful transport protocols may never be received, so stateful packet filters may also use timeouts to close such connections.

Application-layer semantics can also play a role in connection tracking. Several common application-layer protocols, such as FTP, make use of more than one transport-layer connection. Stateful firewalls often have extensions that perform limited parsing of application-layer messages to treat these secondary connections as parts of the connections that created them, greatly simplifying firewall rules.

The ability to track connection state gives stateful packet filters many advantages over simple packet filters: it allows rule-sets to more accurately reflect legitimate traffic patterns and reduces the chance of passing packets that should be rejected. However,

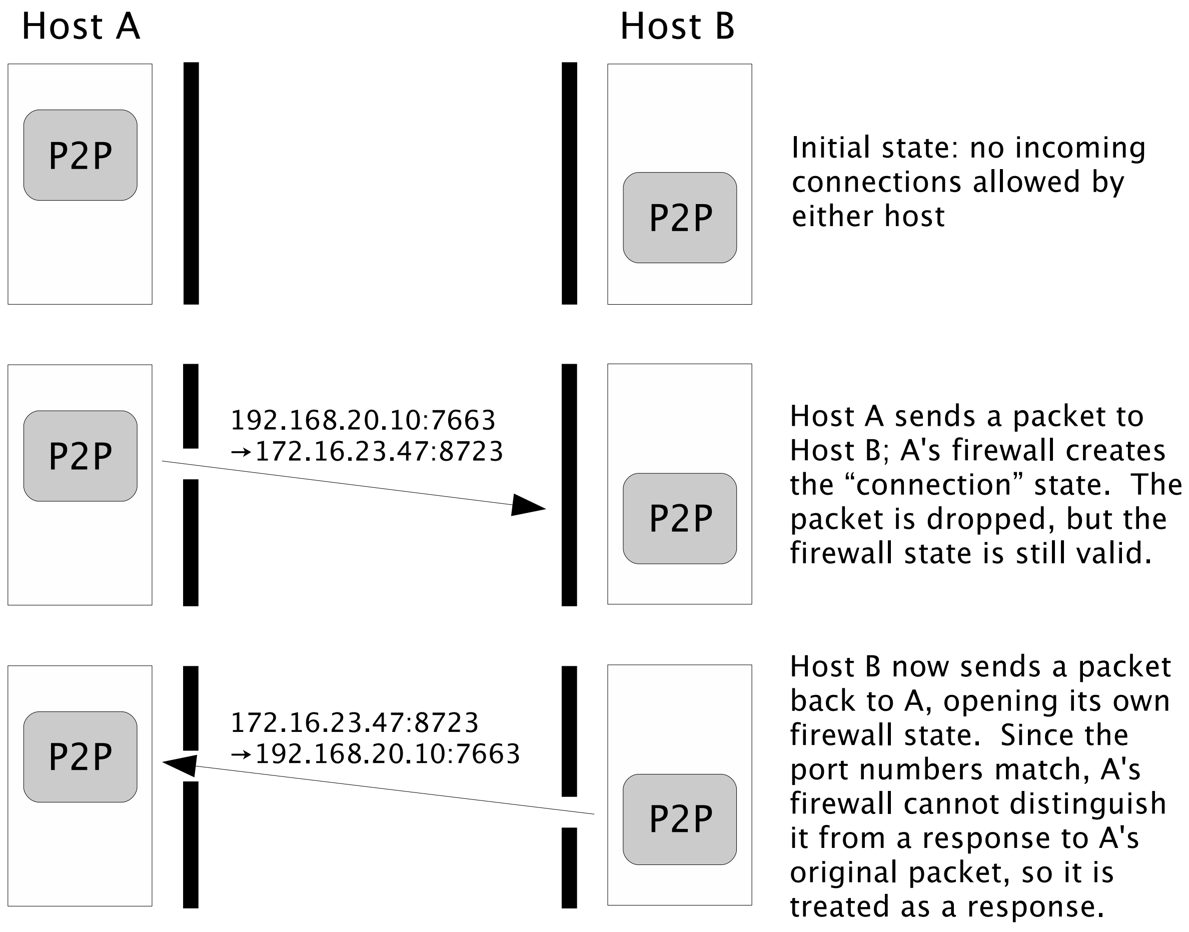

Host A
Host B
P2P
P2P
Initial state: no incoming connections allowed by either host
P2P
192.168.20.10:7663 →172.16.23.47:8723
P2P
Host A sends a packet to Host B; A's firewall creates the "connection" state. The packet is dropped, but the firewall state is still valid.
P2P
172.16.23.47:8723 →192.168.20.10:7663
P2P
Host B now sends a packet back to A, opening its own firewall state. Since the port numbers match, A's firewall cannot distinguish it from a response to A's original packet, so it is treated as a response.

Figure 2.2: Flaw in UDP state tracking

this comes at a cost: stateful packet filters are far more complex and require large amounts of memory for state tracking. Additionally, the methods of generating state information for stateless protocols like UDP can be fooled. Two hosts whose firewalls allow outbound UDP packets between their addresses but deny inbound packets can nevertheless establish a connection by both attempting to connect to each other, as in Figure 2.2; many peer-to-peer applications use this technique to bypass firewalls [Sch06a]. As with simple packet filters, stateful packet filters have limited knowledge of application-layer protocols or of the actual users and programs sending and receiving the packets that they examine.

Most modern operating systems come with built-in stateful packet filters, as do most dedicated firewall appliances on the market. Typical examples include Linux's `iptables` [Wel06a] and OpenBSD's `pf` [KHM06]. Details of how `iptables` tracks connections are available in [And06].

Application filters

Packet filters are only capable of identifying users by IP address; all users of shared computers are treated alike. Likewise, packet filters are only capable of identifying programs by port numbers and treat all programs using the same ports alike. They cannot implement per-user access restrictions or limit network access to specific applications. There is good reason for this: the information necessary to implement such filters, user and application names, is not included in IP packet headers and thus is not available to packet filters. This information is generally only available at the hosts sending or receiving the packets in question, thus preventing any form of network firewall from using it. However, host firewalls do have access to this information; most modern host firewalling software (including PC firewalling packages, such as Check Point's `ZoneAlarm` [Zon07]) make use of it to implement user and application filtering[1] [CBR03, p. 226], as well as stateful packet filtering.

Unlike packet filters, application filters can tell the difference between legitimate software and malware that uses the same ports: for instance, spyware that communicates with valid HTTP messages over port 80/TCP would be indistinguishable from a web browser to a packet filter, but an application filter would be able to tell the difference. Application filters can also be used to block certain software that is known to have questionable security records, even if preventing its installation is not feasible or

[1]There is no standard term for this type of firewall. Some authors use the terms "application-based filter" or "program filter"; others refer to what I call "application gateways" as "application filters".

it is integrated into an operating system and cannot be removed, while still permitting network access to functionally equivalent software. They can also be used to prevent any network access when no user is logged in.

Application filtering is primarily used by egress firewalls, but may also be used on ingress.

Circuit gateways

Circuit gateways [CBR03, p. 186] are proxy servers that run at the transport layer. They don't allow end-to-end connections through themselves: instead, protected clients are required to connect to a gateway and communicate with it using a special protocol to request connections to the outside world. The gateway then makes the requested connection (if it is allowed by the gateway's security policy) and forwards data between the two connections. A similar approach is taken with local servers that want to make services available from outside the protected network.

Because circuit gateways re-assemble data at the transport layer before passing it in or out, they can easily protect against IP-layer attacks such as source routing and pathological fragmentation. They are even able to exchange data between the IP-based Internet and local networks using other network-layer protocols or between logically disconnected networks. However, their abilities to filter traffic are not much different than those of stateful packet filters, despite requiring significantly more resources. Circuit gateways may require user names, passwords, and other information from clients before creating connections to the outside, potentially allowing them to filter based on local user and application, but they still have no knowledge of remote users, nor of application-layer protocols.

Circuit gateways are generally used as network firewalls; host-based circuit gateways would be pointless. A common circuit gateway system is SOCKS [KK92, LGL$^+$96].

Application gateways

Application gateways [CBR03, p. 185] work as proxies at the application layer. As with circuit gateways, connections are not end-to-end but are made by the gateways on request. In addition to the information available to circuit gateways, application gateways are able to filter based on application-layer information, and possibly also user and application information, but, due to the diversity and complexity of application-layer protocols, separate gateways are required for each application.

Typical uses of application gateways are to filter for malware, malformed messages that could represent attacks against servers, and content that should not be entering or leaving the protected network, such as pornography or trade secrets. The trade-off for this power is that application gateways require large amounts of memory and processing time and frequently cannot be made fully transparent to users. Most common web proxies and SMTP servers are able to function as application gateways for their respective protocols; gateways for more obscure or proprietary protocols can be difficult to find.

Distributed firewalls

Traditionally, network firewalls have been the most common method for protecting large numbers of hosts; they are easy to deploy and manage. However, they do have a number of disadvantages:

- Unauthorized or improperly protected network links can allow users to bypass firewalls entirely.
- Laptops and other roaming hosts move in and out of protected areas; while outside of the network perimeter, they are not protected by its firewalls.

- Firewalls at network choke-points are single points of failure and possible performance bottlenecks.
- In networks with more than one route to the Internet, it is possible for packets to leave over one link and their responses to arrive on another. Stateful packet filters and gateways will not be able to re-assemble connections under these conditions.

As a way of improving on these weaknesses, Bellovin [Bel99] proposed using *distributed firewalls* in which each host in a protected network runs its own firewalling software but receives configuration from a central management server. Such a system has a number of advantages:

- The firewall is independent of network topology; hosts no longer need to be classified as "inside" or "outside" of a security perimeter.
- There is no longer a single point of failure or performance bottleneck; if one host goes down, others are unaffected.
- The firewall can base its decisions on additional information that is not available to most network firewalls, such as user and application names and dynamically-assigned ports, without needing to resort to computationally expensive application-layer processing.
- Hosts that move between networks or otherwise change addresses frequently, such as laptop computers, are easily protected, regardless of their current locations.

The independence of network topology provided by this model is particularly important: many of the threats faced by modern networks are from the inside (spyware, worms carried into protected areas on laptops, malicious insiders, etc.). The traditional model of walled-in networks with barbarians outside that forms the basis of most firewall architectures provides little to no defense against internal attackers.

One disadvantage of distributed firewalls is that hosts cannot make any assumptions about spoofed IP addresses on the local network and don't know which hosts to trust. Bellovin [Bel99] suggests working around this by using IPsec to cryptographically verify host identities. In any case, this is a stronger form of authentication than relying on IP addresses, and works even if IP addresses change. However, IPsec adoption is nowhere near the point of making this practical over domains larger than corporate networks, so hosts will still have to rely on address-based authentication for communication with hosts outside of their administrative domains. Also, many older and less-capable systems are still in use for which IPsec support is not available.

Other disadvantages of distributed firewalls lie in the difficulty of managing configuration. Management servers may either attempt to push configuration updates to all hosts, in which case any hosts that are currently unreachable do not receive the updates, or rely on hosts polling for and pulling updates, in which case hosts that neglect to poll regularly (or are prevented from doing so) do not receive them. It is difficult to ensure that all hosts that should be running firewalling software actually are; hosts whose firewalls are malfunctioning or not present may not be protected at all. Finally, some malware (such as Y3K Rat 1.6 [Whi01]) attempts to disable local security software on hosts that it infects; if this succeeds, then the firewall is disabled at the moment when it is needed most. In one implementation [IKBS00] of Bellovin's concept [Bel99], simply disabling the policy daemon will disable the firewall. Since there are many fewer avenues for malware to be executed on network firewalls, this attack is much less of a threat for traditional firewalling systems.

Hybrid firewalls

Bellovin [Bel99] recognized some of the weaknesses of distributed firewalls in his original paper and suggested that *hybrid firewalls*[2], combinations of distributed and choke-point elements, could address some of them. There are a number of ways that such a system could be structured:

- A network of non-mobile hosts, each protected by distributed firewalling software but without cryptographic authentication of peers, could use simple network firewalls to drop inbound packets with spoofed internal addresses. This would prevent many spoofing attacks under the assumption that no internal hosts are engaging in spoofing.

- A network that uses distributed firewalling could employ network firewalls as a secondary protection mechanism: this would protect hosts whose firewalling software is missing, malfunctioning, or not configured correctly. If the network firewall failed, it could fail open without leaving the network completely exposed.

- Networks that contain both mobile and stationary hosts could use network firewalls with knowledge of network topology to protect stationary hosts, while using a distributed firewall and an IPsec gateway to allow protection for and secure communication with mobile hosts.

2.4.2 Intrusion Detection Systems

Unlike firewalls, intrusion detection systems (IDSs) take a reactive approach, attempting to identify attacks in progress or to detect evidence of past attacks and taking appropriate countermeasures. IDSs can use either statistical methods or pattern-

[2]The term "hybrid firewall" has been used by many authors to refer to many different things, including hardware-accelerated packet filters and stateful firewalls (as a hybrid of packet filters and circuit gateways).

matching to detect intrusions, can operate by monitoring logs or events on individual hosts or by monitoring network traffic, and can run periodically or in real time [DDW99].

Responses made by IDSs depend on the type of IDS and the type of attack. Single-host signature matchers, such as some anti-virus engines, that detect successful intrusions may attempt automated clean-up operations or may simply notify an operator that clean-up is needed. Worm traffic detected on a local network may indicate that an intrusion has taken place; an IDS detecting such an event may attempt to isolate the infected host(s) in order to contain the worm [MSVS03]. Port scans detected at a perimeter firewall may result in all traffic from the scanning hosts being blocked [Sol98] or subjected to rate limitations [Wil02]. Unfortunately, automated responses can often cause unintended side effects: false positive detections may cause more harm over the long term than undetected intrusions and, even when attackers cannot hide from IDSs, they may attempt to trigger inappropriate responses. Signature-based detectors can be fooled into making inappropriate responses by deliberately matching the signature of different attacks, port scanners can spoof large numbers of source addresses in order to make port scan detectors block off large segments of the Internet, and worms can use spoofed source addresses to trick IDSs into quarantining too many hosts [PN98]. In many cases, the only safe automated response to intrusions is to inform the operator.

Intrusion Prevention Systems (IPSs) use statistical or pattern-matching methods to attempt to automatically detect and disrupt attacks in real time [KVV05]. Due to their similarities in function, some IDSs (including Snort [MJ]) are also capable of functioning as IPSs. Many “deep” packet filters are essentially combinations of stateful packet filters and simple IPSs [Ran05].

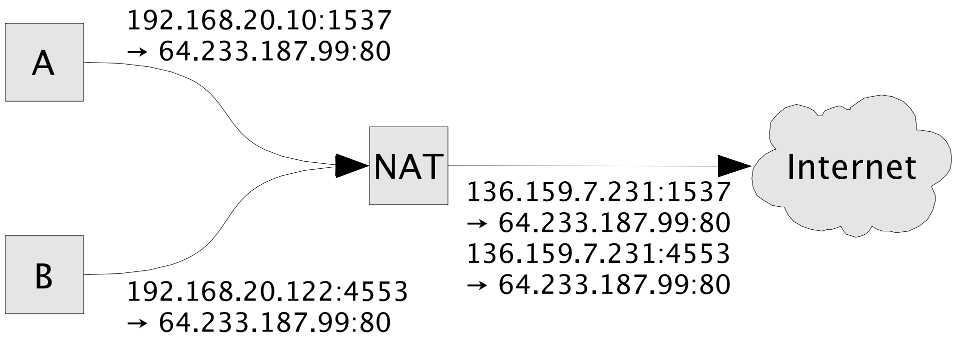
2.4.3 Network Address Translators

Network Address Translators (NATs) [SE01, SH99] are devices that re-write the source IP addresses of traffic leaving a network and the destination addresses of traffic entering it. This process is known as *network address translation*.[3]

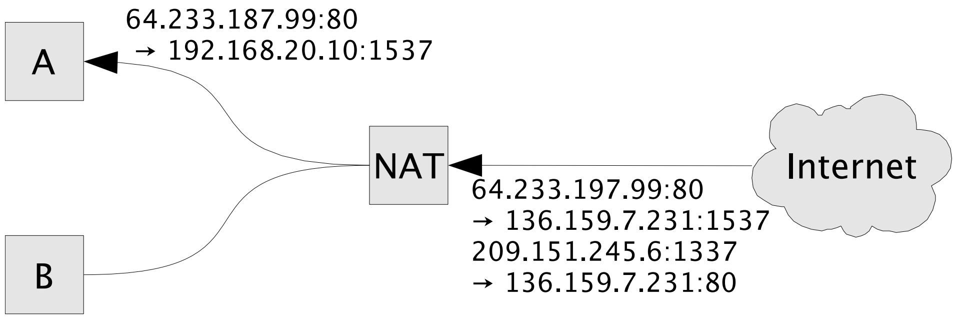
The most common reason using for network address translation is to share a single IP address among many devices. Devices on a network are assigned IP addresses that are only valid inside the network (usually chosen from the RFC 1918 private address blocks [RMK$^+$96]); they can communicate amongst each other using these addresses, but the addresses are not recognized by the Internet at large. In order to connect such a network to the Internet, all traffic entering and leaving the network is routed through a NAT, which re-writes the source addresses of all outbound packets to its own external IP address (as in Figure 2.3(a)), which *is* valid on the Internet. In order for responses to outbound packets to be correctly received, the NAT keeps state information allowing it to re-write the destination addresses of packets it receives to the appropriate internal addresses (as in Figure 2.3(b)). In such a system, the internal addresses are known as *private addresses* and the externally-valid address of the NAT is known as the *public address*. Since TCP and UDP connections are typically identified by the combination of source and destination addresses and source and destination ports, such a NAT cannot handle two internal hosts making connections to the same external host and port from the same source ports; to work around this, a variation called *network address port translation*, which also re-writes source port numbers on outbound packets, is used. Other variations on this system allow NATs to use more than one public address; some NATs support one-to-one mappings between private

[3]Confusingly, the acronym "NAT" is often applied to either the noun phrase "network address translator" or the verb phrase "network address translation". In this document, I shall use the former meaning.

and public addresses.



(a) Outbound traffic



(b) Inbound traffic

Figure 2.3: Traffic passing through a NAT.

As a side-effect of network address translation used in this manner, outside hosts cannot normally initiate connections through a NAT: without state information to indicate which inside host should receive a packet, the NAT will simply drop it (as with the 2nd packet arriving at the NAT in Figure 2.3(b)). Therefore, though NATs are not designed as network security devices, they can function as simple packet filters. Since networks employing NATs require that all outbound traffic pass through them, they are ideal locations to place network firewalls; most available NATs have at least limited packet-filtering capabilities, and many existing firewalls can also function as NATs. The state-tracking mechanisms used by NATs are similar to those used by stateful packet filters and have the same weaknesses with regards to stateless transport protocols.

Network address translation can also be used in reverse in order to share a single IP address among many network servers or to load-share between them: this technique is often called *port forwarding*. The term *destination network address translation* is also used; in this case, re-writing source addresses of outbound packets is known as *source network address translation*.

NATs resemble circuit gateways in that outside hosts see only the NATs' addresses, rather than those of the protected hosts, but differ in that protected clients and servers communicate directly with outside hosts. NATs only re-write addresses; packet boundaries and other network-layer semantics are preserved.

2.4.4 VPNs and Encrypted Channels

Many attacks against network services are only possible because of IPv4's lack of protection against sniffing and modification and its lack of verification that packet source addresses are correct. All of these problems can be solved by proper application of cryptography: encryption of payload data makes sniffing useless; data integrity checks can prevent insertion and modification attacks, and user or host authentication allows spoofed source addresses to be detected.

Virtual Private Networks (VPNs) are private networks that allow confidential communication over insecure public networks. VPNs are normally implemented by tunnelling ordinary, insecure protocols inside encrypted channels on top of standard network and transport protocols. This can be done at different protocol layers. TLS [DR06] is normally used for encrypting application protocols on top of TCP, but can also be used to tunnel IP on top of TCP or UDP [Yon06]. IPsec [KS05] and other protocols tunnel IP packets inside a secure channel on top of IP. All of these are complicated cryptographic protocols that provide host authentication, key exchange, and confidential, integrity-protected channels.

The benefits of VPNs and encrypted channels come at a cost. Encryption is CPU-intensive, adding a cost to communication that may be prohibitive for low-powered or busy hosts. VPN software often requires operating system support, which may not be available for some platforms. Tunnelling of any sort makes it impossible to block access to applications by port number; a growing trend on the Internet is the tunnelling of various protocols on top of HTTP [Alb04, BP04]. Furthermore, encryption makes firewalling difficult: unless a firewall knows all encryption keys in use, it cannot decrypt tunnelled traffic to make filtering decisions. This leaves security up to destination hosts themselves, just as it would be without firewalls; some firewalls may block encrypted traffic for this reason. Thus, it is beneficial to use a technology that provides the minimum required security at the minimum cost, rather than using more powerful VPN systems, wherever possible.

Chapter 3

Stealthy Authentication Mechanisms

Knock, knock
Who's there?
Doris
Doris who?
Doris locked, that's why I knocked!
– `http://www.knock-knock-joke.com`

A common goal in firewall policy design is to limit which remote users can connect to particular services. There are many reasons for implementing such access restrictions, including

- strengthening a defense in depth: adding an extra layer that attackers must break through before reaching anything important,
- protecting systems with known unpatched vulnerabilities from attackers until patches are available, while still allowing access to certain authorized users, and
- adding a measure of user authentication to legacy or proprietary systems with inadequate integrated security measures.

A side benefit of limiting access to services at a firewall in this manner is that unauthorized users will have difficulty even learning of the existence of the protected service; port scans against the service from unauthorized hosts cannot tell the difference between a port that appears closed because nothing is using it and a port that appears closed because a firewall is intervening. This adds a degree of security to the service: attackers are unlikely to attack services that they don't know about, so measures that make services more difficult to detect will reduce the number of attackers aware of the service and therefore the number of attacks made. Since the number of attackers is finite, a reduced number of attackers and attacks reduces the probability of a security breach.

Regrettably, most existing firewalls aren't very good at implementing such restrictions. One common method is to assume that trusted users connect only from certain small sets of trusted computers with known IP addresses, and allow connections only from these addresses. This has many limitations: attackers can spoof trusted IP addresses or hijack trusted hosts and trusted users may attempt to connect from hosts not in the trusted set. Since many computers on today's Internet do not have static IP addresses, instead relying on DHCP servers to assign addresses that change over time, allowing access from one particular computer may require granting access to many thousands of IP addresses; in the case of a worm outbreak, there is a strong chance that an attacking worm will reside on a computer with one of these addresses. Adjusting the set of trusted IP addresses usually requires manual re-configuration by a firewall administrator. The other method available is to use a world-accessible service that uses some form of user authentication to identify trusted users and grant them temporary access to the protected service, by creating a temporary association between the trusted user and its IP address. Unfortunately, this approach typically has drawbacks as well. Exploitable flaws in security and authentication software are discovered regularly [UC06a, UC06b, UC07a, UC07b]. Since the authentication service is visible to the world, it can be attacked by anyone, and since it controls firewall rules, successful attacks could be used to completely bypass the firewall.

Clearly, since IPv4 headers include no information about users, there is little that can be done with the approach of filtering by fixed sets of source IP addresses. The model of authenticating to a world-accessible service and requesting access has potential, but needs enhancements to correct the weaknesses described above. In particular:

- Since the whole point of the firewall access restrictions is to keep unauthorized users from connecting at all, the authentication service should be no easier for attackers to communicate with than the protected services. It must still be world-accessible, but it could be hidden in some manner. One way to accomplish this is for communication with it to use a covert channel.

- Since any attacker who discovers and is able to communicate with a hidden authentication service has already displayed significant skill and resources, the authentication service must be cryptographically secure. Also, it should be as simple as possible, so that it can easily be audited and reasonable assertions made about its vulnerability to attack.

Using a complex, highly visible mechanism would present a risk of attack not significantly different than that of the original, now protected, service; using a hidden service that provides strong authentication and is simple enough to easily audit provides a target that is both less likely to be compromised if attacked and less likely to be attacked at all.

This chapter will begin by discussing covert channels over networks and then describe several ways that covert channels could be used by firewall authentication services. Two existing techniques, *port knocking* and *single packet authorization*, will be discussed in detail, including their design issues, strengths, weaknesses, and current implementations.

3.1 Covert Channels over Networks

Covert channels over networks are somewhat different from covert channels between processes on the same system [Lam73]; networks are *designed* to facilitate communication. However, it is possible to covertly encode information into separate, unrelated streams [Gir87] or into unremarkable packets that will normally elicit no detectable response. For this discussion, channels designed to resist detection by attackers passively monitoring traffic (i.e., they remain undetectable while information is being exchanged) will be termed *passive-covert channels*, whereas those designed to remain undetectable to active probing (i.e., inactive communication endpoints cannot be discovered) will be called *active-covert channels*. These terms are not mutually exclusive; a channel can be both active- and passive-covert.

In order to qualify as an active-covert channel, anything that an attacker could be expected to send to the address of a covert channel endpoint must not trigger any remotely-observable behaviour changes or any responses different from those of inactive addresses. Traditionally, network services exchange data through open TCP or UDP ports. However, TCP requires that connection-establishing SYN segments be acknowledged; this makes any TCP service visible to port scans [SHM02] and hence not covert. UDP services are not required to respond in any way, but many firewalls respond to UDP packets sent to closed ports with ICMP *port unreachable* messages; no such message is sent in response to UDP packets sent to open ports, so UDP services are often also visible to port scans. This suggests that only UDP services, and only on some hosts, are suitable for implementing active-covert channels. However, it is also possible to communicate over closed network ports. Firewall authentication systems using such communication channels are generally known as *Single Port Authorization*, or *SPA*, systems: these will be discussed further in Section 3.3.

In the context of the firewall authentication service being discussed, it cannot be assumed that streams of packets between clients and servers already exist, so options for creating passive-covert channels seem very limited. Since packets sent to or from such services will be visible to anyone monitoring the traffic at the servers, any channel that creates and accepts packets, whether over open ports or closed, cannot normally be considered passive-covert. However, passive-covert channels of a sort can be constructed by encoding information as packets resembling Internet background noise, which will appear uninteresting to packet sniffers (assuming that the volume of actual background noise remains significantly louder than that of traffic masquerading as background noise) and thus be ignored.

Fortunately, the Internet's background noise is both loud and varied [PYB+04]. Any host connected to the Internet can expect to receive significant amounts of unrequested and unwanted traffic from sources including

- worms seeking targets through horizontal port scans [SPW02];
- "hackers" gathering information about target systems through vertical port scans [Fyo97];
- backscatter from DDoS attacks [MSB+06];
- Windows Messenger spam [LUR03]; and
- attacks against running services [PYB+04].

This traffic comes in many forms, including TCP SYN and RST segments, UDP datagrams of various sizes, and ICMP *echo-request*, *echo-response*, and *port unreachable* messages. This variety provides ample opportunities for creating passive-covert channels.

Given this, a covert channel could be constructed by encoding data in packet payloads resembling one of the above. However, most of the packets in this unwanted traffic do not normally carry payloads (including most port scans and DDoS backscatter), and most of those that do are easily matched to various categories by intrusion detection systems. Any attempt to use this method to construct a passive-covert channel must therefore take care to ensure that its packet payloads resemble legitimate background noise. Passive-covert SPA systems can be created using such a method.

An alternative is to encode information in the short packets typical of port scans and DDoS backscatter. Since these packets usually carry little to no payload, information must be encoded in packet header fields. A wide variety of mechanisms for encoding data in Internet packets headers exist [BR], but not all are suitable for use in this scenario. Channels using fields in MAC-layer protocols [Wol89] cannot be used over routed networks. Channels involving manipulations of source IP addresses or ports cannot safely be used, since NATs will re-write source addresses and ports and routers may drop packets with source addresses invalid for the interface on which they arrived. Neither can destination IP addresses [Gir87] be safely used, since authentication servers may not be capable of receiving packets addressed to different hosts. Some channels, such as those based on IP fragmentation strategies [Ahs02], can carry only 1 to 4 bits of data per packet and are both unnecessarily slow and don't allow re-ordering on delivery (see Section 4.2). Additionally, most of these schemes, as well as most other possible {TCP|UDP}/IP header modifications, require that clients be privileged applications (which cannot directly manipulate most packet header fields on most systems); this may not be practical or appropriate. One header field remains that *can* be set by unprivileged applications: the destination port. Using TCP and

UDP destination port fields to transfer information is known as *port knocking*, and will be discussed further in Section 3.2. Since such traffic strongly resembles “normal” background noise (but see Section 3.2.5), port knocking can accurately be described as passive-covert.

3.2 Port Knocking

Port knocking originated as a mechanism for transmitting authentication information across closed network ports using a digital equivalent of secret knocks on a door. If the traditional mechanism of connecting a service to a TCP or UDP port and requiring the service to provide any necessary authorization checks is compared to propping a door open and putting a guard inside, then port knocking is equivalent to leaving the door locked and only opening it if a recognized knock sequence is heard; the guard, if present, can make a second authorization check.

To authenticate, a port knocking client encodes authentication information as a sequence of TCP or UDP port numbers and sends packets to each of these ports on a server; if the server is able to decode the port sequence and recognizes the authentication token, then it grants access to some service to the client. Port knocking servers don’t listen on any ports and don’t respond to anything sent by clients (without first receiving knock sequences that are difficult to guess), making them invisible to any type of scan, and thus active-covert. Typically, the firewalls on port knocking servers are configured to drop all packets sent to them without returning any responses. In most cases, this makes server machines themselves invisible to scans[1]. Moreover, to third-party observers, port knock sequences are difficult to distinguish from port

[1]Some routers are configured to send ICMP *host unreachable* errors when a packet is sent to a host that doesn’t exist; if the server’s gateway is configured thus, then the server cannot be made invisible to hosts outside of its local network segment.

scans [Ras06]. Given that any Internet-accessible computer can expect to be scanned several times per day [The01, YBU03], observers would consider port scans to be fairly unremarkable events, making port knocking passive-covert as well.

3.2.1 Authentication Using Port Knocking

The key differences between most existing port knocking systems are in how authentication is accomplished. There are three types of authentication systems used in port knocking systems today: plain-text, cryptographic, and one-time. All are based on the existence of some secret shared between client and server; this secret is assumed to be distributed by some out-of-band mechanism.

Plain-text port knocking

The simplest port knocking systems are ones where the knock sequence itself is used as a shared secret; knowledge of the secret implies that the user is authorized to access the protected service. If a client transmits packets to a specific sequence of server ports (for instance, 1145, 1087, 1172, 1244, and 1031, in that order), then the server performs some action (such as opening the SSH port to the client host, as in Figure 3.1). Possibly the first true port knocking server, `cd00r` [FX 00, SZ04], uses such a shared secret to open a root shell. Many other port knocking systems are capable of using this mode, including Krzywinski's `Portknocking Perl Prototype` (`PKPP`) [Krz06] and older versions of `fwknop` [Ras04].

When no authenticated users are connected, and no services are open, such a system is secure against active attacks over a network. The server never responds to any message that it receives, and its only remotely-observable behaviour change is when a valid sequence is recognized. If a sequence consists of 8 port numbers randomly distributed over a space of 256 ports, then an attacker must attempt $\frac{256^8}{2} \approx 9.2 * 10^{18}$

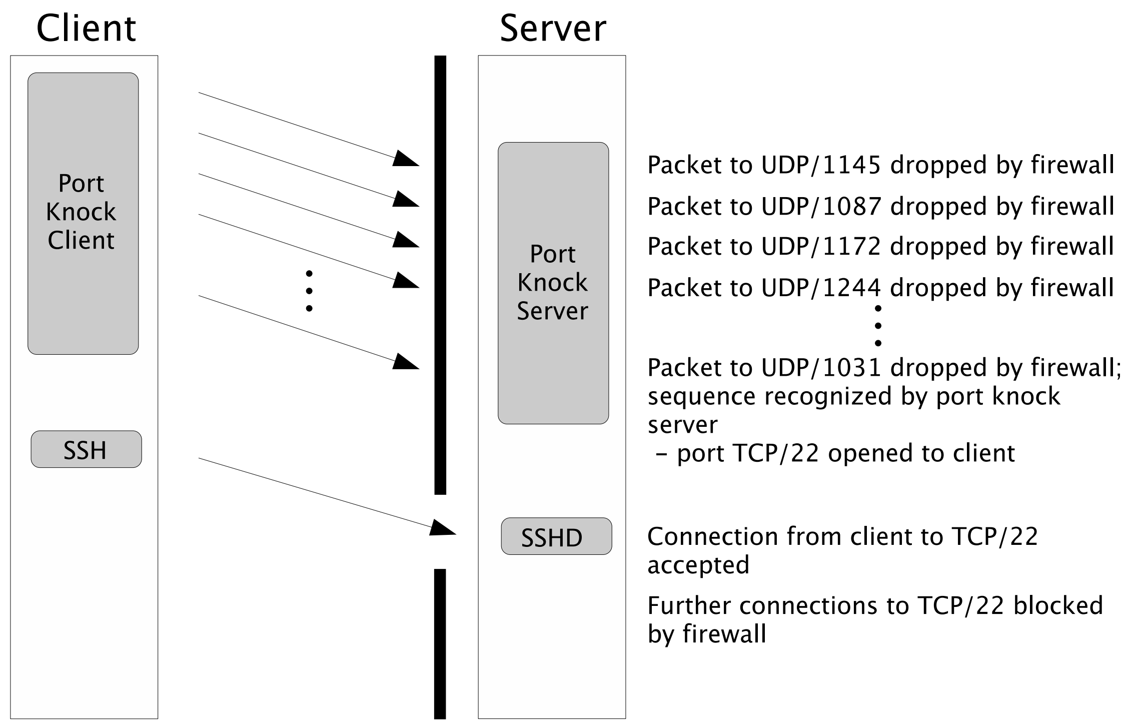

Client
Server
Port Knock Client
SSH
Port Knock Server
SSHD
Packet to UDP/1145 dropped by firewall
Packet to UDP/1087 dropped by firewall
Packet to UDP/1172 dropped by firewall
Packet to UDP/1244 dropped by firewall
Packet to UDP/1031 dropped by firewall; sequence recognized by port knock server
- port TCP/22 opened to client
Connection from client to TCP/22 accepted
Further connections to TCP/22 blocked by firewall

Figure 3.1: Simple port knocking example. A port is opened in the firewall in response to a specific port sequence.

sequences on average to guess the correct one. Since each sequence must be sent over the network (224 bytes per sequence using UDP), network bandwidth is the limiting factor, making parellelization useless. No foreseeable network technology will allow such a brute-force attack to be conducted in any reasonable amount of time, and any such attempt would be obvious to a monitoring system.

However, such a system can be trivially compromised by a passive sniffing attack. Since the authentication token is the knock sequence itself, which must necessarily be transmitted in plain text, and there are no limits on the time frame over which the sequence is valid, a sniffing attacker could read the sequence from the network and replay it at will to gain access to the protected service. Though this type of port knocking system can be assumed to be secure against worms and unsophisticated attacks, defense against more dedicated attackers with more sophisticated attacks requires a stronger form of authentication.

Cryptographic port knocking

Several attempts have been made to improve on plain-text port knocking by encrypting a message using a shared secret and encoding the result into a port sequence. A typical example is Doyle's system [Doy04], in which a client concatenates and pads its IP address, the requested server port number, and an "open"/"close" flag into a 64-bit message, encrypts it using the Blowfish cipher and a shared secret key, and encodes the result into 8 port numbers; the server decrypts the message and either opens or closes the requested port to the client's IP address, depending on the flag. Another is Krzywinski's `PKPP` [Krz06], which allows any combination of the client's IP address, the destination port, the current time, arbitrary bytes, random bytes, and checksums to be sent in plain text or encrypted using an arbitrary cipher and a secret key, optionally using a specified initialization vector.

Unfortunately, many of these systems are based on the flawed premise that encryption alone provides authentication. In fact, it *doesn't*. A client's ability to supply an encrypted message *can* prove to a server that it knows a secret key, but only if

1. the server can identify valid decrypted messages,
2. the server can associate the message with the client, and
3. the message cannot have been captured from an earlier exchange and replayed.

It is fairly easy for a server to identify valid messages: the usual method is to compute a MAC or hash covering relevant parts of the plaintext and append it to the plaintext before encryption. Upon receiving such a message, the server would decrypt it, recompute the MAC or hash from the plaintext, and verify that it matches the supplied value. An alternate strategy is to include some value that is known to the server, such as a *nonce* (a value that is used no more than once for a given purpose [MvOV96, p. 397]) or timestamp, in the plaintext; on receipt, a server would decrypt the message and verify that the expected value is present. Either condition ensures that the message was generated by a client in possession of the shared secret, assuming that a suitable encryption algorithm was used (there are pitfalls to each of these techniques when used with certain types of MACs or stream or block ciphers; see [MvOV96] for details). A message generated or modified by anyone else would decrypt to random garbage and would not (with high probability) contain the correct MAC, hash or nonce. Creating an association with the client can be accomplished by including the client's IP address in the ciphertext. It is more difficult to protect against replay; this usually requires some time-variant parameter (such as a nonce, sequence number or timestamp) to be included in the plaintext; servers need to maintain state to identify messages that have already been used or compare timestamps against the current time. Timestamp-based systems have additional challenges, which are discussed further in the next section. More information on encryption-based authentication systems is available in [MvOV96, ch. 10].

Doyle's system (and Krzywinski's, in most modes[2]), lacking any redundancy in its messages or checks against values known to the server, is incapable of detecting invalid messages, so it will decrypt and execute *any* message that it receives. Since the performance of firewalls tends to degrade significantly as the number of rules increases [Har02, KP05], an attacker could conduct a denial-of-service attack by sending random messages to a server, which will pollute its firewall tables with large numbers of random rules.

Furthermore, Doyle's system (and again, Krzywinski's in most modes) lacks any

[2]Krzywinski's system is advertised as a prototype, "not specifically designed for production environments"; given its many insecure modes, it should *not* be used in production without extreme care taken in its configuration.

mechanisms for detecting replayed messages. Upon receipt of a replayed “open” message, a server will open whatever port was originally requested to the original client’s IP address. Although attackers usually cannot control the source IP addresses used by valid clients, they may be able to assume those addresses at later times, either by IP spoofing [dVdVI98], being assigned those addresses by DHCP servers, or simply taking control of clients’ computers. It is much easier to abuse “close” messages: a “close” message could be replayed to cut off a valid client’s connection, making an easy denial-of-service attack.

A final weakness in Doyle’s and Krzywinski’s systems (in some modes) is that the contents of the encrypted messages are known to attackers. Anyone capable of sniffing messages from a network can determine the client IP addresses, server port numbers, and the actions performed through traffic analysis. With this information in hand, known-plaintext attacks [Sch06b] can be made against weak encryption keys. If an attacker can determine the key, then it could craft any authentication message that it likes.

One-time port knocking

The third, and generally most secure, category of existing port knocking systems consists of those that use one-time passwords (OTP) or timestamps. Due to the one-time nature of the authentication messages generated by these systems, they are resistant to replay attacks and, when properly implemented, give no information about any master keys in use.

There are four basic types of one-time authentication suitable for port knocking:

- **Time-dependent authentication**.

  Quite a few existing port knocking systems incorporate timestamps into some sort of authentication message, including `PortKnockO` [SF06] and `sig2knock`

[TC04]. Servers in such systems can detect replayed messages by checking the current time against messages' timestamps; messages with old (or future) timestamps are deemed to be invalid.

The main disadvantage of these systems is that they rely on low transmission delay and low clock drift between clients and servers. In order to account for transmission delay and imperfect clock synchronization, any timestamp-based authentication token must be valid within a window of at least several seconds in either direction; windows of up to a few minutes may be required for some systems. Within this window, captured tokens *are* still valid and can be replayed by attackers. Servers may fix this flaw by caching tokens until they expire and comparing new messages against them to check for duplicates (`sig2knock` [TC04] does something similar) or by using coarse-grained clocks (such as minute counters) and allowing only one token per time period (as in `PortKnockO`). Such work-arounds put constraints on how often valid users may authenticate, which may be problematic under some circumstances. These approaches may also require that server state be persistent across restarts, if it is possible for a token to be used and still be valid after a server restart.

- **Sequential password updates** [MvOV96].

  This type of system requires a secure channel for the server to send the next key back to the client after successful authentication. Since port knocking typically provides authentication only, no such channel necessarily exists, making this sort of system difficult to implement. Cappella and Tan designed a hybrid port knocking system using this type of scheme [CK04].

- **Fixed passwords lists and streams derived from master secrets** [MvOV96].

  These schemes require clients and servers to keep track of indices into their password streams. This prevents streams from being shared between users, unless users have some out-of-band mechanism for sharing indices. Also, server crashes could result in clients and servers disagreeing on the current index, preventing further successful authentication.

One solution to this problem is to use the current time as an index; *Spread-Spectrum TCP (SSTCP)* [BHI+02] takes this approach. In this system, clients generate time-dependent pseudorandom streams, and send N elements from the stream starting from the current time. Servers generate the same streams and compare them against what they receive; if $M < N$ elements within a time window match, then authentication is deemed successful. This scheme has the advantage of being able to handle a limited number of dropped or re-ordered packets, but suffers from the weaknesses of time-dependent authentication protocols.

- **S/Key-based schemes** [Hal94].

  S/Key is based on a password stream derived from a master secret, in which servers prompt clients with the current indices. This requires servers to send information back to clients; depending on how this is done, it could compromise the server's stealth (see Section 4.1 for suggestions on designing challenge-response port knocking systems). An S/Key-based port knocking system is implemented in `CÖK` [Wor04].

3.2.2 Port Knocking System Designs

As previously mentioned, a key advantage of port knocking over many other IP covert channels is that clients can be implemented as unprivileged user-space applications that encode authentication information as port numbers and attempt to connect to each port in turn. The design of port knocking servers is more complicated, and there are many variations in use.

Perhaps the most obvious design element of a port knocking server is the method that it uses to read packet headers. Categorized this way, there are five main types of port knocking systems:

- **Firewall log scrapers**

  The simplest port knocking servers usually try to monitor firewall logs. Any firewall software that logs anything will certainly log source IP addresses and destination ports, so all the required information is available.

  Doyle's system [Doy04] is typical of this approach: it periodically hashes firewall log files to determine if they contain any new information and, if so, extracts and parses it. As might be expected, this approach is extremely inefficient. Hashing log files is a slow and computationally-intensive process; firewall logs are often on the order of tens of megabytes in size. If changes are detected, then log files must be further analyzed to extract the new records, which must then be parsed to obtain the required port numbers. Taking into account the time required for the packet records to be written to the log file in the first place and the time until the log scraper next runs, the delay introduced by this procedure is significant. This can be mitigated by reducing the time between log hash computations, but this quickly becomes very computationally expensive, especially since the log scraper must run even when no port knocking is in progress. Log files are also a very noisy communication channel; the firewall may log many packets other than the ones useful to the port knocking server, which must detect and discard the others. Furthermore, log files may be rotated while a knock sequence is in progress; correctly detecting and handling this adds additional complexity to the log reader.

An improvement on this design is to read log messages directly through a named pipe, as is done by `fwknop` [Ras04]. The periodic hashing and file processing are thus eliminated, although text parsing and unwanted message detection are still required.

Log scraping should not be used in conjunction with plain-text authentication, since this type of system necessarily requires that port numbers be stored in log files [Nar04]. If an attacker is able to obtain such logs, then it could determine the secret sequence without needing to sniff network traffic.

- **Packet sniffers**

  A better way for port knocking servers to receive packet headers is to use packet sniffing software, such as `libpcap` [JLM06] on most Unix-based systems or `WinPCAP` [Deg06] on Microsoft Windows. Such software delivers packet headers (and optionally packet contents) to applications in real-time, eliminating the need to scrape logs and parse text. Most packet sniffing software can also be configured to only deliver packets destined to certain ports, reducing the potential for irrelevant noise.

  Krzywinski's system [Krz06] can use this method, as can newer versions of `fwknop` [Ras06].

- **User-space firewall hooks**

  An alternative to packet sniffers is to use platform-specific firewall hooks to pass specific packets directly to user-space programs. Since only the requested packets need to be passed to user-space, this method has slightly better performance than packet sniffing. Some firewalls even allow user-space hooks to decide whether to pass or drop packets; these have the important advantage of not requiring that firewall rules be modified directly to pass packets from clients that have successfully authenticated, since the user-space hook can pass them itself.

On Linux, such a system can be implemented using `libnetfilter_queue` [Wel06b] or its older cousin, `libipq`. No port knocking systems using this method are known to exist, although one has been proposed [ALHF06].

- **Kernel drivers**

  Another method is to build port knocking servers directly into the firewalling subsystems of operating system kernels. This eliminates the issue of passing packets to user-space and the associated processing time. It also allows packets to be passed after successful authentication without directly modifying firewall tables. However, such a system is considerably harder to implement than an equivalent user-space system, due to the difficulty of developing kernel-mode software. Kernel mode also makes it relatively difficult to load configuration or save state. On modern systems, the performance gains possible from this design are slight and not necessarily worth the trouble.

  Nevertheless, kernel-mode port knocking systems have been developed, including `SSTCP` [BHI$^+$02] and `PortKnockO` [SF06].

- **UDP listeners**

  A final method of implementing a port knocking system is to use user-space programs that listen on open UDP ports. Since valid port sequences are not known *a priori* when using cryptographic messages, such servers would be required to listen on large numbers of ports simultaneously; it is generally easier to use log parsers or packet sniffers. This design is feasible for systems employing plaintext authentication messages: servers need only listen on ports on which they expect to receive packets. However, care must be taken to avoid two types of attacks:

1. If the server only listens on one port at a time (the one that it expects next), then a simple port scan covering that port will cause the server to open the next port in the list. At that point, another port scan can be conducted; after at most n scans, any sequence of n ports will have been discovered and the server will open the protected service. The attacker never finds out what the secret sequence is, but the time complexity of a brute force attack against such a system is linear, rather than exponential.
2. If the server listens on more than one port at a time and is running on a system that sends ICMP messages in response to connections to closed ports, then a single port scan will expose both the existence of the server and the set of ports used. Brute force attacks are still exponential, but the server is no longer active-covert.

One attempt at a port knocking system of this type is `jPortKnock` [Gre05], which unfortunately is vulnerable to both types of attacks.

As previously mentioned, it is not always necessary to add rules to firewall configurations in order to open ports; some port knocking servers are capable of passing packets on their own. Although this approach may be more complicated, it is often more efficient and less error-prone. Avoiding adding rules also has the advantage of not requiring that any rules be removed later.

Another major issue facing the design of port knocking servers in how to close open ports when they are no longer needed. Approaches to this issue include the following:

- **Separate close knocks**

  One approach to closing ports is to use separate "close" knock sequences. This way, a client can open a port, leave it open for as long as it likes, and close it again when ready. This approach also allows clients to open more than one connection per authentication exchange, which is useful if one wants to open many connections in a short period of time without incurring the overhead of authentication each time, but this is also a disadvantage: attackers masquerading as the client can also open connections without authenticating. Other disadvantages include the increased overhead of sending "close" knocks and the possibility of clients forgetting to send them at all. Despite these issues, many existing systems use close knocks, including Doyle's [Doy04] and Krzywinski's [Krz06].

- **Timeouts**

  An alternative method is to close open ports after a timer expires [Krz03]. Such a timer could start when the port is opened, or when the last traffic was seen on that port. This approach has the advantage of not requiring users to send "close" knocks, but has the potential to cut off authenticated clients' sessions. It still has the property of allowing more than one connection to the port.

- **Firewall connection tracking**

  A final method is to not attempt to track connections at all, but rather to allow only the first packet of a connection through the firewall and rely on the firewall's connection tracking system to do the rest. Of course, this assumes that the firewalling software in use is capable of tracking connections; stateless packet filters like `ipchains` [Rus00] cannot. If an actual rule is added to the firewall to allow this first packet to pass, then it can be removed after a timeout (as with `fwknop` [Ras06] or after some monitoring system has detected a connection establishment; if no rule was added to the firewall (as in `PortKnockO` [SF06]), then the server needs to update its state after passing the first packet.

Port knocking can use TCP or UDP ports. Generally, UDP is preferred, since TCP headers are larger than UDP headers and TCP SYN segments are likely to cause unnecessary resource allocations in stateful firewalls and NATs.

Another important design issue is the size of the port range that is used. TCP and UDP port numbers are 16 bits wide, putting an upper bound of 16 bits on the amount of data that can be transmitted per packet, and the server must monitor the full range of 65,536 ports. (By using *both* TCP and UDP ports, a maximum of 17 bits can be sent per packet.) However, it may be necessary to limit the port range used to a smaller set, thus limiting the data sent per packet, in order to avoid monitoring ports used for other purposes, such as outgoing connections or active services. Many port knocking systems are designed to operate over 256 ports, allowing 8 bits of data per packet; finding an unused range of this size should not be difficult on most systems. It is also possible to use disjoint port ranges.

Some port knocking systems allow clients to request that any port be opened; others limit requests to specific pre-configured actions. Allowing arbitrary actions makes server configuration simpler, but introduces a few problems. Allowing arbitrary ports to be opened puts constraints on the format of authentication messages: they must be able to carry a port number that can be decoded by the server. Also, the lack of restriction on possible actions means that an attacker who has obtained an authentication key can open *any* port, rather than being limited to a small set of possible actions.

It is possible to use a single, globally shared secret for authentication, or various per-user or per-port secrets. Global secrets are especially useful with servers that allow arbitrary requests; the number of separate shared secrets required by such a server for per-user or per-port authentication would not be practical. However, it is difficult to audit who did what and when while using global secrets, and there are administrative difficulties in changing keys that are shared by multiple users; per-user or per-port secrets (or a combination thereof) allow finer-grained control at the expense of increased configuration complexity.

One final design issue of port knocking systems is what port they *actually* open. Most servers open the requested port directly. Cappella and Tan's system [CK04] takes a different approach by opening a random port, forwarding the requested service to that port, and sending the random port number in an encrypted message back to the client; the client can then decrypt this message and connect to the random port. There is extra overhead involved in this system, but it does introduce an association between the authentication exchange and the subsequent connection, the lack of which is a major problem in most port knocking systems (see Section 3.2.3 for details).

3.2.3 Weaknesses of Port Knocking

As presented, port knocking has a number of drawbacks:

- **Efficiency**

  A minimal UDP datagram is 224 bits long at the network layer [Pos80]; a minimal TCP segment is 320 bits [Pos81a]. Given that port knock sequences typically contain 64 to 160 bits of information and that only 8 to 16 bits can be transmitted in a single packet, port knocking is an extremely inefficient method of transmitting data, requiring 896 to 6400 bits to be sent over 4 to 20 packets. The same amount of information could be sent in a single packet of only 288 to 480 bits.

However, the packets required for port knocking can still be sent in well under a second (see Chapter 4), even on relatively slow network links, so efficiency is not necessarily a limiting factor.

- **Vulnerability to packet loss and out-of-order delivery**

  In most implementations, proper decoding of port-knock sequences is dependent on the order of arrival. According to a simple experiment presented in Appendix A, the probability of small packets sent in rapid bursts across busy routed networks being delivered out of order, as is common for port knocking systems, may be as high as 80%. Decoding will also fail if any packets are lost; under the same circumstances, up to 30% packet loss may occur. Port knocking, as presented in this chapter, provides no mechanisms to allow servers to re-order packets on delivery or detect lost packets; under these conditions, port knocking systems that do not compensate for out-of-order delivery will almost always fail.

  Not all port knocking systems suffer from these problems: `SSTCP` [BHI+02] is capable of handling a limited number of out-of-order or dropped packets.

- **Failure in the presence of network address translation**

  Port knocking systems (and any other IP address-based authentication systems) that encode the client's local IP address into authentication messages will fail if authentication messages pass through NATs; successful authentication will result in servers opening ports to clients' private addresses, which are likely not even routable and certainly not correct. A number of existing systems suffer from this problem, including Doyle's [Doy04] and Krzywinski's [Krz06].

  An attacker that controlled a host with the same *public* IP address as a legitimate user's *private address* could take advantage of this flaw. If the legitimate user attempts to authenticate to a port knocking server that relies on the user to specify its own IP address, then the server would grant access to the attacker. However, due to the improbability of this arrangement, this is not a likely attack scenario.

- **Lack of association between authentication and connection**

  In most existing port knocking systems, there is no logical association between the authentication sequence and the connection that is subsequently opened. This means that after a successful authentication, anyone with the client's IP address can connect to the server (hereafter referred to as a *race attack*). An attacker could hijack a successful authentication by blocking further transmissions from a client after it authenticates, but before it makes a connection, and then assuming the client's identity and connecting itself. This problem is especially severe in the presence of NAT; to a server that has obtained the public IP address of a client, all hosts that share the client's public address look alike. An attacker that shares the client's public address does not need to block transmissions from the client; it just needs to connect to the server first.

  Cappella and Tan [CK04] presented a partial solution to this problem by opening a random port and sending the port number to the client in an encrypted message. This makes it impossible for an attacker to predict what port is to be opened before the client makes a connection, but does not prevent the attacker from locating the random port by scanning or blocking the client's connection and substituting its own.

- **Susceptibility to denial-of-service attacks**

  There are several possible denial-of-service attacks against port knocking servers. Maddock [Mad04] pointed out that an attacker could prevent a client from authenticating by sending packets with the client's source address to random ports on the server while the client is trying to authenticate; if any of these packets went to ports being monitored by the port knocking server, then the client's sequence would be corrupted and authentication would fail. Manzanares et al. [MMETC05] suggested that an attacker could effect a resource-consumption attack against a known port knocking server by sending packets with random forged source addresses to random ports. The server would attempt to process port knocking sequences from all of the sources that it sees, possibly consuming large amounts of memory and CPU time. If the server drops packets when it gets overloaded, then this could also prevent valid clients from authenticating. A port scan from a single source at a sufficiently high rate may also be sufficient to overload a server's processing resources, particularly if the server uses a computationally-intensive cryptographic protocol.

- **Failure in the presence of egress filters**

  Many networks filter outbound network traffic, often only allowing traffic destined to a small set of well-known ports to pass. Such firewalls may drop traffic destined to some or all of the various seemingly-random TCP or UDP ports used by port knocking servers; in such an environment, port knocking will not work.

Improvements to port knocking that address many of these weaknesses will be introduced in Chapter 4.

3.2.4 Uses of Port Knocking in Malware

Like any technology, port knocking can be used for evil. Use of port knocking is not limited to exchanging authentication information and manipulating firewall configurations: it can be used as a transport protocol for any sort of data. Since port knocking is easily mistaken for port scans or other innocuous traffic, it can be used to pass data beneath the radar of intrusion detection systems (IDSs): for instance, "hackers" could use port knocking to conceal back doors installed on compromised systems or pass commands to software on zombie computers [Kri04], and spyware could use port knocking to send captured information to a controller [Gee05]. At least two existing port knocking implementations, `cd00r` [FX 00] and `TocToc` [0ld02], were designed specifically for use as back doors. According to various reports on the Internet [Bra, Mul05, Sym04] and elsewhere [SZ04], port knocking may already be in wide-spread use by malware.

Even if an IDS does detect such a use of port knocking, the proper response to a port scan is likely to be inappropriate to the real situation; IDSs usually assume that a port scan is a precursor to an attack [JPBB04] and that no action beyond blocking the scan is needed, whereas port knocking used in this manner implies that an attack has already succeeded and that cleanup is necessary. Since the uses suggested above generally require little data to be transmitted and are probably not time-sensitive, malware using port knocking in this manner could take advantage of many port scan detector evasion techniques [JPBB04, Sol98] to further discourage discovery.

Fortunately, blocking port knocking is not difficult. Since port knocking requires that data be transmitted to relatively large numbers of ports on target systems, firewalls that prevent delivery of these packets will effectively block this means of covert communication. Port scan detection tools [JPBB04, LK02, Sol98, SHM02] should also be effective. Firewalls on the hosts targeted by port knocking traffic will not necessarily be effective, since most of the channels used by port knocking servers to receive packets (see Section 3.2.2) do not require that packets be delivered to applications in the usual manner; however, network firewalls will be effective. Likewise, since port knocking malware may subvert firewalling software [Kri04] or send packets without going through the standard network stack, host firewalls may not be effective at blocking locally-generated port knocking traffic, but network egress firewalls will effectively disrupt port knocking systems.

3.2.5 Distinguishing Port Knocking from Port Scans

One of the advantages (and disadvantages) of port knocking listed above is that, to a third-party observer, it strongly resembles a port scan or DDoS backscatter. However, there are methods that can be used to distinguish between the two.

- Sequences of probes to multiple ports that result in an observable change in the behaviour of the target host may indicate port knocking [MMETC05]. This is not a guaranteed indicator: port scans may be followed by connection attempts to ports discovered to be open, and hosts being scanned may respond by closing ports to the scanning host, performing DNS lookups on the scanning host, or even scanning the scanner in return, depending on what or who is listening on the host being scanned. Most of the time, a port scan will not result in any remotely-observable changes of the target's behaviour at all. However, port scans are unlikely to immediately follow the closure of connections to the target, cause the target to begin listening on new ports, or cause the target to send other types of data. Actions such as these strongly suggest that the port sequence was a port knocking exchange, rather than a simple port scan.

- The characteristics of a suspected port knock sequence can be compared against common port scanning software and scan types. The popular `Nmap` port scanner typically precedes port scans with a ping (ICMP *echo-request*) and a TCP ACK to port 80, probes a set of about 1600 low-numbered ports in random order using random high-numbered source ports, and will retry any ports that do not respond at all [Fyo06]. This particular behaviour would be atypical of a port knocker, whereas a single pass across a small number of randomly-chosen, monotonically-increasing, high-numbered ports would be highly improbable for a port scan. However, `Nmap`'s operation is highly configurable, other port scanners will have different signatures, and port knockers can be deliberately designed and configured to mimic the behaviour of port scanners, so this method is not guaranteed to be accurate either.

Designing a technique to differentiate between port scans and port knocking in the general case is a topic for further research and may not even be possible.

3.3 Single Packet Authorization

Single Packet Authorization, or SPA, has the same goals as port knocking, but, instead of encoding authentication information in a series of port numbers, it encodes it in the payload of a single UDP datagram (see Figure 3.2). This allows for authentication messages of several kilobytes to be used without concern for packet reordering.

The information encoded in an SPA message is generally similar to what might be encoded in a port knocking sequence (see Section 3.2.1). A message containing a plain-text secret could be used, although most existing implementations use some sort of encrypted or one-time message. Unfortunately, the phenomenon of misused cryptography and broken authentication protocols is not limited to port knocking. *Cryptknock* [Wal04] (actually an SPA system, despite its name) uses an unauthenticated Diffie-Hellman exchange to generate a session key which is then used to encrypt a shared secret; if the server accepts the secret, then it allows unrestricted access to the originating host. Since this protocol doesn't associate the shared secret with the

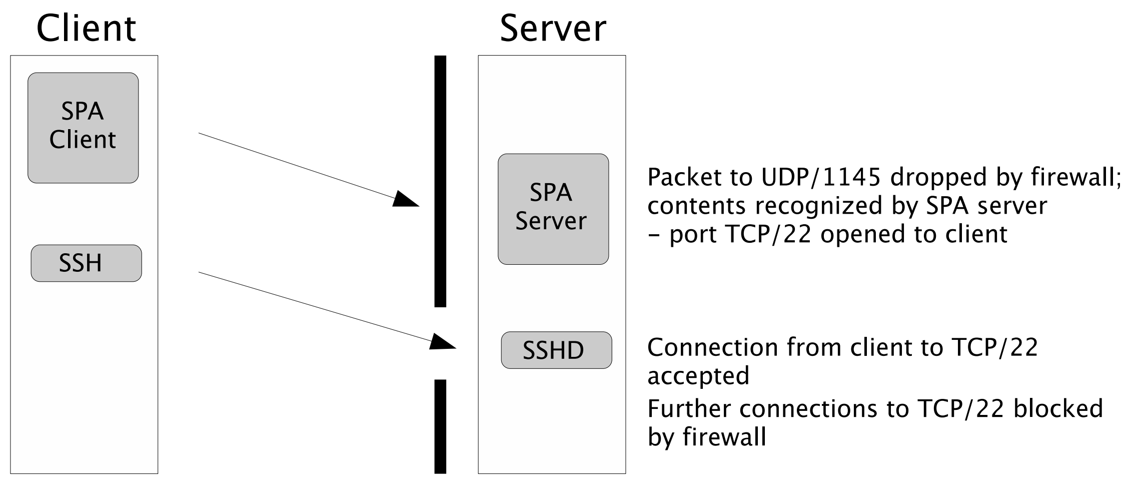

Client
Server
SPA Client
SSH
SPA Server
SSHD
Packet to UDP/1145 dropped by firewall; contents recognized by SPA server – port TCP/22 opened to client
Connection from client to TCP/22 accepted
Further connections to TCP/22 blocked by firewall

Figure 3.2: SPA example. A port is opened in the firewall in response to an authentication packet.

client's IP address, an attacker could break it by re-writing the client's IP headers to make it appear to the server that they originated at the attacker, then forwarding the server's responses back to the client. Alternately, the standard man-in-the-middle attack against Diffie-Hellman [Sta03] would allow an attacker to recover the shared secret. `Tailgate TCP` (`TGTCP`) [BHI+02], `Doorman` [War05], `tumbler` [GC04], and `fwknop` [Ras06] implement SPA with more robust authentication schemes.

SPA servers that use packet sniffers or related technologies may need to limit packet sizes to the path MTU between client and server (typically a minimum of 576 bytes [MD90]) in order to avoid fragmented packets. However, this is still ample room for authentication information in messages of this size.

3.3.1 Advantages of SPA

As Rash [Ras06] points out, SPA is easier to implement and less failure-prone than port knocking and is probably preferable in most circumstances.

1. An SPA server can be written as a normal network service on an open port. Since UDP services are not required to respond to messages that they receive, and the protocol does not automatically generate any response, a non-responding UDP service on an open port on a system that silently drops unexpected packets is indistinguishable from a closed port to a port scan. An SPA server can therefore be written as a normal network service, without needing to resort to packet sniffers or any platform-specific mechanisms. (However, such a design may put constraints on the available mechanisms to manipulate firewall states.)

2. Since only one packet must be sent before opening a connection, an SPA authentication exchange takes much less time and is less vulnerable to packet loss than port knocking, besides being immune to packet reordering.

3. NATs and stateful firewalls between SPA clients and servers will only have to allocate resources for at most one logical connection, rather than one for each knock as required with port knocking.

4. Compared to port knocking, SPA can use relatively large authentication messages without sacrificing performance and reliability.

3.3.2 Disadvantages of SPA

Despite being much less sensitive to packet ordering than port knocking, SPA systems will still fail if a connection attempt reaches the firewall before the authentication packet has been received and processed. They will also fail if an authentication packet, or a fragment of one, is dropped or corrupted in transit.

SPA servers typically cannot be implemented as log readers, since SPA systems need to access packet payloads and firewalls generally don't log any more than packet headers. However, this is not usually a problem, since log reading is an inefficient design compared to its alternatives (see Section 3.2.2) and is seldom used for SPA.

Egress filters may not pass outbound traffic destined to unusual UDP ports, but SPA servers could run on ports regularly used for “normal” traffic. For instance, most egress filters will permit DNS traffic; SPA messages bound to a server running on port 53/UDP would likely pass unmolested.

3.3.3 Variations on SPA

It is possible to encode SPA messages into the payloads of any protocol. In [Ras06], Rash suggests using the payloads of ICMP or GRE messages. In theory, raw IP messages with no transport headers at all could also be used. Although such systems have the potential to be extraordinarily stealthy (ICMP *echo-request*, (“ping”) messages are very common in the Internet background radiation [PYB+04], although GRE and raw IP messages are rather unusual), they do present some implementation challenges. User applications cannot usually read ICMP, GRE, or raw IP payloads, requiring that servers using such encodings hook into network stacks at a lower level (for example, using packet sniffers). More importantly, unprivileged user programs cannot directly send such messages, thus requiring clients to be privileged applications.[3] Currently, `fwknop` [Ras06] and `Cerberus` [Epp04] implement SPA over ICMP.

Barham et al. [BHI+02] suggested a variation on *TGTCP* in which the authentication information would be attached as a payload to the SYN segment opening a TCP connection; this would prevent race attacks and be invulnerable to out-of-order delivery. However, this approach is not without weaknesses: it only works for TCP

[3]ICMP-based SPA systems may be able to use the `ping` command as a client. Although standard on most operating systems, `ping` is a privileged application with the ability to attach payloads to ICMP messages. The Linux and OpenBSD versions allow users to specify up to 16 bytes of payload data, although the Windows XP version does not have this capability.

ports, it requires modifications to the client's network stack to attach authentication information to outgoing TCP SYN segments, it requires a kernel-level server that can process packets before they reach the transport layer of the firewall's network stack, and some egress filters may drop SYN segments carrying data.

3.3.4 Active-covert SPA

Since SPA traffic is visible to packet sniffers and is not disguised as background noise, it cannot normally be considered active-covert. However, it is possible to give active-covert properties to SPA by encoding the payload as something that might normally be seen in background traffic and setting the destination port to match. For instance, authentication information encoded as ASCII text and sent to port 1026/UDP with the proper headers would resemble Windows Messenger spam [LUR03], and a message resembling Intel x86 machine code sent to port 1434/UDP might be mistaken for the `Sapphire` worm [MPS$^+$03].

3.4 Application-layer Covert Channels

It is also possible to encode authentication information as common messages at the application layer, such as DNS or HTTP requests. To send the bytes "153, 187, 89" using a DNS analogue to port knocking, one could look up the hosts "153.somedomain.zz", "187.somedomain.zz", and "89.somedomain.zz" from a DNS server on the firewall. An HTTP equivalent would be to request "153.html", "187.html", then "89.html" from a web server on the firewall. An SPA analogue would be to send the entire message as one request ("153187089.somedomain.zz" or "153187089.html"), although this may not work well with long messages. `CÖK` [Wor04] is one existing system that uses DNS messages to encode authentication information.

Implementing such a system would require that the DNS or HTTP server listen on an open port and hence be visible to port scans. However, to such a scan, it would appear to be an ordinary DNS or HTTP server; it could be configured to serve valid content or just default information but would not give away the information that it was involved in an authentication scheme. Since minimal subsets of both protocols can be implemented very simply, having a minimal DNS or HTTP server exposed to the Internet should present little risk. A major advantage of such a system is that most firewalls, including those that would block normal port knocking or SPA traffic, will allow DNS and HTTP traffic to pass. Optionally, the next connection after a successful authentication could be forwarded to the desired service; this would allow connections to arbitrary ports to be made through firewalls that would otherwise block them.

3.5 Concerns about "Security by Obscurity"

Port knocking systems in particular have often been accused [BBO05, MMETC05, Nar04] of being nothing more than "security by obscurity". Generally, these claims are based on assumptions that the security of port knocking authentication systems is based solely on them remaining hidden, or that concealing security-sensitive information is bad and that all details of security systems should be visible.

Beale [Bea00] describes a system relying on security by obscurity as one that relies on critical knowledge about the system's design being kept secret, though the secret information could be discovered by an outsider without unreasonable effort. The authentication systems used by traditional services such as telnet and FTP, which send passwords in plain text, are generally considered insecure by modern standards [Bel89], but their well-specified protocols and documented reliance on the secrecy only of small easily-changeable per-user passwords leave them well outside of the realm of security by obscurity. This is equally true for similarly-designed services such as SSH, which employ cryptography to protect secrets in transit.

The security of port knocking systems and other covert authentication schemes is also dependent only on the knowledge of small, easily changeable secrets; in the case of port knocking systems, these are port sequences. Systems that send their secrets are equivalent in security to telnet, whereas those that use cryptographic protocols are more akin to SSH. In neither case does the security of the authentication system depend on any other property. The covertness of the communication channels being used is not necessary; if the same information was transmitted across normal, open ports, the system would remain secure. Rather, the covertness only serves to increase the level of effort required to attack the systems. As Beale points out, concealing an already-secure service is not a weakness but rather has a number of advantages, reducing the number of attacks faced by the system and forcing attackers to do more work, which both slows them down and makes their actions more obvious.

Chapter 4

Improvements to Port Knocking and SPA

The 1st Law of Cryptography: Don't design your own crypto.
The 2nd Law of Cryptography: Don't implement your own crypto.
– Rennie deGraaf

As presented in the previous chapter, port knocking and SPA have a number of weaknesses. In this chapter, novel techniques for addressing some of these weaknesses and other improvements to port knocking are introduced. First, I introduce methods for conducting challenge-response authentication using SPA or port knocking, which improve on some of the limitations of the authentication algorithms described in Section 3.2.1 and enable authentication of the server as well as of the client. Next, a variety of strategies for ensuring that port knock sequences can be correctly decoded, even if delivered out of order, are discussed. Third, two alternate methods for encoding information into port sequences are introduced. Finally, several possible ways of associating authentication exchanges with connections and preventing race attacks are described.

4.1 Challenge-response Knocking

Besides the authentication strategies listed in Section 3.2.1, it is also possible for port knocking and SPA systems to authenticate using challenge-response algorithms. Such systems would provide strong authentication without the issues of synchronizing clocks and password indices between clients and servers found in one-time authentication schemes (Section 3.2.1).

A challenge-response system necessarily requires that servers return information to clients before authentication is complete, thus potentially exposing their existence to attackers. However, this loss of stealth can be minimized if clients send fairly long request sequences to trigger the issuance of challenges (with 8 to 10 bytes generally being long enough, as argued in Section 3.2.1), and if unrecognized request sequences go unanswered by servers. Such a request sequence would be comparable to a plain-text authentication token: it would be practically impossible to guess but could be identified by sniffing a legitimate user's authentication exchange. Obviously, an attacker that intercepts a request sequence could replay it to receive a challenge, but it can be assumed that by this point the attacker knows of the existence of the port knocking server, so issuing a challenge results in no loss of stealth. As long as the authentication mechanism is secure, this does not introduce any security vulnerabilities.

In a pure port knocking system using challenge-response authentication, a client would send an initial message by port knocking; if recognized, the server would issue a challenge in a single UDP packet, followed by a response from the client, again sent by port knocking. The challenge could theoretically also be sent by port knocking, but if the client is assumed to be an unprivileged application, then it may not be able to efficiently receive data by port knocking, and it may be behind a firewall that would prevent it from receiving such data at all. However, stateful firewalls generally *do* allow responses to UDP messages to pass [And06], so a response sent to the source port of the port knocking messages should be allowed to pass. Since active attackers cannot be expected to get to the challenge state and further hiding from passive attackers that have detected request and challenge messages is of limited value, an alternate design, *hybrid port knocking*, can be used. In such a system, both the challenge and response packets would be sent in single UDP packets; this could significantly speed up the authentication process. In an SPA system employing challenge-response authentication, all three messages would be sent in separate UDP packets.

Note that either form of challenge-response port knocking compromises the passive-covertness of the port knocking channel, unless special care is taken to ensure that the challenge (and response, in the case of hybrid port knocking) resembles something that would legitimately have been sent in response to the apparent port scan of the request.

4.1.1 Basic Unilateral Authentication

The ISO two-pass unilateral authentication [ISO95, MvOV96] is designed to authenticate a client A to a server B; no attempt to authenticate the server to the client is made. A slightly modified version of this algorithm, intended to be suitable for port knocking or SPA, is shown in Algorithm 4.1. In the following discussion, I refer to message 1 as the *request*, message 2 as the *challenge*, and message 3 as the *response*.

Algorithm 4.1 Challenge-response unilateral authentication

```
1: $A \rightarrow B$: $req$
2: $B \rightarrow A$: $N_B$
3: $A \rightarrow B$: $MAC_{K_{req}}(N_B, ID_A, ID_B, req)$

where $A$ is the client
      $B$ is the server
      $req$ is a request for authentication
      $N_B$ is a nonce chosen by $B$
      $K_{req}$ is a secret key shared by $A$ and $B$
      $ID_X$ is the IP address of $X$
      $MAC$ is a cryptographic message authentication function
      , (a comma) represents concatenation
```

A begins the sequence by sending a request, which serves both to initialize the protocol and identify the operation to be performed upon successful authentication. Cryptographically, this must be considered public information. B responds to a recognized request by issuing a unique nonce as a challenge, to which A responds with a MAC covering the nonce, the IP addresses of A and B, and the request sequence, keyed with a symmetric key associated with the request. Upon receipt of the response, B will re-compute the MAC using the request that it received, the nonce that it sent, A's IP address as taken from the packet headers, and B's own IP address. If the MAC is valid, then B will perform the requested operation; otherwise, no action will be taken.

If HMAC-SHA1 is used as the MAC algorithm, then response messages will be 160 bits long. Due to birthday attacks [MvOV96], nonces used in this situation should be at least half the bit length of the MAC, or 80 bits in this case. As argued above, a hard-to-guess sequence of 8 to 10 bytes will serve as a request sequence.

This protocol is suitable for port knocking or SPA systems employing either pre-configured or user-issued commands. Servers can identify pre-configured commands by associating unique request sequences with each command. User-issued commands could be constructed by appending port numbers and other information to request sequences; if the command must be kept secret, it can be encrypted using K_{req}. No integrity-checking information is needed in such request messages, because they are covered by the MAC in the response message, and none of the information in the command will be acted upon until after successful authentication. Pre-configured commands are best for port knocking systems, due to the relatively high overhead of sending data, but there is little penalty for attaching additional data to SPA requests.

A's IP address has been added to the MAC in Step 3 of the original algorithm to prevent possible *Mafia fraud*-based attacks [DGB87] (see Figure 4.1) in which an attacker C initiates the protocol and receives a challenge, intercepts (and blocks) a challenge issued to A in another protocol session, and forwards its own challenge to A in order to get A to generate a valid response for C (see Algorithm 4.3 for an example). By covering both IDs with the MAC, Algorithm 4.1 prevents C from subverting the protocol to authenticate to B as itself, but does not prevent C from subverting the protocol by masquerading as A. Such an attack still requires A to initiate an authentication exchange itself before it will generate a response, and would have the effect of causing B to perform the action specified by C's request. If A's and C's requests are the same, then there is no security breach: B does exactly what A asked, under A's credentials, and nothing more. If the request caused a port to be opened, then C could attempt to connect to it while continuing to masquerade as A, but this is then equivalent to C ignoring the authentication exchange and attempting to hijack a successful authentication, as in Section 3.2.3. If A's and C's requests are different, then authentication will fail, since A also covered its own request in the MAC and B will expect C's request when verifying it.

Instead of adding A's address to the MAC in the response, a MAC covering ID_A and N_B could have been added to the challenge message; this would have equivalent security properties but increase the amount of data to be transmitted. This is the approach suggested by van Oorschot and Stubblebine [vOS06] for preventing Mafia fraud-based attacks.

The original version of Algorithm 4.1, presented in [dAJ05], did not cover the request in the MAC and therefore depended on the keys associated with the two requests being different in order to resist Mafia frauds.

4.1.2 Authentication in the Presence of NATs

One flaw in Algorithm 4.1 is that it requires the client, A, to know its identity as seen by the server, B. Unfortunately, if the client is behind a NAT, then it may not know its public address and may not even know that the NAT exists. (Since the server is intending to receive connections, it is assumed to have have a valid public address.)

In the above protocol, A will use its private address PID_A to compute the response, but if A's address is re-written on the packets that it sends, then B will use A's public address ID_A to verify it and authentication will fail.

Algorithm 4.2 NAT-aware unilateral authentication

```
1: $A \rightarrow B$: $req, PID_A$
2: $B \rightarrow A$: $N_B, ID_A, MAC_{K_{req}}(PID_A, ID_A)$
3: $A \rightarrow B$: $MAC_{K_{req}}(N_B, ID_A, ID_B)$

where $A$ is the client (prover)
      $B$ is the server (verifier)
      $req$ is a request for authentication
      $PID_X$ is the private IP address of host $X$
      $ID_X$ is the public IP address of host $X$
      $N_B$ is a nonce chosen by $B$
      $K_{req}$ is a secret key shared by $A$ and $B$
      $MAC$ is a cryptographic message authentication
        function
      , (a comma) represents concatenation
```

In an earlier work [dAJ05], I presented an algorithm called "NAT-aware unilateral authentication" (Algorithm 4.2) which I claimed would authenticate a client A that doesn't know its public IP address to a server B while resisting Mafia frauds. Unfortunately, I have since discovered an attack against this algorithm, shown in Algorithm 4.3.

Algorithm 4.3 Attack against NAT-aware unilateral authentication

```
1: $A \longrightarrow B$:     $req, PID_A$
2: $C \longrightarrow B$:     $req, PID_A$
3: $B \rightarrow /A$:        $N_B, ID_A, MAC_{K_{req}}(PID_A, ID_A)$
4: $B \longrightarrow C$:     $N'_B, ID_C, MAC_{K_{req}}(PID_A, ID_C)$
5: $C(B) \rightarrow A$:      $N'_B, ID_C, MAC_{K_{req}}(PID_A, ID_C)$
6: $A \longrightarrow B$:     $MAC_{K_{req}}(N'_B, ID_C, ID_B)$
7: $C \longrightarrow B$:     $MAC_{K_{req}}(N'_B, ID_C, ID_B)$
```

In this attack, an attacker C waits for a legitimate client A to initiate the protocol and then opens its own protocol session by sending a copy of A's request. C then blocks the delivery of B's challenge to A and substitutes its own challenge. (The notation $B \rightarrow /A$ means that B sends a message to A, but it is not delivered.) A accepts B's assertion that its public IP address is ID_C and generates a response, which B rejects since it knows A's correct public IP address. However, this response is valid for C, which is then able to complete the protocol successfully without knowing K_{req}. This attack also works if C blocks the request and response sent by A or masquerades as B to A.

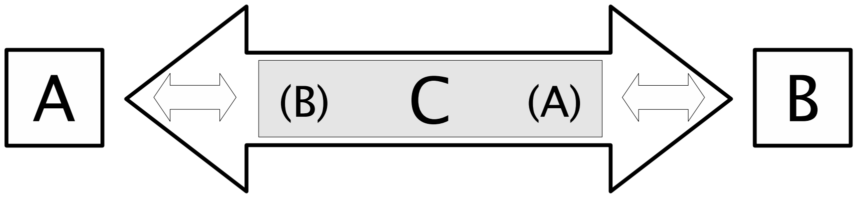


Figure 4.1: The Mafia fraud. A and B think that that they are authenticating to each other, but C is forwarding messages between them with the goal of convincing A that it is B and B that it is A.

Since this attack is exactly what Algorithm 4.1 was designed to prevent, there is no point in ever using Algorithm 4.2. However, the Mafia fraud (a man-in-the-middle attack related to the *grandmaster postal-chess problem* [MvOV96, BD90] and known as the *MiG-in-the-middle* attack in military contexts [And01], see Figure 4.1), which works because the verifier, B, has no knowledge of the physical location of the prover, A, is generally considered difficult to prevent and is frequently ignored by authentication systems [AS02]. None of the general solutions to this problem available are appropriate under these circumstances because A cannot practically be isolated during authentication [BBD+91], the only communication channel available can be accurately monitored by attackers [AS02] and doesn't have a constant communication delay [BD90], and A's physical description (here, A's IP address is sufficient – see Section 4.1.1) cannot be included in the exchange [Des88] because A *doesn't know* its

IP address.

The lesson to learn here is that it is difficult to prevent Mafia frauds when the client does not know its identity (in this case, its public IP address). Even if it does, any other computer sharing the same public IP address (i.e., any computer behind the same NAT) can still carry out a Mafia fraud. The Mafia fraud is generally irrelevant in protocols that combine authentication with key agreement, because, even if an attacker succeeds in authenticating as some other entity, it will still not know the agreed-upon key [DvOW92]. Some of the solutions to the race attack problem presented in Section 4.4 are based on key agreement techniques, so they may mitigate this problem. Another partial solution is for clients to request their public IP addresses from trusted third party identity oracles before starting the authentication protocol.

In addition to this attack, since *req* is not covered in the MAC in the response, C could substitute A's request with another one employing the same key to cause B to execute a command other than the one that A intended. This is easily corrected by adding *req* to the MAC, as in Algorithm 4.1.

4.1.3 Mutual Authentication

Since Algorithm 4.1 provides only unilateral authentication, it provides no authentication of the server to the client. To prevent attackers from masquerading as legitimate servers to clients, the authentication algorithm must be changed to support mutual authentication. Algorithm 4.4 is a variation on the SKID3 protocol [Res95], modified for use with port knocking or SPA.

This algorithm extends Algorithm 4.1 by adding a nonce generated by A to the request message and a MAC on that nonce to the challenge message, which allows B to prove to A that it, too, possesses the secret key K_{req}. Adding the request to the MAC in the challenge allows the client to ensure that the correct request was received,

Algorithm 4.4 Challenge-response mutual authentication

```
1: $A \rightarrow B$: $req, N_A$
2: $B \rightarrow A$: $N_B, MAC_{K_{req}}(N_B, N_A, ID_A, req)$
3: $A \rightarrow B$: $MAC_{K_{req}}(N_A, N_B, ID_B, req)$

where $A$ is the client
      $B$ is the server
      $req$ is a request for authentication
      $N_X$ is a nonce chosen by $X$
      $K_{req}$ is a secret key shared by $A$ and $B$
      $ID_X$ is the IP address of $X$
      $MAC$ is a cryptographic message authentication function
      , (a comma) represents concatenation
```

in the case that more than one request uses a particular key. Assuming that A knows its public IP address, this algorithm is resistant to Mafia frauds since the MAC in the challenge covering A's IP address allows A to ensure that N_B was, in fact, intended for it.

In an earlier work [dAJ05], I presented an algorithm based on Algorithm 4.2 which I claimed could prevent Mafia frauds for clients which do not know their public IP addresses; this algorithm is vulnerable to an attack similar to Algorithm 4.3 and therefore provides no better security than the basic SKID3 protocol. Jeanquier [Jea06] gave a similar algorithm that resists modification to requests through the addition of a MAC on req and NB to the request message, but is similarly vulnerable to Mafia frauds.

4.1.4 Analysis of Challenge-response Authentication

Due to the increased message complexity of challenge-response protocols, they require significantly longer execution times than unilateral authentication schemes. This demands an analysis of the execution times required for these types of protocols. For the purposes of these calculations, it is assumed that request sequences and nonces are 10 bytes each, IDs are 4-byte IPv4 addresses, MACs are 20 bytes, and pure port knocking is used. Including all additional protocol data, Algorithm 4.1 (unilateral authentication) will have requests of 80 bits, challenges of 80 bits and responses of 160 bits; Algorithm 4.4 will use requests of 160 bits, challenges of 240 bits and responses of 160 bits. A "fast" network (similar to residential broadband) is assumed to have a bandwidth of 1 Mbps and a round-trip time (RTT) of 50 ms, and a "slow" network (similar to residential dial-up) has a bandwidth of 48 Kbps and a RTT of 200 ms; no packet loss is assumed in either case. All processing time is disregarded.

Using this model, the execution time (the time that passes between a client starting to send a request and a server receiving the entire response) for a time-based authentication protocol employing a single 160-bit message can be estimated by the following formula:

$$time = \frac{\lceil \frac{160}{d} \rceil * 244}{kbps} + \frac{rtt}{2} + \left(\left\lceil \frac{160}{d} \right\rceil - 1 \right) * ipd$$

and the execution time for challenge-response port knocking can be estimated by

$$time = \frac{\left(\lceil \frac{req}{d} \rceil * 244\right) + (cha + 244) + \left(\lceil \frac{res}{d} \rceil * 244\right)}{kbps} + \frac{3 * rtt}{2} + \left(\left\lceil \frac{req}{d} \right\rceil + \left\lceil \frac{res}{d} \right\rceil - 2 \right) * ipd$$

where d is the number of data bits sent per packet, $kbps$ and rtt are the network bit rate and round-trip time, ipd is the delay between sending packets, the size of a UDP/IP header is 244 bits, and req, cha and res are the number of bits in request, challenge, and response messages.

As seen in Table 4.1, execution times for challenge-response port knocking protocols are not unreasonable, even on slow networks. Unilateral challenge-response authentication takes little more than twice the time required for a time- or one-time-password-based authentication scheme using a single 160-bit message; mutual authentication takes only slightly longer than unilateral. For all but the smallest port ranges (< 64 ports) and the longest delays, protocol execution time is dominated by the RTT,

Data bits per packet	Ports required	Execution time (seconds)					
		Slow network			Fast network		
		Time-based	Alg. 4.1	Alg. 4.4	Time-based	Alg. 4.1	Alg. 4.4
1	2	0.846	1.426	1.803	0.060	0.129	0.147
2	4	0.473	0.866	1.056	0.042	0.102	0.111
3	8	0.352	0.684	0.813	0.037	0.093	0.099
4	16	0.286	0.586	0.686	0.033	0.088	0.093
5	32	0.249	0.530	0.608	0.032	0.086	0.089
6	64	0.226	0.497	0.561	0.031	0.084	0.087
7	128	0.207	0.469	0.524	0.030	0.083	0.085
8	256	0.193	0.446	0.496	0.029	0.082	0.084
9	512	0.184	0.432	0.477	0.029	0.081	0.083
10	1024	0.174	0.418	0.459	0.028	0.080	0.082
11	2048	0.170	0.413	0.449	0.028	0.080	0.082
12	4096	0.165	0.404	0.440	0.028	0.080	0.081
13	8192	0.160	0.399	0.431	0.027	0.079	0.081
14	16384	0.156	0.390	0.421	0.027	0.079	0.080
15	32768	0.151	0.385	0.412	0.027	0.079	0.080
16	65535	0.146	0.376	0.403	0.027	0.078	0.079

Table 4.1: Estimated execution times for port knocking

requiring no more than twice the time required for a TCP handshake. Since port knocking is typically used to protect services that see few and infrequent connections, an extra connection establishment phase of up to half a second is not unreasonable; many existing services can take as long to initialize and negotiate session keys.

From these results, there is relatively little performance to be gained by using port ranges larger than 256 to 1024 ports; as suggested in Section 3.2.2, administrators should have little difficulty finding unused UDP port ranges of this size.

4.2 Disorder-resistant Knocking

As discussed in Section 3.2.3, most port knocking systems will fail if knock packets are delivered out of order. This section introduces port knocking techniques that do not suffer from this problem. Unfortunately, all of the techniques in this section *will* fail if any packets are lost. Since no packet delivery order is assumed, loss can only be handled by timing out exchanges with suspected packet loss and starting again.

For the purposes of the analysis in this section, the challenge-response unilateral authentication protocol of Algorithm 4.1 will be used under the network model described in Section 4.1.4.

4.2.1 Using Inter-packet Delays

The data in Appendix A suggest that networks re-order packets most often when inter-packet times are low, so simply forcing a high inter-packet time will prevent most re-ordering. My experiments suggest that using 2 ms inter-packet delays will decrease the probability of any given packet being delivered out of order to below 1%; a message of 160 bits, encoded in 20 knocks, should therefore be delivered correctly with a probability of about 90%. For most port knocking applications, this is probably adequate; if re-ordering continues to be a problem, the inter-packet delay can be increased further. Both Rash [Ras06] and Jeanquier [Jea06] suggest that inter-packet delays of 500 ms would be needed to prevent out-of-order delivery; my analysis suggests that this is far more than necessary.

Table 4.2 shows the relationships between inter-packet delays and estimated total protocol execution times for both fast and slow networks. Compared with the times from Table 4.1 for the same algorithm without inter-packet delays, execution times when using 2 ms delays and moderate port ranges on fast networks are less than 20% percent longer than with no delays at all; using 10 ms delays yields penalties no higher than 80% (see Table 4.3). Compared to no-delay execution times, port knocking with inter-packet delays takes much higher penalties on slow networks; however, total execution times of up to a full second are likely still reasonable for most purposes.

Data bits per packet	Ports required	Execution time (seconds)							
		Slow network				Fast network			
		2 ms	4 ms	6 ms	10 ms	2 ms	4 ms	6 ms	10 ms
1	2	1.902	2.378	2.854	3.806	0.605	1.081	1.557	2.509
2	4	1.102	1.338	1.574	2.046	0.338	0.574	0.810	1.282
3	8	0.824	1.000	1.158	1.474	0.251	0.409	0.567	0.883
4	16	0.702	0.818	0.934	1.116	0.204	0.320	0.436	0.668
5	32	0.622	0.714	0.806	0.990	0.178	0.270	0.362	0.546
6	64	0.575	0.653	0.731	0.887	0.162	0.240	0.318	0.474
7	128	0.535	0.601	0.667	0.799	0.149	0.215	0.281	0.413
8	256	0.502	0.558	0.614	0.726	0.138	0.194	0.250	0.362
9	512	0.482	0.532	0.582	0.682	0.131	0.181	0.231	0.331
10	1024	0.462	0.506	0.550	0.638	0.124	0.168	0.212	0.300
11	2048	0.455	0.497	0.539	0.623	0.122	0.164	0.206	0.290
12	4096	0.442	0.480	0.518	0.594	0.118	0.156	0.194	0.270
13	8192	0.435	0.471	0.507	0.579	0.115	0.151	0.187	0.259
14	16384	0.422	0.454	0.486	0.550	0.111	0.143	0.175	0.239
15	32768	0.415	0.445	0.475	0.535	0.109	0.139	0.169	0.229
16	65535	0.402	0.428	0.454	0.506	0.104	0.130	0.156	0.208

Table 4.2: Execution time for port knocking using Algorithm 4.1 and inter-packet delays

Data bits per packet	Ports required	Percent difference in execution time							
		Slow network				Fast network			
		2 ms	4 ms	6 ms	10 ms	2 ms	4 ms	6 ms	10 ms
1	2	33.3%	66.7%	100%	167%	369%	738%	1106%	1845%
2	4	27.3%	57.5%	81.8%	136%	231%	463%	694%	1157%
3	8	20.5%	46.2%	69.3%	115%	170%	340%	510%	849%
4	16	18.8%	39.6%	59.4%	90.4%	132%	364%	395%	659%
5	32	17.4%	34.7%	52.1%	86.8%	107%	214%	321%	535%
6	64	15.7%	31.4%	47.1%	76.5%	92.9%	186%	279%	464%
7	128	14.1%	28.1%	42.2%	70.4%	79.5%	159%	238%	419%
8	256	12.6%	25.1%	37.7%	62.8%	68.3%	137%	205%	341%
9	512	11.6%	23.1%	34.7%	57.9%	61.7%	123%	185%	309%
10	1024	10.5%	21.1%	31.6%	52.6%	55.0%	110%	165%	275%
11	2048	10.2%	20.3%	30.5%	50.8%	52.5%	105%	158%	262%
12	4096	9.41%	18.8%	28.2%	47.0%	47.5%	95.0%	143%	238%
13	8192	9.02%	18.0%	27.1%	45.1%	45.6%	91.1%	137%	228%
14	16384	8.21%	16.4%	25.4%	41.0%	40.5%	81.0%	122%	203%
15	32768	7.79%	15.6%	23.4%	40.0%	38.0%	75.9%	114%	190%
16	65535	6.91%	13.8%	20.7%	34.6%	33.3%	66.7%	100%	167%

Table 4.3: Percent difference in estimated execution time for port knocking using Algorithm 4.1 and inter-packet delays relative to the same algorithm without delays

These data suggest that using inter-packet delays is a feasible approach to the out-of-order delivery problem.

4.2.2 Using Sequence Number Fields

Most protocols that accommodate out-of-order delivery attach sequence numbers to packets so that they can be sorted on arrival. The same can be done with port knocking: the port number can be divided into sequence number and data fields. If the sequence number field is wide enough to uniquely identify all packets, then the server can sort by sequence number when it receives the expected number of packets from a client; no inter-packet delays are necessary.

Unfortunately, this encoding limits the amount of data that can be carried per packet. There is no way to encode a 160-bit message using 13 or more data bits per packet; to do so would require sequence numbers at least 4 bits wide, which is more than can fit in 16-bit port numbers. Likewise, there is no way to encode both data fields and sequence numbers wide enough to uniquely identify all packets over a range of fewer than 512 ports. Table 4.4 shows what can be done with this method. Sequence number field sizes are chosen to use the smallest possible port range for each data field size. There is little point in using 1, 2, 3 or 10 data bits per packet, since more efficient encodings are possible over the same port ranges, but they are included for completeness.

These results show that for any given port range size, using sequence numbers always yields lower execution times than inter-packet delays of 2 ms or longer, with the added guarantee that received sequences can *always* be decoded, regardless of how badly out of order they are. Since the request and response messages are of different sizes, a slight speed gain could be achieved by using different field sizes for each.

Data bits per packet	Sequence number bits per packet	Ports required	Execution time (seconds) Slow network	Execution time (seconds) Fast network
1	8	512	1.426	0.129
2	7	512	0.866	0.102
3	6	512	0.684	0.093
4	6	1024	0.586	0.088
5	5	1024	0.530	0.086
6	5	2048	0.497	0.084
7	5	4096	0.469	0.083
8	5	8192	0.446	0.082
9	5	16384	0.432	0.081
10	4	16384	0.418	0.080
11	4	32768	0.413	0.080
12	4	65535	0.404	0.080

Table 4.4: Estimated execution times and port range sizes for port knocking using Algorithm 4.1 and sequence number fields

4.2.3 Using Disjoint, Monotonically Increasing Ranges

Encodings using sequence number fields often use unnecessarily large port ranges, since the full range of possible sequence numbers is not always used. This can be avoided by making sequence numbers implicit rather than explicit. One way to do this is to use distinct, disjoint, monotonically increasing port ranges for each knock. To send n knocks each carrying b bits of data, n disjoint ranges of 2^b ports would be defined; the first knock would be sent to the first range, the second to the second range, and so on. On arrival, the original data would be reconstructed by numerically sorting the port numbers received and subtracting the beginning of the corresponding range from each.

A major advantage of this technique is that the port ranges used aren't required to be contiguous. For example, the value 0x793F94DC could be sent at 8 bits per knock to port ranges 1024-1279, 1280-1535, 1756-2011 and 2048-2303 as knocks to ports 1145 (=1024+0x79), 1343 (=1280+0x3F), 1904 (=1756+0x94) and 2268 (=2048+0xDC).

Table 4.5 shows the numbers of ports required and estimated execution times for

Data bits per packet	Number of ports required	Execution time (seconds)	
		Slow network	Fast network
1	320	1.426	0.129
2	320	0.866	0.102
3	432	0.684	0.093
4	640	0.586	0.088
5	1024	0.530	0.086
6	1728	0.497	0.084
7	2944	0.469	0.083
8	5120	0.446	0.082
9	9216	0.432	0.081
10	16384	0.418	0.080
11	30720	0.413	0.080
12	57344	0.404	0.080

Table 4.5: Estimated execution times and port range sizes for port knocking using disjoint, monotonically increasing ranges

the possible numbers of data bits per knock. The execution times are the same as when using explicit sequence numbers, but the required port ranges are smaller. Once again, there are no possible encodings using more than 12 data bits per knock.

4.2.4 Using Differences in a Monotonically Increasing Range

Compared to the previous method, a much simpler decoding process can be achieved by encoding values as differences in a monotonically increasing sequence of ports. The first value in a knock sequence would be generated by adding the data value to the beginning of the port range; subsequent knock values would be generated by adding the corresponding data values to the previous knock value. Decoding is straightforward: sort the port numbers numerically and subtract the previous value from each. This technique can also be used with non-contiguous port ranges at the expense of a more complicated decoding procedure. For example, the value 0x793F94DC could be sent at 8 bits per knock to a contiguous range starting at port 1024 as knocks to ports 1145 (=1024+0x79), 1208 (=1145+0x3F), 1356 (=1208+0x94) and

1576 (=1356+0xDC).

The port range sizes and execution times for this method are identical to those in Table 4.5. The top values in port ranges will seldom be used but are necessary in the degenerate case where all data values are large. For example, any value starting with 0x01 and encoded at 8 bits per knock will never use the top 254 ports in its range, regardless of what bytes follow.

4.2.5 Repeating Sequence Numbers

If long knock sequences must be used and available port ranges are extremely limited, then the number of ports required can be limited at the cost of a chance of failure by assuming that all packets sent with "sequence numbers" (either explicit or implied) congruent modulo n arrive in order and reset the packet sequence number every n packets. If sequence numbers are reset every 20 knocks, then port range requirements will be identical to those presented above in this section, although execution times will be longer.

The data presented in Appendix A suggests that packets sent at maximum speed with sequence numbers that differ by 20 will be delivered out of order about 3.7% of the time; this implies that knock sequences of 40 packets with sequence numbers that repeat every 20 packets will be decodable on arrival only 47% of the time. Even when recycling sequence numbers every 40 packets, a 47-packet message will have only a 90% chance of being delivered successfully, and packet loss is a significant issue for sequences longer than about 50 knocks. This suggests that repeating sequence numbers is not a practical method for reducing port range sizes while sending at the maximum rate. However, while sequences of 20 packets sent with 1 ms inter-packet delays and no sequence numbers have a 40% chance of failure, a similar sequence sent with 1 ms inter-packet delays and sequence numbers repeating every 5 packets will be delivered correctly almost 98% of the time. Therefore, while repeating sequence numbers isn't a practical optimization on its own, it is useful for reducing delivery failure rates at minimal cost in when used conjunction with inter-packet delays.

4.3 Alternate Port Knocking Encodings

To generate a cryptographic or one-time port knock sequence, one needs an algorithm to convert a message into a sequence of port numbers. The most common technique (the "standard encoding") is to split the message into bytes and interpret each as a port number within a contiguous block of 256 ports. Other systems use larger or smaller port ranges, requiring slightly more complicated conversion algorithms, but all of them are essentially the same: a message m of b bits is converted into a sequence over 2^n $(1 \leq n \leq 16)$ ports by padding m to an integer multiple of n bits, splitting it into $\lceil b/n \rceil$ pieces of n bits each, and interpreting each piece as a port number. This algorithm is simple and easy to implement, can use any size of port range, and is easily reversed to obtain the original message (assuming that all pieces arrive and that the correct order of the pieces is known). By applying an extra mapping function, it can also be made to work with disjoint port ranges.

However, if it is intended to make a port knock sequence resemble a port scan, then this algorithm is less than ideal. If a set of more than b ports is used, then the sequence will consist of a small number of ports, seemingly randomly chosen from a large space. If a smaller set of ports is used, then, with high probability, many ports in the set will occur once in the sequence, while some will appear more than once and others not at all. Very small sets of 16 ports or fewer will result in each one appearing several times, and, with high probability, each will appear a *different* number of times. None of these cases particularly resembles a typical port scan.

4.3.1 Permutation Knocking

Algorithm 4.5 Keyed set permutation

```
1: $srand(M)$
2: $n \leftarrow size(S)$
3: for $i \leftarrow 0$ to $n - 1$ do
4:     $swap(S[i], S[rand(i, n)])$
5: end for

where $M$ is the message to send
      $S$ is the set to permute
```

An alternate approach is to encode information as a permutation of a small, fixed set of ports ("permutation encoding"). Since there are $n!$ permutations of a set of n elements, a permutation of a set of n ports can encode $\lfloor \log_2(n!) \rfloor$ bits of information. A message of 160 bits would require only 41 ports. Since the set of ports used for such an encoding does not need to be contiguous, a small set of well-known, frequently-scanned ports could be used. If the message to send is used as the seed for a cryptographically-secure pseudorandom number generator [BBS86], then the Randomize-in-Place algorithm from [CLRS01] can be used to generate the permutation; see Algorithm 4.5. Once all ports in the set have been received by a server, it can determine if authentication was successful by generating the expected permutation from locally-available information and comparing the two.

For example, 16 bits of information can be encoded in the permutations of 9 ports ($\lfloor \log_2(9!) \rfloor = 18$). Using the `rand()` function in GNU libc 2.4.11[1] and the port set

$$\{\texttt{21, 22, 23, 25, 53, 80, 110, 137, 139}\}$$

(typically used for ftp, ssh, telnet, smtp, dns, pop3, netbios-ns, and netbios-ssn), Algorithm 4.5 would map the 16-bit value 0x0539 to the permutation

[1]This is a linear additive feedback pseudorandom number generator [Sel07] and is *not* cryptographically secure. It is used here only as an example.

$$\{\texttt{139, 53, 23, 22, 25, 137, 21, 110, 80}\},$$

and map 0x1F48 to

$$\{\texttt{139, 137, 22, 21, 110, 23, 53, 80, 25}\}.$$

Unfortunately, this technique does not provide a method of sending information to a server that is not already locally known to that server (such as nonces, user names, or user-specified commands). If a port knocking client using this encoding needed to send such information, then it would need to be sent in a separate message using some other encoding. Alternately, if the server could restrict the variable values to a sufficiently small domain, it could simply try all of them, but this both increases the computational effort required from the server and reduces the protocol's security (an attacker might now only need to find a valid permutation for *any* user, rather than a particular given user). As a result of this, permutation knocking cannot be used with Algorithm 4.1, although it can be used with single-message, time-based authentication protocols.

Also, although this technique is arguably more stealthy than the standard port knocking encoding, it achieves this at a cost of reduced efficiency. For any given message size, permutation knocking requires both a greater (or equal) number of server ports and a greater (or equal) number of knocks than an appropriately-chosen standard encoding. A mathematical proof of this is given in Appendix B.

Finally, this method is not resistant to out-of-order delivery. In order to protect against out-of-order delivery, one of the techniques from Section 4.2 must be used. All of the techniques involving sequence numbers require use of much larger sets of ports and result in packets only being sent to small subsets of them; relative to "normal" permutation knocking, this would be easily distinguished from port scanning and would therefore negate the additional stealth provided by permutation knocking.

Since port scans frequently delay between sending packets in order to avoid detection, inter-packet delays remain an option for providing disorder-resistance to permutation knocking without compromising its design goals.

4.3.2 Bit Knocking

Since a message to be sent to a port knocking server is just a sequence of b bits, a final approach is to define a set of b ports on a server, numbered 0 through $b-1$, and send a knock to port b if bit b is set in the data and no knock if it is not set. These can be sent in any order. If the message is statistically random (such as a nonce or a MAC output), then $b/2$ bits will be set on average, so an average of $b/2$ knocks will be required. Such a knock sequence can be decoded by a server by simply initializing a data value to zero and setting bits as the appropriate knocks are received.

Since the server doesn't know *a priori* how many bits are set in the data, it can assume that all knocks will arrive within a fixed time window after the first. If the server is expecting a MAC or hash value, it can check the data for validity after each bit is received, although this won't work for values that the server doesn't know (such as nonces) and makes the protocol easier to attack. Neither of these methods can be used when the message to be sent is all zeroes; the server will receive nothing and won't know to do anything. Alternately, the client could send a "stop knock" to an extra port outside the data range to signify the end of data; since this knock may be delivered out of order, the client should delay 2-10 ms before sending it.

Bit knocking requires port sets larger than required for permutation knocking, but still small enough to resemble legitimate port scans. It requires many more knocks than the standard encoding, but, unlike permutation knocking, it *is* resistant to out-of-order delivery and uses fewer ports to do so than any of the sequence number-based encodings in Section 4.2. Using this encoding, Algorithm 4.1 will take an average of

1.053 seconds on a slow network or 0.111 seconds on a fast one under the conditions specified in Section 4.1.4.

4.4 Preventing Race Attacks

Most existing port knocking and SPA systems do not logically associate their authentication exchanges with the connections that are subsequently opened, potentially allowing attackers to gain access to protected services by waiting for authentication exchanges to complete successfully and then impersonating legitimate clients to connect (*race attacks*). Some partial solutions to this problem have been proposed, described in Chapter 3; further possibilities will be introduced here.

Note that no attempt is made here to prevent attacks after connections have been opened, such as TCP connection hijacking [Mor85]. If such attacks are a concern, then a system that provides both authentication and confidentiality of connections, such as IPsec or TLS, should be used instead of (or in addition to) port knocking or SPA.

4.4.1 Server-chosen Port Numbers

Cappella and Tan's port knocking system [CK04] avoids the issue of unassociated authentication exchanges and connections by letting the server, rather than the client, choose the port that is to be opened after a successful authentication. In their system, the server chooses a random high-numbered port, forwards it to the requested service, and sends the new port number back to the client in an encrypted message. Their system uses a one-time authentication scheme and servers that only respond after successful authentications, but their method can also be used with challenge-response protocols; the encrypted port number can be appended to challenge messages and forwarded to the destination only if authentication is successful.

Of the solutions to the race problem presented here, the simplest and easiest to implement is to use server-chosen port numbers. However, this solution is equivalent to choosing a 16-bit session key, which may be considered too small for good security. It is possible, though improbable, for an attacker to discover the open random port after authentication has completed but before the legitimate client has had the chance to connect. Having the source port for the client also chosen randomly would strengthen this method to the equivalent of a 32-bit session key, although it may be difficult to make this work with client software that cannot bind to all possible local port numbers.

If mutual authentication is being used, then the server-chosen random port number could be replaced with one agreed upon using something like the Diffie-Hellman protocol [DH76]. However, care must be taken that the key agreement is not vulnerable to man-in-the-middle attacks and that it is properly associated with the authentication [DvOW92].

4.4.2 TCP ISN Agreement

As an alternative to server-chosen or mutually-agreed port numbers, TCP initial sequence numbers (ISNs) could be selected and used to identify connections. For unilateral authentication, servers could choose ISNs for clients and send them back in encrypted packets or, for mutual authentication, both parties could choose each others' ISNs or negotiate them using something like Diffie-Hellman. Any connection opened after the authentication exchange that does not use the expected TCP ISNs would be rejected.

TCP ISNs are 32 bits wide (64 bits if both client and server ISNs are chosen), so such a system would be considerably more secure against brute-force attacks than choosing destination ports. Unfortunately, such a system would be difficult to implement, since it requires that either clients and servers are both implemented in kernel-space (where TCP ISNs are typically chosen) or that operating system kernels be patched to support user-space ISN generation. Also, such a scheme can only be used with TCP; it won't work with UDP and other protocols that lack sequence numbers.

4.4.3 Combining Authentication and Connection Establishment

There is no better way to create logical associations between authentication exchanges and connection establishments than to build the authentication into the connection establishment itself. Tailgate TCP [BHI$^+$02] accomplishes this by attaching authentication information as data to clients' TCP SYN segments. An equivalent using challenge-response authentication could also attach authentication data to servers' SYN and clients' ACK segments in TCP connection establishment. A parallel using UDP would be to forward the port used for authentication to the requested service on the servers' side after successful authentication and require the client to use the same local port as the authentication client. Due to the way that most stateful firewalls generally track UDP "connections" (see Section 2.4.1), they will think that everything is part of the same connection.

Similar systems can be constructed at the application layer with services that use connections passed to them from wrapper programs (such as `inetd` [LFJ$^+$86] and `tcpd` [Ven92] on Unix-based systems), rather than opening network connections themselves. Such services neither know nor care about any data transmitted over a connection before it is passed to them, so the wrapper program could conduct an authentication exchange with the client.

Unfortunately, this type of system has its drawbacks. Some routers and firewalls may drop or strip TCP SYN segments with payload data, and attaching authentication data to the final ACK segment in TCP connection establishment will not work with TCP implementations that attach application data to such segments. The TCP version also requires kernel-level servers that can process authentication information before it reaches the transport layer of the server's network stack. The UDP version requires that client port numbers be chosen by the authentication client, rather than choosing their own. The application-layer version requires either specialized clients that conduct authentication exchanges immediately after opening connections (which is hardly transparent to the application) or clients that can receive connections handed off from wrapper programs (which are not common). Finally, although this type of system works well with SPA-based authentication schemes, it is difficult to use with port knocking-based systems.

4.5 Implementing Port Knocking and SPA

The preceding sections have presented a large number of options for the designs of port knocking and SPA systems. Some, I have shown to be suboptimal or undesirable. Others are more practical, although ideal choices depend strongly on the environments under which port knocking or SPA systems will be used. In this section, I present the designs of prototype port knocking and SPA systems that demonstrate the feasibility and advantages of several of the techniques presented in this thesis.

The two systems, port knocking and SPA, were designed to be identical except for the communication mechanism used. Both use unilateral challenge-response authentication, following the protocol in Algorithm 4.1. Server configuration is organized into a series of rules, each associating a unique request sequence with secret key and a port to be opened upon successful authentication. Since clients may not know their public IP addresses, required by Algorithm 4.1, rules may also contain directives to ignore client IP addresses in authentication; in this case, both clients and servers substitute the value '0' for the client address.

The port knocking system uses "pure" port knocking over UDP using the standard encoding and the implicit sequence numbering scheme from Section 4.2.3: clients send requests and responses, while servers issue challenges in single UDP packets. While this scheme does require large numbers of ports, it is guaranteed to be disorder-resistant and minimizes protocol execution time. If the required port range was an issue, bit knocking would be a good alternative. The SPA system uses single UDP packets for all messages.

Both the port knocking and SPA systems attempt to prevent race attacks by using server-chosen port numbers: when a server issues a challenge to a client, it attaches a random port number, encrypted with AES128-ECB and the secret key associated with the request sequence issued, to the message. If the client successfully authenticates, then the server forwards traffic sent to this port by the client to the port specified by the rule matching the request used. Since AES-128-ECB uses 16-byte blocks, while port numbers are only 2 bytes, each encrypted block sent also includes a 7-byte random number (a *confounder*) to foil known-plaintext attacks and the first 7 bytes of a SHA-1 hash covering the random port number and confounder to prevent forgery. A cipher with a smaller block size, or a stream cipher, could have been used instead of AES, but the resistance to known-plaintext attacks and forgery is useful and there is little penalty in increasing the size of a single UDP datagram by 14 bytes. The method of using server-chosen port numbers to prevent race attacks is not particularly strong by cryptographic standards — it is equivalent to negotiating a 16-bit session key — but it is the only method presented in Section 4.4 that does not require significant modifications to existing client software, operating systems, or both.

After successful authentication exchanges, traffic from a single connection from the client to the random port is accepted; all other traffic is dropped. Connection tracking is deferred to the firewall's built-in connection tracking system.

Both servers were implemented on Linux systems using recent 2.6.x-series kernels. They use the `NFQUEUE` kernel module and `libnetfilter_queue` library provided with recent versions of `iptables` to copy packets destined to specific ports to userspace programs that implement the authentication algorithms; these packets are then dropped. When an authentication exchange completes successfully, the userspace program sends a message to a custom kernel driver requesting that the next connection from the client to the randomly-chosen port number within a specified time be redirected to the destination port of the rule matched. Packets associated with such connections can be matched by `iptables`' built-in `conntrack` match[2] and accepted by a normal `iptables` rule; no run-time modification of firewall rules is needed. Higher efficiency might have been achieved at the cost of increased development difficulty by moving the authentication systems into kernel modules, but I do not feel that the difficulty of moving the core functionality into the kernel is worth the limited efficiency benefits. Clients were designed to operate as unprivileged userspace applications that do not rely on any system-specific interfaces; they were only tested on Linux, but should be easily portable to other operating systems.

One issue to consider when implementing port knocking servers that has not previously been discussed is how request sequences are matched. Servers cannot predict the order in which request packets will arrive, and due to the possibility of packet loss and request retransmission, they cannot make assumptions on where request sequences begin, either. The simple way to decode requests under these conditions is to process packets in the order of their sequence numbers, rather than order of ar-

[2]Linux's `netfilter` firewalling subsystem, used by `iptables`, is based on lists of rules consisting of sets of "match" functions and a "target" function; the target is called if all matches succeed. Many different match and target functions are available, and more can be added through kernel modules. More details on `netfilter`'s architecture are available in [And06].

rival, but this requires that servers cache port numbers with high sequence numbers until all packets with lower sequence numbers have been received. This allows attackers to launch resource-consumption attacks by sending large numbers of packets with random source addresses and sequence numbers greater than 1. In order to handle packet loss, port numbers must eventually expire from servers' caches, but sufficiently high rates of malicious packets could consume large amounts of memory. This can be avoided by processing packets in order of arrival and immediately discarding anything that cannot be part of a recognized request sequence. However, while efficient string-matching algorithms such as Aho-Corasick [AC75] work well for sequences that are processed in sequence order, they are suboptimal for matching input in random order. Aho-Corasick in particular would require storage space exponential in the length of request sequences and exponential processing time at start-up to match against input presented in random order. Other algorithms can be devised which, despite being suboptimal for input presented in order, do not degrade in performance when input is out of order; details are beyond the scope of this thesis. Since request strings are necessarily presented to SPA servers in order with fixed starting positions, they do not suffer from this problem; servers can use tries [Fre60] to match request sequences very efficiently.

4.5.1 Performance of SPA and Port Knocking

Covert authentication systems such as SPA and port knocking are not the only means to grant authenticated access to services through firewalls; existing protocols such as SSH and IPsec can be used to achieve the same goal. These services provide much more functionality than do SPA and port knocking, including connection integrity and confidentiality, but do so at a price: they are complex and visible, leading to relatively high threat exposure (as mentioned in Section 3), and the extra work that they perform makes them slower and more resource-intensive than SPA or port knocking.

To measure the performance penalty imposed by these other services, I compared the execution times of my SPA and port knocking systems against an SSH system. To fairly compare the three, they must perform a comparable task; I chose to make authenticated connections to a server and execute the trivial program `true`. The procedure to test SPA and port knocking was to first authenticate to the SPA or port knocking server, parse the random destination port number from the authentication client's output, send the command "/bin/true" to a simple execution server listening on the protected port, and wait for the response. The request messages used for both SPA and port knocking were 80 bits long; the port knocking test used 7 data bits per packet. SSH clients accept a command to execute on successful authentication as an argument; this was set to "/bin/true". SSH was tested using 2048-bit RSA public key authentication and a key server.

Both real and CPU execution times were measured for all systems. CPU times were measured as the sum of the user and system times given by the `time` command. For SPA and port knocking clients, CPU time includes the authentication, parsing, and connection programs; SPA and port knocking server CPU times include the authentication server, execution server, and the command executed. CPU times for SSH are the total execution times of the SSH client and server and the command executed. Client times were measured by repeatedly running the full client command sequence and averaging the times measured; a single server process was used to handle all connections, and it's execution time was averaged for the number of connections made. SPA and port knocking clients were run 1000 times per test; SSH clients were only run 50 times per test to save time.

Network	Bandwidth	Round-trip time
Dial-up	21.6 kbps	55.3 ms
Broadband	674 kbps	15.8 ms
LAN	36.4 Mbps	0.063 ms

Table 4.6: Test network characteristics

Tests were conducted over three types of network:

1. a 56 kbps dial-up modem link,

2. a residential broadband network, and

3. a 100 Mbps ethernet LAN.

Due to limitations imposed by network topology and hardware availability, I was unable to use comparable hardware for clients and servers, or to use the same client computer for all tests. The server used for each of these tests was a 10-year old PC with a 200 MHz Intel Pentium MMX processor and 128 MiB of RAM running Linux 2.6.20. The client used for the LAN test was a 1920 MHz AMD Athlon XP with 1 GiB of RAM running Linux 2.6.20; the client for the other tests was a dual-processor 860 MHz Intel Pentium III with 256 MiB of RAM, running Linux 2.4.21. Bandwidth and round-trip time measurements for the three networks, obtained by transferring a 1 MB file by FTP and running the `traceroute` utility, are given in Table 4.6.

The results of my test, to three significant figures, are given in Table 4.7. All times are in milliseconds. Generally, they are as predicted: port knocking takes slightly longer than SPA, due to the larger amount of data transferred, while SSH takes significantly longer, due to the larger amount of data transferred and larger amount of processing done. There are a few anomalies in the data for which I have no good explanation, such as port knocking over broadband requiring less client CPU time than SPA, but the general trend is still clear. Unfortunately, the difference in network

Measurement	Network type	Execution time		
		SPA	Port knocking	SSH
Real time	Dial-up	528 ms	680 ms	25100 ms
	Broadband	114 ms	206 ms	2880 ms
	LAN	61.8 ms	83.8 ms	2000 ms
Client CPU time	Dial-up	19.7 ms	19.9 ms	62.4 ms
	Broadband	19.8 ms	16.3 ms	58.0 ms
	LAN	4.92 ms	7.05 ms	17.5 ms
Server CPU time	Dial-up	6.80 ms	7.01 ms	2330 ms
	Broadband	14.5 ms	25.3 ms	2340 ms
	LAN	17.0 ms	27.1 ms	1770 ms

Table 4.7: Execution times for SPA, port knocking, and SSH

conditions between those experienced in this test and those used in the simulation of Table 4.4 makes comparison with the simulated results difficult.

Given these results, it is clear that SPA and port knocking, even using challenge-response protocols, are practical and faster than SSH for authenticating connections to firewalls. Due to similarities in the computations performed, I would further expect that other alternative authentication protocols, such as IPsec or SSL, would have execution times similar to SSH.

4.6 Summary

In this chapter, I have discussed methods for performing challenge-response authentication over port knocking and SPA, introduced several methods for making port knocking resistant to out-of-order delivery and two novel ways of encoding data into port knocking sequences, and described a few possible ways to prevent race attacks. The feasibility of several of these techniques has been demonstrated through implementation.

Challenge-response authentication can solve many of the problems of the various single-message authentication protocols used in existing port knocking systems but at the expense of increased message complexity and execution time. However, execution time is not prohibitively high under reasonable network models, thus making it suitable for most port knocking and SPA applications. Unlike existing systems, which provide only unilateral authentication, the challenge-response architecture also makes mutual authentication possible.

Port knocking can be made resistant to out-of-order delivery by either enforcing inter-packet delays to reduce the chance of packets being delivered out of order or by encoding explicit or implicit sequence numbers into knock sequences. Experimental data suggest that inter-packet delays of as little as 2 ms are adequate for ensuring that packets are delivered in order most of the time, but methods using implicit sequence numbers are both more reliable and faster, though at the expense of requiring larger port ranges.

Messages are normally encoded into port knock sequences by splitting them into fixed-size pieces and interpreting each as a port number; two alternate methods are to encode messages as permutation of a fixed set of ports or by sending knocks to ports representing all of the bits that are set in each message. Though both of these new techniques require smaller port ranges than the standard encoding, they unfortunately require significantly longer execution times and have a greater chance of failure, due to the larger number of knocks that they require.

Race attacks can be prevented by negotiating the port number to be opened or the client's and server's TCP ISNs, or by combining the authentication exchange with the connection establishment. Unfortunately, none of these techniques are both compatible with existing software and safe against all attacks.

Chapter 5

Improvements to Application Filtering

I don't need a firewall; nobody is interested in my data.

– Far too many people

Port knocking and SPA are primarily designed to communicate information about users to remote ingress packet filters, allowing them to filter on a per-user basis without relying on assumptions about IP addresses. However, neither are ideal for communicating with egress firewalls on local networks. On relatively trusted local networks, stealth is irrelevant: everyone on the inside knows where the firewall is and what it is capable of doing; there's no value in hiding it. Also, whereas filtering by applications is of limited use to ingress firewalls (allowing connections from poorly-implemented clients presents little risk to the server and malware is not likely to be able to authenticate as a valid user), it is of significant value to egress firewalls. Finally, whereas port knocking and SPA are designed to protect services that receive small numbers of connections, egress firewalls must handle much larger numbers; the overhead of port knocking may be a liability in this case.

Application filtering provides an efficient mechanism for host and network firewalls to make decisions based on the users and programs that are sending and receiving network traffic. This chapter discusses existing designs and implementations of application filters and a variety of common problems with such systems. It then presents partial solutions to some of these problems.

5.1 Existing Application Filtering Systems

Since user and application information is not included in IP packet headers, this information is generally not available to network firewalls. Some circuit and application gateways use special protocols to authorize connections that authenticate users (such as SOCKS [LGL$^+$96]), but no existing network firewalls seem to make any checks on the applications making connections.

Host firewalls, on the other hand, do have access to user and application information. Although the built-in firewalling systems on most operating systems don't support application filtering, most third party packages do. Judging by its behaviour, `ZoneAlarm` [Zon07] (a popular commercial firewalling package for Microsoft Windows) checks hashes of programs and shared libraries that attempt to make connections and performs some tracking of programs that make connections on behalf of others. By default, when it detects a connection attempt from an unknown program which is not otherwise explicitly allowed or denied by the current configuration, it prompts the user for a decision. Unfortunately, information about the algorithms used internally by this and other proprietary packages is not publicly available. Prior to August 2005, the `netfilter` firewalling system on Linux supported a form of application filtering through the `owner` match module. This support was removed in kernel 2.6.14 due to conflicts with kernel locking [HM05]; the `owner` match still supports user and group matching. Originally, this module attempted to match packets to specified program names or process IDs by iterating over the list of currently running processes, searching for one matching a given command name or process ID, and then iterating over all files held by that process to check if any of them matched the socket used by the packet being matched; there was no interactive component. `TuxGuardian` [dS06] provides application-filtering support to Linux using a different approach: rather than working through netfilter, it uses the *Linux Security Modules* system [WCM+02] to hook into attempts to open sockets or listen for connections and call out to a user-space program to ask for permission to allow the attempt. The user-space program checks both the program's name and the MD5 hash of the executable file against its configuration, optionally prompts the local user if the program is not recognized, and allows or denies the request.

5.2 Problems with Application Filtering

Application filtering is a useful and powerful technique, but it cannot be trusted absolutely. It has many flaws, both in concept and in implementation, that restrict what application filters can feasibly do.

5.2.1 Dealing with Unrecognized Applications

The most obvious problem is what to do when unrecognized applications attempt to make connections. There are three possible responses in this situation:

1. always allow the request,
2. always deny the request, or
3. ask the user if the program should be allowed to use the network.

The first option is dangerous: the firewalling software cannot be expected to have a comprehensive database of all untrustworthy software and will end up letting worms and spyware communicate with impunity. No firewall should accept by default; application filters are no different.

The second requires that the firewall have a comprehensive database of all trusted software. This may be appropriate for professionally-managed networks, where the set of allowed applications is well defined and relatively small, but is problematic in other situations where no such list is available. It is infeasible to require firewall vendors to ship lists of trusted software with their products and impractical to trust them if they do. It is equally infeasible to require end users to build such databases when installing the firewalling software. Applications themselves can obviously not be trusted to assert their own trustworthiness to the firewall.

The third option is also troublesome. People will make mistakes and grant access to programs that they meant to deny. Many users are not sufficiently familiar with their computers to know what applications they should trust and under what circumstances they should trust them. Many will simply click “accept” to any request for a connection, regardless of what is asking [Nor83]. Others will not know the names of all trusted programs, but will recognize that applications should only make connections in response to their requests and will only grant permission shortly after performing some action that could reasonably be expected to make a network connection. However, malware could abuse this trust by monitoring other software and attempting to make network accesses only when something else does. This is compounded by the fact that many network-using programs do not make connections themselves but pass requests to other processes via inter-process communications (IPC), resulting in the application names being presented to the user having little to do with what the user is running. For instance, in some versions of Microsoft Windows, the Windows Update program’s connection requests show up to many firewalls as originating in a program called `wupdmgr.exe`, despite being initiated by Microsoft Internet Explorer (`iexplore.exe`). Also, many components of Microsoft Windows (such as the DHCP client, network browser, time synchronizer, and messenger) make connections through a program called `svchost.exe` [Mic06], and some legitimate software (such as time synchronizers) runs automatically, rather than in response to user inputs; users who are unaware of this may deny network access to important software. Finally, malicious programs could explicitly instruct users to allow them access; suitably convincing messages would persuade many users to grant access when they otherwise would not [CBR03].

5.2.2 Application Spoofing

It is not sufficient for application filters to identify trusted programs solely by executable file names. Anyone could give an arbitrary program the same name as a trusted program; without further checks, an application filter could be completely bypassed in this manner. Some existing malware attempts to use this approach by mimicking operating system components or other trusted software. For instance, the `Welchia` worm stores a copy of itself with the name `svchost.exe` under a different path than the legitimate program of this name on Microsoft Windows systems [Sym07d].

A slightly stronger approach is to identify trusted programs by the absolute paths of executable files. This would prevent the sort of file name spoofing used by `Welchia` but would not catch malware that modifies or overwrites trusted programs, such as some versions of the `Spybot` (which insert themselves into the command-line FTP client on Microsoft Windows systems) [Sym07c] or `Erkez` (which attempts to overwrite executable files belonging to Symantec products) [Sym07b] worms. This approach would also be ineffective if the firewall's view of the directory tree could be changed; on Unix-based systems this could be done by mounting a new filesystem on top of an existing one so that a new program occupies the path of the original.

A stronger approach again is to identify trusted programs by cryptographic hashes of executable files. This would detect any attempts to modify or replace trusted programs and would defeat the above worms. However, even this method can be attacked. Malware need not live in executable programs; it can live in shared libraries. Legitimate shared libraries could be modified to launch malware as a side-effect of a normal library call, or malicious plugins could be written to launch malware instead of their advertised functions. For example, the `Fuwudoor` back door takes advantage of `svchost.exe`'s ability to run code from arbitrary shared libraries on Microsoft Windows systems to launch itself [Sym07a]. File hash checking may also be vulnerable to race conditions: malware could overwrite a legitimate file, launch, and overwrite itself with the original, legitimate file before attempting network access. An application firewall that checked file hashes would see only the legitimate trusted program, and grant access to the malware. The directory re-mapping trick suggested above could also be used to accomplish this.

Attacks using plugins or shared libraries could be prevented by not only verifying the executable files making network requests, but also all shared libraries currently linked; however, malware could switch shared libraries just as easily as it could executables. Defeating the file-switching attack is more difficult; unless the operating system prevents modifications to currently-loaded executable files and shared libraries, then no checks of these files can be trusted. Malcode inserted into a legitimate running process via a buffer overflow or other exploit also cannot be detected using checks against files.

5.2.3 Interpreted Languages and Virtualization

Application filtering relies on being able to uniquely identify the application that is requesting a connection. With traditional compiled programs, this is feasible; each instance of such an application is loaded and has its resources managed by the operating system, so it is possible to map packets to programs. Unfortunately, when programs written in interpreted languages load or request resources, the operating system sees the interpreter, not the program itself. To an application firewall, a news reader running in a Java virtual machine is indistinguishable from a backdoor running in a Java virtual machine.

This could be fixed by requiring interpreters to pass information to operating systems about what program they are currently executing, but the diversity of interpreters makes this infeasible. Growing numbers of applications have built-in Turing-complete scripting languages capable of making network connections, and there is no feasible way for operating systems to check if a given program can execute arbitrary code within its execution context. Even if there was, this moves critical firewalling functionality to untrusted user-space programs; there is no way to prevent interpreters from lying to the operating systems about what they are running. Finally, the name of a script is largely meaningless to a firewall; an application filter must make some check against the script's codebase. But interpreted languages may not have code that exists on disk; it might exist solely in a memory buffer or even be retrieved on demand from an external source. In other words, application filtering is mostly useless against interpreted programs.

The same problem applies to programs running in virtualized environments. No program running in a virtualized environment, compiled or interpreted, can be accurately identified by the host operating system unless the virtualized environment is trusted to the same degree as the host OS and can supply information about internal processes to the host. Although the relatively small numbers of virtualization platforms available makes adding such support feasible, many existing virtualization packages (such as `VMware` [VMw07]) are designed to run operating systems that were not specifically designed for virtualization, and may not even be aware that they are running in virtualized environments. Thus, proper support for application filtering virtualized environments would be at cross-purposes with many existing tools and is not likely to be available any time soon.

5.2.4 Connections by Proxy

Programs that use network resources do not necessarily make network connections themselves. They can instead start different programs or pass requests to existing processes to perform actions on their behalf [CBR03]. An application firewall will see connections originating in the processes that attempt to open them, not the processes that requested them. This flaw can be abused both ways: legitimate programs that pass network connections to others, such as `inetd`, may be fooled into passing connections to malicious software, and malware could use legitimate software, such as web browsers, FTP clients or the `ping` command, to do its dirty work.

Detecting the true originator of network connections requires tracking process parent-child relationships and IPC. Some commercial application firewalls, including `ZoneAlarm`, appear to use some form of this, but information on the algorithms involved is not publicly available.

5.2.5 Attacks Against Firewalling Software

Instead of attempting to exploit weaknesses in how application firewalls verify that applications are trusted, malware could attack the application firewalls directly. Malware that attempts to shut down security software is known to exist in the wild [Sym07b]; user-space application filters may have no effective defense against this sort of attack. Malware running at sufficiently high privilege levels may be able to bypass firewalls and send or receive packets without firewall checks. Malware may even be able to interfere with firewalls' perceptions of file contents or paths by intercepting system calls; rootkit software frequently uses this technique to hide itself from IDSs.

This is not a likely attack against host firewalls — any malware that has compromised a host to the point that it can do this is more likely to simply disable, cripple, or circumvent the firewall — but could be used against application filters on network firewalls. For these reasons, host firewalls should never be trusted to the same degree as network firewalls, and even network firewalls should not fully trust packet metadata that they cannot independently verify.

5.3 An Improved Architecture for Application Filtering

As pointed out above, application filtering has many flaws. In this section, I present an architecture for application filtering that addresses many of them. It is possible that some or all of these techniques are used by existing proprietary software, but to the best of my knowledge, none of these have appeared in published literature.

5.3.1 Application Filtering by Network Firewalls

Since user and application information is normally not available to network firewalls, application filtering is normally only done by host firewalls. However, there is no reason why it can't be done at network firewalls. Protocols like SOCKS, TAP [Ber92] and Ident [Joh93] can be used to pass information about users to network firewalls; these could be extended to supply application information as well. There are disadvantages to such a system, the most notable being the overhead required for passing user and application information and the possibility of forgery, but there are also a number of advantages. Since there are few avenues to execute arbitrary code on network firewalls (as compared to hosts employing host firewalls), it is more difficult for malware to attack and circumvent network firewalls. Also, network firewalls allow centralized policy management without needing to distribute policy to hosts, thus avoiding the problems that policy distribution entails (see Section 2.4.1).

Two basic methods can be used for application filtering; either can be extended to enable application filtering by network firewalls. One way is to look for packets that open connections and then look up the users and applications that sent or will receive them. This is the approach taken by the pre-2.6.14 Linux `owner` match, and is necessarily inefficient since the application must be identified by matching the properties of the packet to a socket in kernel data structures, an operation which requires at least $O(n)$ operations (where n is the number of currently-executing processes) and requires locking. However, if this is done inside a stateful packet filter, which already has the ability to detect new connections, then it is trivial to combine application and packet filtering. The other way is to hook into appropriate system calls (such as `connect()` and `accept()`) to intercept applications' attempts to open connections. Since this must be done on the hosts running the applications and occurs within the applications' execution contexts, it is trivial to identify the applications involved using this method. However, packet filtering is not normally done inside system calls, so it may be difficult to perform both packet and application filtering using this method.

Either design may be extended to support application firewalling at network firewalls. Again, two basic approaches are possible: hosts wishing to open connections can pre-approve them with firewalls, or firewalls can detect attempts to open connections and ask hosts for application information. Network firewalls can only detect new connections by packet analysis, so the obvious place to implement application filters is inside stateful packet filters or circuit gateways. Hosts using the pre-approval strategy could detect new connections using either method (although system call hooking is more efficient) and send information about the user and application to the firewall before sending the connection-opening packet. Inbound connections can be handled in the same way: a host that detects an application attempting to listen for connections can inform the firewall of this and let the firewall decide what to do when an attempt to connect arrives. Pre-approval requires a minimum of only one extra message (for outbound connections, pre-approval messages could be attacked to connection-establishment packets, but this may interfere with some existing protocols), sent from the host to the firewall, making it relatively fast, but does have a few drawbacks. Since pre-approval messages can be generated at any time, malicious hosts could attempt to flood firewalls with pre-approval messages for connections that they have no intention of establishing. Pre-approval messages must expire after some period of time in order to prevent firewalls from accumulating too many valid pre-approval messages for programs that have exited or will otherwise never follow through. Short lifetimes will require hosts listening for connections to periodically send new pre-approval messages as long as they continue to listen; long lifetimes may make forgery easier. When using the call-back strategy, hosts could either look up processes via socket addresses on demand or build a list of processes that are attempting to open connections via system call hooking, then look up requesting processes there. Either method is slower than the pre-approval strategy, especially since this approach requires a minimum of two messages: a request from a firewall to the host and a response. However, approval lifetimes and DoS attacks against the firewall are no longer an issue.

In order to make forgery of pre-approval messages more difficult, pre-approval messages for outgoing packets should include a hash of the packet to be sent. Since the contents of connection-establishing packets may be predictable, it may be preferable to use a MAC that covers both the packet and a nonce or timestamp instead. The contents of packets won't be known when pre-approval messages are created for hosts that want to receive connections, but MACs covering the hosts' addresses could be used in this case. Similarly, information requests for outgoing messages under the call-back strategy should contain a hash or MAC covering the packet being sent; the originating host can then verify that it actually sent that packet.

In order for any form of application filtering at network firewalls to work, the operating systems (although not the user-space processes) of all protected hosts must be trusted; malicious operating systems (such as those infected with rootkits) could lie to the firewall about what processes are running or any other requested information. However, if application filtering is combined with some other form of firewalling on a network firewall, then traffic from hosts with malicious operating systems is still subject to some form of filtering, whereas such traffic can bypass any form of host firewall.

5.3.2 Preventing Application Spoofing

Since attempts to verify that connections come from legitimate applications by checking files on demand suffer from race conditions and are prone to abuse, an alternate method must be found. One possibility is to verify programs and libraries when loaded, as done by integrity shells [Coh89], and cache the hashes of each file until needed. This scheme will generally require that all programs and libraries, not only those involved in making network connections, be verified; if this overhead is too high, then programs expected to make network connections could be flagged for load-time verification, and connection attempts from all others could be summarily refused. Applications will take longer to load when using this method, but there will be less overhead when opening connections at run-time. If the operating system can load and verify files atomically, this should not suffer from race conditions. Even if a race condition still exists, it should only be exploitable in a much smaller time window (no more than a few milliseconds) than the race condition with on-demand checking.

However, this approach is not without flaws. As with on-demand file checks, it is ineffective against interpreted programs or those running in virtual machines. Hostile operating systems could substitute hash values for legitimate programs to disguise malware. If a collision can be found in the hash function used, then a malicious program could be substituted for a legitimate one even on a computer with a trusted operating system. A network firewall could foil this last attack by requesting a hash or MAC covering both the program and a nonce, but this requires that hashing be done on demand and that the firewall store copies of all relevant files. An alternative possibility is to eschew files altogether and hash processes' memory images instead. Obviously, writable memory pages must be excluded from such a hash. Unfortunately, this will not work on systems that perform any sort of relocation at load time (including most modern operating systems), since memory addresses inside the code in processes' memory images may differ between executions [Lev00]. Even if all memory addresses were constant between executions, a hostile operating system could still substitute the memory image of a legitimate program in the hashing algorithm or could allow another process to launch a page-replication attack along the lines of Wurster et al.'s [WvOS05]. Finally, this method can't effectively verify programs that execute code from writable pages in memory (although operating systems can prevent this by preventing code execution from writable memory pages, as in PaX [the05]).

Neither method presented here will detect processes that are executing code injected by malware (through buffer overflows or other exploits) rather than their own original code. Other countermeasures (such as non-executable writable memory and address space randomization [dR07, the05, BDS03]) must be taken against these threats.

5.3.3 Detecting Connections by Proxy

Detecting libraries linked to a process, malicious or otherwise, can be done by simply monitoring what it loads or analyzing its memory space (assuming that processes cannot execute code from writable memory). However, detecting actions taken on behalf of other processes is more difficult. In this section, I present an algorithm that will detect many situations where a process could be opening connections on behalf of others.

A process will be considered to be opening a connection on behalf of another if there is a possible causal relationship between an action of another process and the connection establishment. A causal relationship exists whenever a process receives information from another process before opening a connection. For example, a process could read from shared memory or a socket, pipe or other IPC mechanism to which another process had written. Alternately, a process could have been started, directly or indirectly, by another process. Files could also be used for IPC, but, since files are persistent, tracking reads and writes to files over long periods of time may not be practical; for this reason, a configurable limit of n seconds is placed on how long records of file writes are kept. Using these rules, a directed graph of process interactions can be constructed as follows:

- When a process starts another, a link is added from the new process to the old one.
- When a process writes to shared memory or to a pipe, socket, or other object that is shared between processes, its identity is added to a list associated with that object.
- When a process receives a signal from another process or reads from shared

memory, a pipe, socket, or other object that is shared between processes, a link is added from the reader to all processes that have written to it.

- When a process writes to a file or other persistent object, its identity is added to a list associated with that file, which may only be removed after at least n seconds have passed.

- When a process reads from a file, a link is added from the reader to all processes that have written to it within the past n seconds.

Note that the process relationship graph must include processes that have terminated but which still have causal relationships to existing processes. Processes' identities must include all information that the application filter uses to identify processes, such as executable file names, shared library names, and file hashes. Using this graph, all processes having causal relationships to a process that is attempting to open a connection can be identified by performing a traversal of the weakly connected component of the graph rooted at the process that is attempting to open the connection; this set of processes will be known as a *process group*.

As an example, consider the following interactions:

1. Process A starts process B

2. Process B starts processes C and D

3. Process E writes to a pipe which is read by process C

4. Process D writes to a multicast socket, which is read by processes E, B, and F

The graph generated for these interactions will then be that shown in Figure 5.1. The groups for all processes are as shown in Table 5.1.

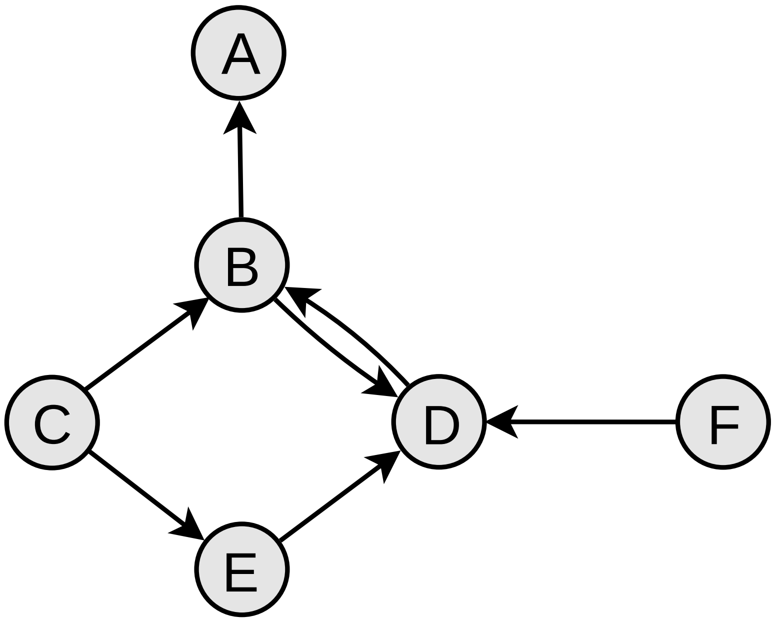


Figure 5.1: Graph of causal relationships between processes.

Process	Process group
A	{A}
B	{A, B, D}
C	{A, B, C, D, E}
D	{A, B, D}
E	{A, B, D, E}
F	{A, B, D, F}

Table 5.1: Process groups for all processes in Figure 5.1

A firewall that attempts to detect connections by proxy in this manner would need to be configured with not only the set of programs allowed to make connections, but also all other programs allowed to be in the process groups of a program that is attempting to make a connection. Using the above example, a firewall could only allow connections from process D if it both recognizes processes A, B and D, and it was configured to allow connections from D while it has causal relationships from A and B. If, for instance, D was a web browser and A and B were recognized operating system components, then connections would be allowed. On the other hand, if A was not recognized by the firewall, then all connections from B would be rejected.

Unfortunately, this form of IPC tracking cannot detect all forms of interaction between processes. For example, the `inetd` program on Unix-based systems receives connections from remote processes and then executes arbitrary commands to handle them. There is no way to detect this interaction when the connection is opened, since the process interaction hasn't occurred yet. Programs may be able to use covert channels, such as those described by Lampson [Lam73] to pass commands in manners undetectable to this algorithm. This algorithm also can't detect any communication that is persistent across reboots, reads from files that were written to by processes that have timed out of the graph, or anything that involves processes on more than one computer. However, it can be used to track most common forms of inter-process communication and will be effective against most attempts to use another program as a network client.

5.4 Implementing an Application Firewall

A full implementation of the application-firewalling techniques presented in this chapter is a topic for further work, but I would suggest a design similar to the following:

- Whenever an application or library is loaded by the operating system, its SHA-1 hash must be computed and stored. On Linux, this might be done by patching the ELF loader in the kernel. It is possible, although not necessary, to extend this functionality into an integrity shell-type system if desired.

- Whenever an application writes to any device that could be used for inter-process communication, a graph similar to the one described in Section 5.3.3 must be updated. This could be accomplished by patching or hooking into kernel system call handlers. A sensible mechanism must be designed for dropping information concerning writes to persistent objects, such as files, after some amount of time must be designed to prevent excessive memory consumption. Ideally, this mechanism should be configurable by system administrators; the `sysctl()` interface on Linux and 4.4BSD-derived operating systems would work well for this.

- When an application makes a `connect()` system call, the host operating system must gather the names and hashes of all processes and libraries in the process group, plus a hash of the source and destination addresses and port numbers, and send this to the firewall. This would be best implemented by patching or hooking into the `connect()` system call handler. When a firewall receives such a message, it must cache it until the connection request arrives.

- When an application makes a `listen()` system call, the host operating system must update a record that maps that application to the port that it is using. This record should be designed so that looking up an application, plus its hashes and group, is a fast operation when given a port number. This would be best implemented by patching or hooking into the `listen()` system call handler.

- When a firewall receives an outbound connection request from an inside host (or from itself), it must check its cache for pre-approval messages. If a appropriate pre-approval message has been received, then the names and hashes of the originating process, all linked libraries, and all processes in its group must be checked. If everything is permitted by site configuration, then the connection request is allowed to proceed. If not, then the failure, and the reason for failure, must be logged and the connection request refused. Similarly, if no appropriate pre-approval message has been received, the connection must be refused. Pre-approval messages must expire and be discarded after some reasonable amount of time; I suggest no more than 500 ms. This functionality could be implemented on Linux in a firewall match function. Due to the complexity of the configuration required (which may involve database lookups), the bulk of this functionality should be implemented in userspace.

- When a firewall receives an inbound communication request from an outside host, it must query the destination of the communication request (which could be itself) to determine what will receive it. When a host receives such a request, it must look up the process that is listening on the appropriate destination port and respond with the names and hashes of the process, all linked libraries, and all processes in its group. When the firewall receives this response, it must verify that the process is allowed to receive connections under the local configuration, and either accept or deny and log the connection attempt. This design requires that the receiver be already waiting for the connection when the firewall receives it, but this should not be a problem in practice. Like the firewall checks on outbound connections, this functionality would be best implemented with a custom firewall match that communicates with a userspace program.

This design uses pre-approval for outbound connections, but call-backs for inbound. My reasoning is that a reasonable bound can be placed on the time interval between a pre-approval message for an outbound connection and the connection request, while no reasonable bound can be placed on the interval between a pre-approval message for an inbound connection and the arrival of a connection request. Also, firewalls have no practical way of determining when processes have stopped listening for new connections after receiving pre-approval messages, and periodic re-issuance of pre-approval messages is wasteful. However, pre-approval is faster and requires fewer messages, so it makes sense to use it where practical.

The system calls mentioned above are appropriate for TCP communication; for other transport protocols, such as UDP, it may be necessary to patch other system calls as well, such as `sendto()`, `sendmsg()`, `recvfrom()`, and `recvmesg()`.

This design is not perfect; for instance, untrusted programs can still access network resources through virtual machines or interpreters, and malicious operating systems can lie to the firewalling system. However, it should be more robust than existing application firewalls.

Chapter 6

Conclusions and Future Work

The most secure computer in the world is one not connected to the Internet. That's why I recommend Telstra ADSL.

– FreeFrag, `http://www.bash.org/?168859`

No firewall is secure against all threats. As Marcus Ranum likes to say, the only completely secure firewall is a pair of wire cutters [Ran]. However, improvements to current firewalling technology are possible. User and application filtering has much potential for strengthening firewall defenses, as they provide good indications of what information is being exchanged without requiring that network traffic be parsed and validated.

6.1 Contributions of this Thesis

In this thesis, I have presented many improvements to the state of the art in user and application filtering.

In Chapter 4, I suggested many improvements to the port knocking and SPA designs described in Chapter 3. First, I examined the possibility of using challenge-response authentication in conjunction with port knocking or SPA. I argued that such a design provides a stronger form of authentication than existing designs, and when used properly, does not result in any loss of stealth compared to other designs. I then suggested a variation on an existing challenge-response authentication algorithm that is suitable for port knocking and SPA, can prevent Mafia frauds under certain circumstances, and corrects problems in the similar algorithm presented in my earlier work [dAJ05].

Secondly, I analysed several methods of making port knocking resistant to out-of-order packet delivery. I discussed various methods of attaching sequence numbers to port knocking packets and of using inter-packet delays to decrease the chance of out-of-order delivery; by comparing the estimated performance of sequence number methods with experimental data showing the frequency of packet loss, I concluded that using sequence numbers is the superior method, although it requires that servers monitor larger numbers of ports.

Next, I suggested two alternate methods for encoding data into port knock sequences. The first, permutation knocking, was proved to be suboptimal compared to the standard encoding of splitting information into n-bit pieces and converting each to a port number, but the second method, bit knocking, shows promise for port knocking to small port ranges while not being vulnerable to out-of-order delivery. Port knocking methods employing sequence numbers are faster, but use more ports; bit knocking is appropriate for environments with limited numbers of ports available for port knocking systems.

Finally, I discussed several methods for preventing race attacks against port knocking and SPA servers. The only one that can be used without significant modifications to existing operating systems and client software is for servers to generate a random port number after successful authentication and have the client connect to it. A stronger method (for TCP connections only) is for the client and server to pre-approve the TCP ISNs to be used in the subsequent connection during the authentication exchange, but the operating system modifications required for this method limit its practicality. All of these methods require that the server return information to the client during authentication; it could be attached to challenge messages issued in challenge-response authentication.

By using these methods, both port knocking and SPA can be made both more secure and more reliable. Performance estimates show that even with the larger messages required by my systems than earlier designs, authentication is still fast, even on slow networks.

Chapter 5 described problems with and presented improvements to existing application firewalling systems. Many modern application firewalls have a number of problems. Application firewalls cannot safely identify trusted programs by name alone; they must also verify executable files and libraries, but this is easier said than done. Since the program that opens a socket is not necessarily the one that is sending or receiving information, application firewalls cannot always reliably identify the origins of network requests either; malicious or untrusted software can take advantage of this to masquerade as trusted software. In particular, application firewalls cannot identify programs that are running within virtual machines or interpreters. Finally, host firewalling software, including most application firewalls, can be disabled or bypassed by malware.

To correct some of these issues, I suggested that application firewalling could also be done by network firewalls. This requires that information about users and programs be sent to network firewalls by hosts; this could be accomplished using either call-back or pre-approval strategies. Software that is directly loaded by the operating system can be reliably identified by checking cryptographic hashes computed at load-time; but this cannot easily be extended to software that runs inside virtual machines or interpreters. Finally, most instances of processes making network connections on behalf of others can be detected by building a directed graph of inter-process communication events that shows causal relationships between processes. An application firewalling system with these improvements will be more robust and reliable than existing systems for which documentation is available, although avenues for attack and abuse still exist.

6.2 Opportunities for Future Work

The designs and algorithms presented in this thesis are not perfect; many improvements to them could be made. I have identified a number of opportunities for further work on both covert authentication systems and application firewalls.

Further Research in Port Knocking and SPA

- All of the port knocking designs presented here (except Barham et al.'s SSTCP) will fail if any packets are lost in transit. This can be corrected for disorder-resistant port knocking schemes by repeatedly re-sending the entire sequence until at least one copy of each packet is successfully delivered, or for non-disorder-resistant schemes by repeatedly re-sending the entire sequence until a complete sequence is successfully delivered. However, this may be unnecessarily inefficient. A better method would be to encode error-correcting information into port knock sequences so that they could recover from limited packet loss; this would require experimental analysis to determine the degree to which port knocking is affected by packet loss.

- The two methods presented in Chapter 3 for differentiating between port scans and port knocking are not fool-proof; it is still possible to disguise port knocking as port scans. I do not believe that it is possible to unambiguously differentiate between the two in the general case, but I have no proof of this. Further research is required to determine if it is possible to differentiate unambiguously (or at least with high probability) between port scans and port knocking and to design an algorithm to do so if it is possible.

- The authentication algorithms presented in Section 4.1 are vulnerable to mafia frauds in the case where the client doesn't know its public IP address (which may happen if it is NATed). Further investigation may reveal a method to prevent mafia frauds that does not rely on public IP addresses.

- Port knocking and SPA were presented in Chapters 3 and 4 using only symmetric-key techniques. Either could also be implemented using public-key algorithms, but the particulars of such designs have not been investigated. The large key sizes required for RSA may be impractical for port knocking, but elliptic curve algorithms should be feasible.

- Traditionally, port knocking systems encode information into port numbers on only a single destination host. This could be extended to multiple hosts: *distributed port knocking* would encode information into sets of ports on more than one destination host. Such a system would require some sort of distributed server that aggregates information from participating hosts before making decisions but has the potential to significantly enlarge available port ranges and thus the amount of information that can be encoded into each port number, as well as making port knocking more difficult to detect.

Further Research in Application Firewalling

- The application firewalling techniques described in Chapter 5 have not been implemented and tested. While I believe them to be practical, I have no figures on the overhead imposed by such a system or the effect of this overhead on network communications. A complete implementation would require significant amounts of work, but would be necessary for a thorough analysis of the efficiency and effectiveness of such an application firewalling system.

- In Section 5.3.2, I identified several ways that my techniques for identifying legitimate applications could be fooled or bypassed using virtual machines, interpreters, or malicious operating systems. I do not believe that these can be solved without making unrealistic assumptions about the trustworthiness of software, but I have no proof of this.

- The process graph mechanism described in Section 5.3.3 should detect causal relationships between process established via communication over normal IPC channels, but does not take into account inter-process covert channels [Lam73]. It also does not take into account programs like `inetd` that accept connections and then execute arbitrary programs to handle them. It may be possible to devise another mechanism that can more accurately identify connections by proxy.

Even without these enhancements, application firewalls and covert authentication systems such as port knocking and SPA, as presented in this thesis, are useful tools that can significantly enhance existing firewall systems by providing them with additional information about who and what is attempting to communicate through them.

Bibliography

[0ld02] 0ldW0lf. TocToc Version 1.7.27 - Linux Backdoor. `http://www.atrix-team.org/toctoc/TocToc-1.7.27.tgz`, 2002. [Accessed 28 Nov. 2006].

[AC75] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[Ahs02] Kamran Ahsan. Covert Channel Analysis and Data Hiding in TCP/IP. Master's thesis, Dept. of Electrical and Computer Engineering, University of Toronto, 2002.

[Alb04] Conan C. Albrecht. How Clean is the Future of SOAP? *Communications of the ACM*, 47(2):66–68, February 2004.

[ALHF06] Pablo Neira Ayuso, Eric Leblond, J. Federico Hernandez, and Luis Floreani. Re: new match extension to implement port knocking in one. Postings to netfilter-devel@lists.netfilter.org; archived at `http://lists.netfilter.org/pipermail/netfilter-devel/2006-October/`, October 2006. [Accessed 17 Oct. 2006].

[And01] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems.* John Wiley & Sons Inc, January 2001.

[And06] Oskar Andreasson. Iptables Tutorial 1.2.2. `http://iptables-tutorial.frozentux.net/`, November 2006.

[AS02] Ammar Alkassar and Christian Stüble. Towards Secure IFF: Preventing Mafia Fraud Attacks. In *Proc. MILCOM 2002*, pages 1139–1144, October 2002.

[Ayc06] John Aycock. *Computer Viruses and Malware*, volume 22 of *Advances in Information Security*. Springer, 2006.

[BBB$^+$06] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for Key Management — Part 1: General (Revised). NIST Special Publication 800-57, May 2006.

[BBD$^+$91] Samy Bengio, Gilles Brassard, Yvo G. Desmedt, Claude Goutier, and Jean-Jacques Quisquater. Secure Implementation of Identification Systems. *Journal of Cryptology*, 4(3):175–183, January 1991.

[BBO05] Jacob Babbin, Simon Biles, and Angela D. Orebaugh. *Snort Cookbook*. O'Reilly, 2005.

[BBS86] Lenore Blum, Manuel Blum, and Michael Shub. A simple unpredictable pseudo random number generator. *SIAM Journal on Computing*, 15(2):364–383, May 1986.

[BC94] Steven M. Bellovin and William R. Cheswick. Network Firewalls. *IEEE Communications Magazine*, 32(9):50–57, September 1994.

[BD90] Thomas Beth and Yvo Desmedt. Identification tokens – or: Solving the chess grandmaster problem. In *Advances in Cryptology – Proc. CRYPTO '90*, LNCS 537, pages 169–176, August 1990.

[BDS03] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error

Exploits. In *Proc. 12th USENIX Security Symposium*, pages 105–120, August 2003.

[Bea00] Jay Beale. "'Security Through Obscurity" Ain't What They Think It Is. `http://www.bastille-linux.org/jay/obscurity-revisited.html`, 2000. [Accessed 1 June 2005].

[Bel89] S. M. Bellovin. Security problems in the TCP/IP protocol suite. *ACM SIGCOMM Computer Communications Review*, 19(2):32–48, 1989.

[Bel99] Steven M. Bellovin. Distributed Firewalls. *USENIX ;login: Magazine*, 24(Special Issue on Security):39–47, November 1999.

[Ber92] Daniel J. Bernstein. TAP. IETF Internet Draft, archived at `http://www.watersprings.org/pub/id/draft-bernstein-tap-00.txt`, May 1992.

[BHI$^+$02] Paul Barham, Steven Hand, Rebecca Isaacs, Paul Jardetzky, Richard Mortier, and Timorhy Roscoe. Techniques for Lightweight Concealment and Authentication in IP Networks. Technical Report IRB-TR-02-009, Intel Research, July 2002.

[BP04] Kevin Borders and Atul Prakash. Web Tap: Detecting Covert Web Traffic. In *Proc. 11th ACM Concerence on Computer and Communications Security*, pages 110–120, October 2004.

[BPS99] Jon C. R. Bennett, Craig Partridge, and Nicholas Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Transactions on Networking*, 7(6):789–798, December 1999.

[BR] Renaud Bidou and Frédéric Raynal. `http://www.iv2-technologies.com/~rbidou/CovertChannels.pdf`. [Accessed 17 Nov. 2006].

[Bra] Tony Bradley. Port Knocking: Knowing the secret can knock open your system. `http://netsecurity.about.com/cs/generalsecurity/a/aa032004.htm`. [Accessed 17 Nov. 2006].

[CBR03] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 2nd edition, February 2003.

[Cer00] Certicom Research. Standards for efficient cryptography, SEC 1: Elliptic Curve Cryptography. `http://www.secg.org/download/aid-385/sec1_final.pdf`, September 2000. [Accessed 16 Feb. 2007].

[CK04] Cappella and Tan Chew Keong. Remote Server Management With One-Time Port Knocking (OTPK). `http://www.security.org.sg/code/portknock2.html`, June 2004. [Accessed 10 Mar. 2006].

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill/MIT Press, 2nd edition, 2001.

[Coh89] Fred Cohen. Models of Practical Defense Against Computer Viruses. *Computers & Security*, 8(2):149–160, April 1989.

[dAJ05] Rennie deGraaf, John Aycock, and Michael J. Jacobson, Jr. Improved Port Knocking with Strong Authentication. In *Proc. 21st Annual Computer Security Applications Conference (ACSAC 2005)*, pages 409–418, December 2005.

[DDN91] Danny Dolev, Cynthia Dwork, and Moni Naor. Non-Malleable Cryptography. In *Proc. 23rd Annual ACM Symposium on Theory of Computing*, pages 542–552, January 1991.

[DDW99] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(9):805–822, April 1999.

[Deg06] Loris Degioanni. WinPcap: The Windows Packet Capture Library. `http://www.winpcap.org/`, October 2006.]Accessed 15 Nov. 2006].

[Des88] Yvo Desmedt. Major security problems with the "unforgeable" (Fiege-)Fiat-Shamir proofs of identity and how to overcome them. In *Proc. Securicom 88: 6ème Congrès mondial de la protection et de la sécurité informatique et des communication*, pages 147–159, March 1988.

[DGB87] Yvo Desmedt, Claude Goutier, and Samy Bengio. Special Uses and Abuses of the Fiat-Shamir Passport Protocol (extended abstract). In *Advances in Cryptology – Proc. CRYPTO '87*, LNCS 293, pages 21–39, August 1987.

[DH76] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[DH98] S. Deering and R. Hinden. RFC 2460: Internet Protocol, Version 6 (IPv6) Specification, December 1998.

[Doy04] Matt Doyle. Implementing a Port Knocking System in C. Physics honors thesis, University of Arkansas, 2004.

[DR06] T. Dierks and E. Rescorla. RFC 4346: The Transport Layer Security (TLS) Protocol, Version 1.1), April 2006.

[dR07] Theo de Raadt. Exploit Mitigation Techniques. Talk slides, AUUG 2004; available at `http://www.auug.org.au/events/2004/auug2004/theo/`, September 2007. [Accessed 27 Feb. 2007].

[dri96] daemon9, route, and infinity. Project Neptune. *Phrack Magazine*, 7(48), July 1996.

[dS06] Bruno Castro da Silva. TuxGuardian – An application-based firewall. `http://tuxguardian.sourceforge.net/`, April 2006. [Accessed 20 Feb. 2007].

[Dub03] Ido Dubrawsky. Firewall Evolution – Deep Packet Inspection. `http://www.securityfocus.com/infocus/1716`, July 2003. [Accessed 28 Feb. 2007].

[dVCIdV99] Marco de Vivo, Eddy Carrasco, Germinal Isern, and Gabriela O. de Vivo. A Review of Port Scanning Techniques. *ACM SIGCOMM Computer Communications Review*, 29(2):41–48, April 1999.

[dVdVI98] Marco de Vivo, Gabriela O. de Vivo, and Germinal Isern. Internet Security Attacks at the Basic Levels. *ACM SIGOPS Operating Systems Review*, 32(2):4–15, April 1998.

[DvOW92] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992.

[Epp04] Dana Epp. Introduction to Cerberus. Talk slides, Fraser Valley Linux Users' Group; available at `http://silverstr.ufies.org/blog/Cerberus.ppt`, June 2004. [Accessed 3 Dec. 2006].

[FLH$^+$00] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. RFC 2784: Generic Routing Encapsulation (GRE), March 2000.

[Fre60] Edward Fredkin. Trie Memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[FX 00] FX of Phenoelit. cdoor.c: packet coded backdoor. `http://www.phenoelit.de/stuff/cd00r.c`, June 2000. [Accessed 15 Nov. 2006].

[Fyo97] Fyodor. The Art of Port Scanning. *Phrack Magazine*, 7(51), September 1997.

[Fyo06] Fyodor. `man` page for nmap 4.03. `http://download.insecure.org/nmap/dist/nmap-4.03.tar.bz2`, April 2006. [Accessed 29 Nov. 2006].

[GC04] John Graham-Cumming. tumbler. `http://tumbler.sourceforge.net/`, 2004. [Accessed 3 Dec. 2006].

[Gee05] David Geer. Malicious bots threaten network security. *IEEE Computer*, 38(1):18–20, January 2005.

[Gib05] Steve Gibson. The Strange Tale of the Denial of Service Attacks Against GRC.COM. `http://www.grc.com/dos/grcdos.htm`, September 2005. [Accessed 25 Jan. 2007].

[Gir87] C. Gray Girling. Covert Channels in LAN's. *IEEE Transactions on Software Engineering*, SE-13(2):292–296, February 1987.

[GPL04] Ladan Gharai, Colin Perkins, and Tom Lehman. Packet Reordering, High Speed Networks and Transport Protocol Performance. In *Proc. 13th International Conference on Computers and Networks (ICCCN 2004)*, pages 73–78, October 2004.

[Gre05] Paul Gregoire. jPortKnock. `http://www.gregoire.org/code/jportknock/`, January 2005. [No longer available as of 2007/01/16; archived at `http://web.archive.org/`].

[Hal94] Neil M. Haller. The S/KEY one-time password system. In *Proceedings of the Symposium on Network and Distributed System Security*, pages 151–157, 1994.

[Har02] Daniel Hartmeier. Design and Performance of the OpenBSD Stateful Packet Filter (pf). In *Proc. 2002 USENIX Annual Technical Conference, FREENIX Track*, pages 171–180, June 2002.

[HB06] Greg Hoglund and James Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2006.

[HM05] Christoph Hellwig and Patrick McHardy. tasklist_lock abuse in ipt{,6}owner. Postings to netfilter-devel@lists.netfilter.org; archived at `http://lists.netfilter.org/pipermail/netfilter-devel/2005-August/`, August 2005. [Accessed 20 Feb. 2007].

[IKBS00] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a Distributed Firewall. In *Proc. 7th ACM Conference on Computer and Communication Security*, pages 190–199, 2000.

[ISO95] ISO/IEC. Information technology – Security techniques – Entity authentication – Part 4: Mechanisms using a cryptographic check function. ISO/IEC std. 9798-4, International Organization for Standardization, Geneva (Switzerland), 1995.

[IT91] ITU-T. Security architecture for open systems interconnection for ccitt applications. ITU Reccomendation X.800, International Telecommunication Union, March 1991.

[Jea06] Sebastien Jeanquier. An Analysis of Port Knocking and Single Packet Authentication. Master's thesis, Royal Holloway College, University of London, September 2006.

[JIDT02] Sharad Jaiswal, Gianluca Iannaconne, Christophe Diot, and Don Towsley. Measurement and Classifiction of Out-of-Sequence Packets in a Tier-1 IP Backbone. In *Proc. 2nd ACM SIGCOMM Workshop on Internet Measurement*, pages 113–114, November 2002.

[JLM06] Van Jacobson, Craig Leres, and Steven McCanne. tcpdump/libpcap. `http://www.tcpdump.org/`, August 2006. [Accessed 15 Nov. 2006].

[Joh93] M. St. Johns. Identification Protocol, February 1993.

[JPBB04] Jaeyeon Jung, Vern Paxson, Arthur W. Berger, and Hari Balakrishnam. Fast Portscan Detection Using Sequential Hypothesis Testing. In *Proc. 2004 IEEE Symposium on Security and Privacy*, pages 211–225, May 2004.

[Ken97] Malachi Kenney. Ping of Death. `http://insecure.org/sploits/ping-o-death.html`, January 1997. [Accessed 25 Jan. 2007].

[Ker83] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX:5–38, January 1883.

[KHM06] Joel Knight, Nick Holland, and Steven Mestdagh. PF: The OpenBSD Packet Filter. `http://www.openbsd.org/faq/pf/index.html`, November 2006. [Accessed 11 Feb. 2007].

[KK92] David Koblas and Michelle R. Koblas. SOCKS. In *Proc. USENIX Security III Symposium*, pages 77–83, September 1992.

[Kli06] Vlastimil Klima. Tunnels in Hash Functions: MD5 Collisions Within a Minute. Technical Report 2006/105, Cryptology ePrint Archive, March 2006.

[KP05] Jzsef Kadlecsik and Gyrgy Psztor. Netfilter Performance Testing. `http://people.netfilter.org/kadlec/nftest.pdf`, May 2005. [Accessed 28 Nov. 2006].

[Kri04] Stuart Krivis. Port Knocking: Helpful or Harmful?: An Exploration of Modern Network Threats. `http://www.giac.org/practical/GSEC/Stuart_Krivis_GSEC.pdf`, May 2004. [Accessed 28 Nov. 2004].

[Krz03] Martin Krzywinski. Port Knocking. `http://www.linuxjournal.com/article/6811`, June 2003. [Accessed 12 Nov. 2004].

[Krz06] Martin Krzywinski. PORTKNOCKING - A system for stealthy authentication across closed ports. `http://www.portknocking.org/`, May 2006. [Accessed 29 Nov. 2006].

[KS05] S. Kent and K. Seo. RFC 4301: Security Architecture for the Internet Protocol, December 2005.

[KVV05] Christopher Kruegel, Fredrik Valeur, and Giovanni Vigna. *Intrusion Detection and Correlation: Challenges and Solutions*, volume 14 of *Advances in Information Security*. Springer-Verlag, 2005.

[Lam73] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[Lev00] John R. Levine. *Linkers & Loaders*. Morgan Kaufmann Publishers, 2000.

[LFJ+86] Samuel J. Leffler, Robert S. Fabry, William N. Joy, Phil Lapsley, Steve Miller, and Chris Torek. An Advanced 4.3BSD Interprocess Communication Tutorial. Computer Systems Research Group, Deptartment of Electrical Engineering and Computer Science, University of California, Berkeley, 1986.

[LGL+96] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. RFC 1928: SOCKS Protocol Version 5, March 1996.

[LK02] C. Leckie and R. Kotagiri. A Probabilistic Approach to Detecting Network Scans. In *Proc. 8th IEEE Network Operations and Management Symposium (NOMS 2002)*, pages 359–372, April 2002.

[LRST00] Felix Lau, Stuart H. Rubin, Michael H. Smith, and Ljiljana Trajkovic. Distributed Denial of Service Attacks. In *Proc. 2000 IEEE International Conference on Systems, Man and Cypernetics*, pages 2275–2280, October 2000.

[LUR03] LURHQ Threat Intelligence Group. Windows Messenger Popup Spam on UDP Port 1026. `http://www.lurhq.com/popup_spam.html`, June

2003. [Accessed 5 Dec. 2006].

[Mad04] Ben Maddock. Port Knocking: An Overview of Concepts, Issues and Implementations. `http://www.giac.org/practical/GSEC/Ben_Maddock_GSEC.pdf`, September 2004. [Accessed 25 Jan. 2005].

[MD90] J. Mogul and S. Deering. RFC 1191: Path MTU Discovery, November 1990.

[Mic06] Microsoft Corp. A description of Svchost.exe in Windows XP. `http://support.microsoft.com/?kbid=314056`, April 2006. [Accessed 12 Feb. 2007].

[MJ] William Metcalf and Victor Julien. snort_inline. `http://snort-inline.sourceforge.net/`. [Accessed 2 Mar. 2007].

[MMB05] Chris Muelder, Kwan-Liu Ma, and Tony Bartoletti. Interactive Visualization for Network and Port Scan Detection. In *Proc. Recent Advances in Intrusion Detection (RAID 2005)*, LNCS 3858, pages 265–283, September 2005.

[MMETC05] Antonio Izquierdo Manzanares, Joaquín Torres Márquez, Juan M. Estévez-Tapiador, and Julio César Hernández Castro. Attacks on Port Knocking Authentication Mechanism. In *Proc. 2005 International Conference on Computational Science and its Applications*, LNCS 3483, pages 1292–1300, May 2005.

[Mor85] Robert T. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software. Computing Science Technical Report 117, AT&T Bell Laboratories, Murray Hill, New Jersey, United States, February 1985.

[MPS$^+$03] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer Worm. *IEEE Security & Privacy*, 1(4):33–39, July 2003.

[MSB$^+$06] David Moore, Colleed Shannon, Douglas J. Brown, Geoffery M. Voelker, and Stefan Savage. Inferring Internet Denial-of-Service Activity. *ACM Transactions on Computer Systems*, 24(2):115–139, May 2006.

[MSVS03] David Moore, Colleen Shannon, Geoffery M. Voelker, and Stefan Savage. Internet quarantine: requirements for containing self-propagating code. In *Proc. 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, pages 1901–1910, March 2003.

[Mul05] Mike Mullins. Be aware of potential threats from port knocking. `http://articles.techrepublic.com.com/5102-1009-5770469.html`, June 2005. [Accessed 17 Nov. 2006].

[MvOV96] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, October 1996.

[Nar04] Arvind Narayanan. A critique of port knocking. `http://software.newsforge.com/article.pl?sid=04/08/02/1954253`, August 2004. [Accessed 25 Jan. 2006].

[Nor83] Donald A. Norman. Design Rules Based on Analyses of Human Error. *Communications of the ACM*, 26(4):254–258, April 1983.

[Pax99] Vern Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.

[PN98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. http://insecure.org/stf/secnet_ids/secnet_ids.html, January 1998. [Accessed 15 Feb. 2007].

[Pos80] J. Postel. RFC 768: User Datagram Protocol, August 1980.

[Pos81a] J. Postel. RFC 793: Transmission Control Protocol, September 1981.

[Pos81b] Jon Postel. RFC 791: Internet Control Message Protocol: DARPA Internet Program Protocol Specification, September 1981.

[Pos81c] Jon Postel. RFC 791: Internet Protocol: DARPA Internet Program Protocol Specification, September 1981.

[PYB+04] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of Internet Background Radiation. In *Proc. 4th ACM SIGCOMM Conference on Internet Measurement*, pages 27–40, October 2004.

[Ran] Marcus J. Ranum. The Ultimately Secure Deep Packet Inspection and Application Security System. http://www.ranum.com/security/computer_security/papers/a1-firewall/index.html. [Accessed 7 Mar. 2007].

[Ran05] Marcus J. Ranum. What is "Deep Inspection"? http://www.ranum.com/security/computer_security/editorials/deepinspect/, May 2005. [Accessed 2 Mar. 2007].

[Ras04] Michael Rash. Combining port knocking and passive OS fingerprinting with fwknop. *USENIX ;login: Magazine*, 29(6):19–25, December 2004.

[Ras06] Michael Rash. Single Packet Authorization with Fwknop. *USENIX ;login: Magazine*, 31(1):63–69, February 2006.

[Res95] Research and Development in Advanced Communiations Technologies in Europe (RACE). *Integrity Primitives for Secure Information Systems: Final Report of RACE Integrity Primitives Evaluation (R1040)*, chapter 6, pages 169–178. LNCS 1007. Springer-Verlag, 1995.

[RMK$^+$96] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. RFC 1918: Address Allocation for Private Internets, February 1996.

[Rus00] Rusty Russell. Linux IP Firewalling Chains. `http://people.netfilter.org/~rusty/ipchains/`, October 2000. [Accessed 30 Nov. 2006].

[Sch05] Bruce Schneier. New Cryptanalytic Results Against SHA-1. `http://www.schneier.com/blog/archives/2005/08/new_cryptanalyt.html`, August 2005. [Accessed 8 Feb. 2007].

[Sch06a] J urgen Schmidt. How Skype & Co. get round firewalls. `http://www.heise-security.co.uk/articles/82481`, December 2006. [Accessed 20 Jan. 2007].

[Sch06b] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., 2nd edition, 2006.

[SE01] P. Srisuresh and K. Egevang. RFC 3022: Traditional IP Network Address Translator (Traditional NAT), January 2001.

[Sel07] Peter Selinger. The GLIBC random number generator. `http://www.`

`mscs.dal.ca/~selinger/random/`, January 2007. [Accessed 6 Feb. 2007].

[SF06] J. Federico Hernandez Scarso and Luis A. Floreani. PortKnockO Project. `http://portknocko.berlios.de/`, October 2006. [Accessed 29 Nov. 2006].

[SH99] P. Srisuresh and M. Holdrege. RFC 2663: IP Network Address Translator (NAT) Terminology and Considerations, August 1999.

[SHM02] Stuart Staniford, James A. Hoagland, and Joseph M. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10(1-2):105–136, 2002.

[SMPW04] Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The Top Speed of Flash Worms. In *Poc. 2004 Workshop on Rapid Malcode (WORM04)*, pages 33–42, October 2004.

[Sol98] Solar Designer. Designing and Attacking Port Scan Detection Tools. *Phrack Magazine*, 8(53), July 1998.

[Spa89] Eugene H. Spafford. The Internet Worm Program: An Analysis. *ACM SIGCOMM Computer Communication Review*, 19(1):17–57, January 1989.

[SPW02] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in Your Spare Time. In *Proc. 11th USENIX Security Symposium*, pages 149–167, August 2002.

[Sta03] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 3rd edition, 2003.

[Sym04] Symantec Corp. News Release: Symantec Internet Security Threat Report Identifies More Attacks Now Targeting e-Commerce, Web Applications. `http://www.symantec.com/press/2004/n040920b.html`, September 2004. [Accessed 25 Jan. 2005].

[Sym07a] Symantec Corp. Backdoor.Fuwudoor. `http://www.symantec.com/avcenter/venc/data/backdoor.fuwudoor.html`, January 2007. [Accessed 21 Feb. 2007].

[Sym07b] Symantec Corp. W32.Erkez.B@mm. `http://www.symantec.com/security_response/writeup.jsp?docid=2004-061110-4018-99`, January 2007. [Accessed 21 Feb. 2007].

[Sym07c] Symantec Corp. W32.Spybot.ACYR. `http://www.symantec.com/avcenter/venc/data/w32.spybot.acyr.html`, January 2007. [Accessed 21 Feb. 2007].

[Sym07d] Symantec Corp. W32.Welchia.worm. `http://www.symantec.com/avcenter/venc/data/w32.welchia.worm.html`, January 2007. [Accessed 21 Feb. 2007].

[SZ04] Ed Skoudis and Larry Zeltser. *Malware: Fighting Malicious Code*. PTR Series in Computer Networking and Distributed Systems. Prentice Hall, 2004.

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, 3rd edition, 1996.

[TC04] Chew Keong Tan and Cappella. Remote Server Management using Dynamic Port Knocking and Forwarding. `http://www.security.org.sg/`

code/sig2portknock.pdf, May 2004. [Accessed 25 Jan. 2005].

[The01] The Honeynet Project. Know Your Enemy: Statistics. http://project.honeynet.org/papers/stats/, July 2001. [Accessed 1 Dec. 2006].

[the05] the PaX Team. Documentation for the PaX project. http://pax.grsecurity.net/docs/index.html, October 2005. [Accessed 23 Feb. 2007].

[UC06a] US-CERT. GnuPG vulnerable to remote data control. Vulnerability Note VU#427009, December 2006.

[UC06b] US-CERT. OpenSSH fails to properly handle multiple identical blocks in a SSH packet. Vulnerability Note VU#787448, September 2006.

[UC07a] US-CERT. Cisco Secure Access Control Server vulnerable to a stack-based buffer overflow via a specially crafted "HTTP GET" request. Vulnerability Note VU#744249, January 2007.

[UC07b] US-CERT. MIT Kerberos Vulnerabilities. Technical Cyber Security Alert TA07-009B, January 2007.

[Ven92] Wietse Venema. TCP WRAPPER: Network Monitoring, access control, and booby traps. In *Proc. USENIX Security III Symposium*, pages 85–92, September 1992.

[Vij04] Jaikumar Vijayan. E-biz Sites Hit With Targeted Attacks, Extortion Threats. http://www.computerworld.com/securitytopics/security/story/0,10801,96183,00.html, September 2004. [Accessed 25 Jan. 2007].

[VMw07] VMware, Inc. Vmware. http://www.vmware.com/, 2007. [Accessed 25 Feb. 2007].

[vOS06] Paul C. van Oorschot and Stuart Stubblebine. On countering online dictionary attacks with login histories and humans-in-the-loop. *ACM Transactions on Information and System Security*, 9(3):235–258, August 2006.

[Wal04] Joe Walko. Cryptknock. http://cryptknock.sourceforge.net/, June 2004. [Accessed 25 May 2005].

[War05] Bruce Ward. The Doorman. http://doorman.sourceforge.net/, September 2005. [Accessed 3 Dec. 2006].

[WCM+02] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proc. 11th USENIX Security Symposium*, pages 17–31, August 2002.

[Wel06a] Harald Welte. Netfilter: firewalling, NAT, and packet mangling for Linux. http://www.netfilter.org/, 2006. [Accessed 15 Nov. 2006].

[Wel06b] Harald Welte. The netfilter.org "libnetfilter_queue" project. http://www.netfilter.org/projects/libnetfilter_queue/, 2006. [Accessed 30 Nov. 2006].

[WFLY04] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. Technical Report 2004/119, Cryptology ePrint Archive, August 2004.

[Whi01] Aoife White. New Trojan disables firewall defenses. `http://www.itweek.co.uk/2057738`, May 2001. [Accessed 23 Mar. 2006].

[wik07] wikipedia.org. Slashdot effect. `http://en.wikipedia.org/wiki/Slashdot_effect`, January 2007. [Accessed 25 Jan. 2007].

[Wil02] Matthew M. Williamson. Throttling Viruses: Restricting propagation to defeat malicious mobile code. In *Proc. 18th Annual Computer Security Applications Conference (ACSAC 2002)*, pages 61–68, December 2002.

[Wol89] Manfred Wolf. Covert Channels in LAN Protocols. In *Proc. Workshop on Local Area Network Security (LANSEC '98)*, LNCS 396, pages 91–101, April 1989.

[Wor04] David Worth. CÖK: Cryptographic One-Time Knocking. Talk slides, Black Hat USA, 2004.

[WvOS05] Glenn Wurster, P. C. van Oorschot, and Anit Somayaji. A generic attack on checksumming-based software tamper resistance. In *Proc. 2005 IEEE Symposium on Security and Privacy*, pages 127–138, May 2005.

[WYY05] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In *Advances in Cryptology – Proc. CRYPTO 2005*, LNCS 3621, pages 17–36, August 2005.

[YBU03] Vinod Yegneswaran, Paul Barford, and Johannes Ullrich. Internet Intrusions: Global Characteristics and Prevalence. In *Proc. 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 138–147, June 2003.

[Yon06] James Yonan. OpenVPN. `http://openvpn.net/`, October 2006. [Accessed 13 Feb. 2007].

[Zon07] Zone Labs, LLC. User Guide for ZoneAlarm security software, version 7.0. `http://download.zonelabs.com/bin/media/pdf/zaclient70_user_manual.pdf`, 2007. [Accessed 15 Feb. 2007].

Appendix A

An Experiment in Out-of-Order Packet Delivery

Understanding how packets are dropped and re-ordered by networks has implications in designing protocols that are resistant to packet loss and out-of-order delivery. Packets may be dropped by IP routers or endpoints due to local congestion, transmission errors, or routing failures. They may be re-ordered in transit due to a number of factors, including routing shifts, load balancing, parallelism in network components (routers, switches, etc.), and scheduling issues at endpoints [BPS99].

Applications that depend on receiving data in its entirety and in the correct order are generally advised to use TCP or other reliable transport protocols, which guarantee that all transmitted data is passed to the application in the correct order. However, some applications that depend on these factors are forced to use unreliable protocols like UDP because of factors such as real-time constraints or a lack of a return channel for acknowledgements. For such applications, detailed knowledge of the packet-dropping and re-ordering properties of IP networks are critical.

A.1 Related Work

Paxson's pioneering work [Pax99] measured re-ordering among data and ACK packets in long-lasting TCP flows in 1994 and 1995, observing re-order rates of data packets between some hosts of up to 36% [1]. Another key observation is that re-ordering behaviour differs dramatically between hosts and changes dramatically over time: a host whose traffic was 15% re-ordered in one trace was only 0.025% re-ordered in another,

[1]ACK packets, while smaller, had significantly lower re-order rates, due to their frequently being sent farther apart than data packets.

and even overall averages between large traces differed from 0.3% to 2.0%. Bennett et al. [BPS99] observed re-ordering in up to 90% of bursts of 10 to 20 512-byte packets in 1997. Jaiswal et al. [JIDT02] measured re-orderings of only 0.5% of the packets traversing a single backbone link. More recently, Gharai et al. [GPL04] tested high-bandwidth flows of large UDP packets (500 bytes and up) and found re-ordering of no more than 1.65%; however, packets in their experiment traversed only backbone links.

All of these experiments focused on the effects of re-ordering on TCP data flows and delay-sensitive UDP applications such as real-time video streaming. No analysis seems to have been done of the occurrence of out-of-order delivery among small packets sent with low inter-packet delays and without acknowledgement. An experiment that measures out-of-order delivery as a function of packet size and inter-packet delay for UDP traffic is presented in the following sections.

A.2 Experimental Design

The experiment was conducted by sending several bursts of traffic from a central server to several clients ("targets") across the Internet. In each burst, 100 packets of the same size were sent to the same UDP port; the size of the packets and the delay time between sending each were varied between bursts. UDP datagrams with 0, 100, and 500 bytes of payload data (consisting of all zero bytes in each case) and inter-packet delays of 0, 1, 2, 4, 6, 8 and 10 milliseconds were tested. The server waited 10 seconds after each burst to ensure that all packets had been delivered to targets before starting the next burst. The UDP source port was modulated from 1 to 100 to provide sequence numbers. Each set of 24 bursts (one for each combination of packet payload sizes and inter-packet delays) was repeated every half hour for 24 hours. Packets were generated by a C program using the raw socket interface for maximum throughput.

Test	City	Routing hops
Target 1	Calgary, Alberta, Canada	7
Target 2	Grand Rapids, Michigan, United States	16
Target 3	New Westminster, British Columbia, Canada	9
Target 4	Charlotte, North Carolina, United States	20

Table A.1: Locations of targets for the out-of-order delivery experiment

The server was a workstation with a single Intel Pentium 4 3.0 GHz processor running Linux 2.6.17 located at the University of Calgary. Targets were various home computers on residential broadband connections owned by volunteers in several cities in North America, listed in Table A.1. Four tests were done on weekdays in early December 2006.

The UDP destination ports used by each target were chosen such that they should not receive any unrelated traffic for the duration of the experiment. Targets used `tcpdump` to capture and log all packets received at this port, which were all dropped without response. No effort was made to differentiate between packet loss or reordering caused by the server, targets, or network devices in between, since it makes no difference to a receiving application: a lost or re-ordered packet is a lost or re-ordered packet.

Since the experiment depended on people volunteering their time and network bandwidth to be targets, it was designed to interfere with their normal use of their computers as little as possible. For privacy reasons, no information was gathered on other activity on the targets or networks during the experiment. Had dedicated target machines been available, more tests over a longer period of time in a more controlled environment would have been conducted.

A.3 Results

After all four tests had completed, the results were analyzed for packet loss and reordering behaviours. For the purposes of the analysis, the following definitions were used:

- A packet is deemed to have been *dropped* if it did not arrive at the target before the first packet of the next test set.
- A packet is *late* if it arrives after a packet with a higher sequence number. (For example, in the sequence "1, 3, 2, 4", "2" is late.
- A packet is *out-of-order* it its index does not match its arrival position. (For example, in the sequence "1, 3, 2, 4", both "2" and "3" are out-of-order.
- *Error* is the number of positions by which a packet's index differs from its arrival order.
- *Average error* is the mean error of out-of-order packets; in-order packets (with errors of 0) are not included.

Results for the four targets using 0, 1 and 2 ms inter-packet delays are summarized in Tables A.2, A.3 and A.4; loss and re-ordering were below 2% for all other tests..

Behaviours differed significantly between targets: Target 1 observed no dropped packets throughout the course of the experiment, but saw very large numbers of re-ordered packets in the 0 ms tests, whereas Target 2 observed many dropped and few re-ordered packets. If the majority of packet loss or re-ordering events occurred close to the server, then the variance between the results for each target should be small, so this suggests that most packet dropping and re-ordering occurred either on network backbones or near targets. All packets were sent with the same TTL (the Linux

	Payload size	Target 1	Target 2	Target 3	Target 4	Combined
Packets dropped	0 bytes	0%	29.3%	22.0%	29.1%	19.9%
	100 bytes	0%	42.7%	37.0%	5.57%	21.2%
	500 bytes	0%	46.7%	37.6%	3.30%	21.8%
Out-of-order packets	0 bytes	80.4%	15.4%	60.6%	52.9%	52.4%
	100 bytes	31.6%	0.68%	30.8%	66.7%	32.3%
	500 bytes	11.8%	0.74%	16.7%	57.5%	21.5%
Late packets	0 bytes	31.8%	8.26%	28.7%	28.3%	24.3%
	100 bytes	14.0%	0.34%	16.0%	34.9%	16.2%
	500 bytes	3.15%	0.19%	7.72%	27.7%	9.58%
Average error	0 bytes	3.97	18.3	5.05	8.04	6.34
	100 bytes	1.75	1.00	1.77	2.64	2.20
	500 bytes	2.27	11.4	1.43	1.48	1.58
Maximum error	0 bytes	35	94	81	84	84
	100 bytes	25	1	18	82	82
	500 bytes	12	40	10	5	40

Table A.2: Summary of results using 0 ms inter-packet delays

	Payload size	Target 1	Target 2	Target 3	Target 4	Combined
Packets dropped	0 bytes	0%	0%	2.20%	0.02%	0.55%
	100 bytes	0%	1.34%	14.5%	0.39%	4.00%
	500 bytes	0%	0%	32.2%	1.37%	8.26%
Out-of-order packets	0 bytes	0%	0%	7.00%	0.13%	1.81%
	100 bytes	12.2%	0.13%	22.6%	27.7%	15.5%
	500 bytes	2.25%	0.26%	7.13%	26.6%	8.94%
Late packets	0 bytes	0%	0%	0.67%	0.07%	0.18%
	100 bytes	6.10%	0.06%	9.89%	13.8%	7.42%
	500 bytes	0.81%	0.13%	3.57%	13.3%	4.39%
Average error	0 bytes	0	0	1.90	1.00	1.88
	100 bytes	1.00	1.00	1.35	1.00	1.12
	500 bytes	1.87	1.00	1.02	1.01	1.07
Maximum error	0 bytes	0	0	24	1	24
	100 bytes	1	1	22	1	22
	500 bytes	32	1	4	3	32

Table A.3: Summary of results using 1 ms inter-packet delays

	Payload size	Target 1	Target 2	Target 3	Target 4	Combined
Packets dropped	0 bytes	0%	0%	2.43%	0.09%	0.62%
	100 bytes	0%	0%	2.24%	0.02%	0.56%
	500 bytes	0%	0%	26.7%	0.02%	6.57%
Out-of-order packets	0 bytes	0%	0%	0%	0%	0%
	100 bytes	0.04%	0%	6.35%	2.30%	2.13%
	500 bytes	0%	0%	0.57%	0.04%	0.15%
Late packets	0 bytes	0%	0%	0%	0%	0%
	100 bytes	0.02%	0%	0.98%	1.98%	0.73%
	500 bytes	0%	0%	0.28%	0.02%	0.07%
Average error	0 bytes	0	0	0	0	0
	100 bytes	1.00	0	2.11	14.6	5.42
	500 bytes	0	0	2.15	1.00	2.07
Maximum error	0 bytes	0	0	0	0	0
	100 bytes	1	0	32	87	87
	500 bytes	0	0	16	1	16

Table A.4: Summary of results using 2 ms inter-packet delays

default of 64), and all arrived at each target with the same TTL, so if any packets took alternate routes, they were all of the same length.

The increasing trend in packet dropping as packet size increases suggests that packet loss is more a function of data rate than packet rate. Packet re-ordering and error, however, appear to be strongly dependent on packet rate. Interestingly, the packet re-ordering rate is higher for 100 byte than for 0 byte packets in the 1 ms and 2 ms tests, despite having a lower packet rate. It is unknown if this is significant or merely an artifact of the small sample size.

Packet dropping and re-ordering rates decreased sharply as inter-packet delays increased; less than 1% were being dropped or re-ordered on average in all tests using 4 ms or longer delays. However, Target 3 persisted in dropping up to 3% and re-ordering up to 1% in all tests, so dropping and re-ordering *does* still occur with longer delays. Additionally, Target 3 dropped the 4^{th} packet of every burst (sent to port

7/udp) for unknown reasons, probably a firewall misconfiguration.

Figures A.1 and A.2 show the occurrence of packet re-ordering as a function of packet sequence number and the average occurrence of error values while using 0-byte UDP payloads. The rate of packet re-ordering seemed to remain relatively constant over the course of the experiment. The plurality of out-of-order packets were only out of place by one position, but non-trivial numbers were out of order by up to 10 places; one packet was observed with an error of 94 out of a maximum possible of 99. Average error magnitudes decreased significantly as inter-packet delay increased, but small percentages of packets were still observed with high error values. Error rates and values followed similar trends when using 100-byte and 500-byte payloads, as seen in Figures A.3, A.4, A.3 and A.6. The dependence of re-ordering on packet rate is again clearly visible; 0-byte packets, with the highest packet rates, were re-ordered both most often and by the most positions, whereas fewer 100-byte and 500-byte packets were re-ordered and those that were were re-ordered by fewer positions. It is unknown why more 100-byte and 500-byte packets were re-ordered than 0-byte packets in the 1 ms tests.

Unlike packet re-ordering, packet loss is strongly dependent on packet sequence numbers, as seen in Figures A.7, A.8 and A.9. Apart from Target 3's dropping of the 7^{th} packet of every burst, little packet loss occurred until about the 50^{th} packet of each burst, when the percentage of packets being dropped in the 0 ms tests increased dramatically. High packet loss in tests with longer inter-packet delays didn't start until much later, if at all. Since the percentage of packets dropped differed greatly between targets, most of the packet loss likely occurred at or near the targets, rather than near the server, and may be due to router queue overflows.

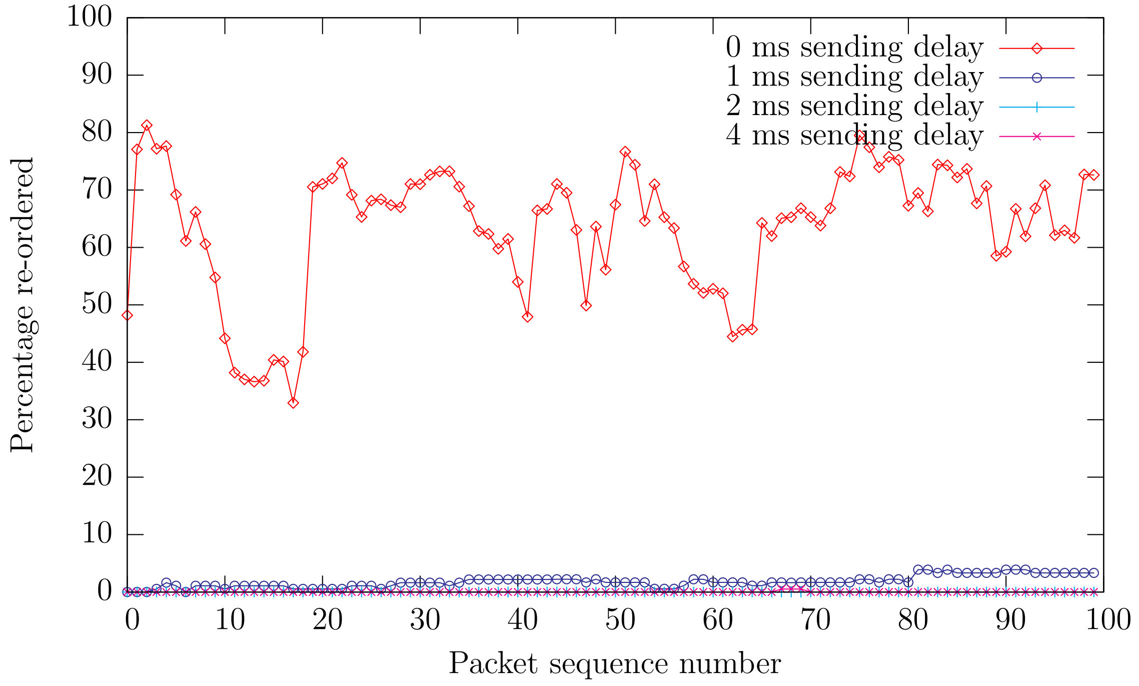


Figure A.1: Out-of-order packets using 0 byte payloads

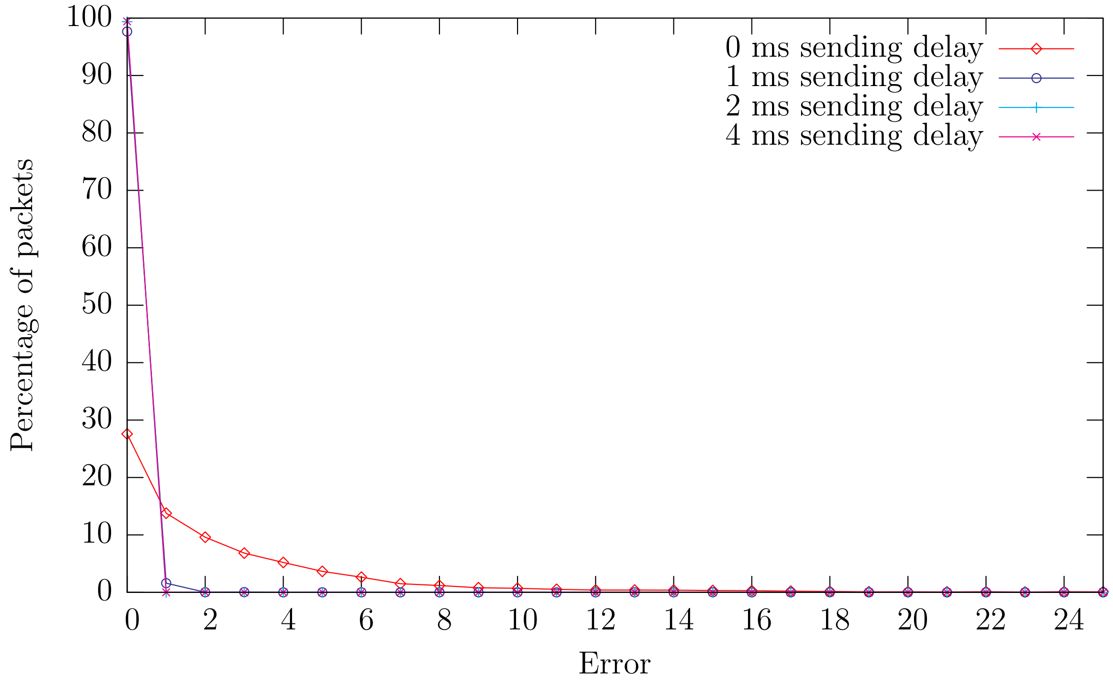


Figure A.2: Magnitude of errors using 0 byte payloads

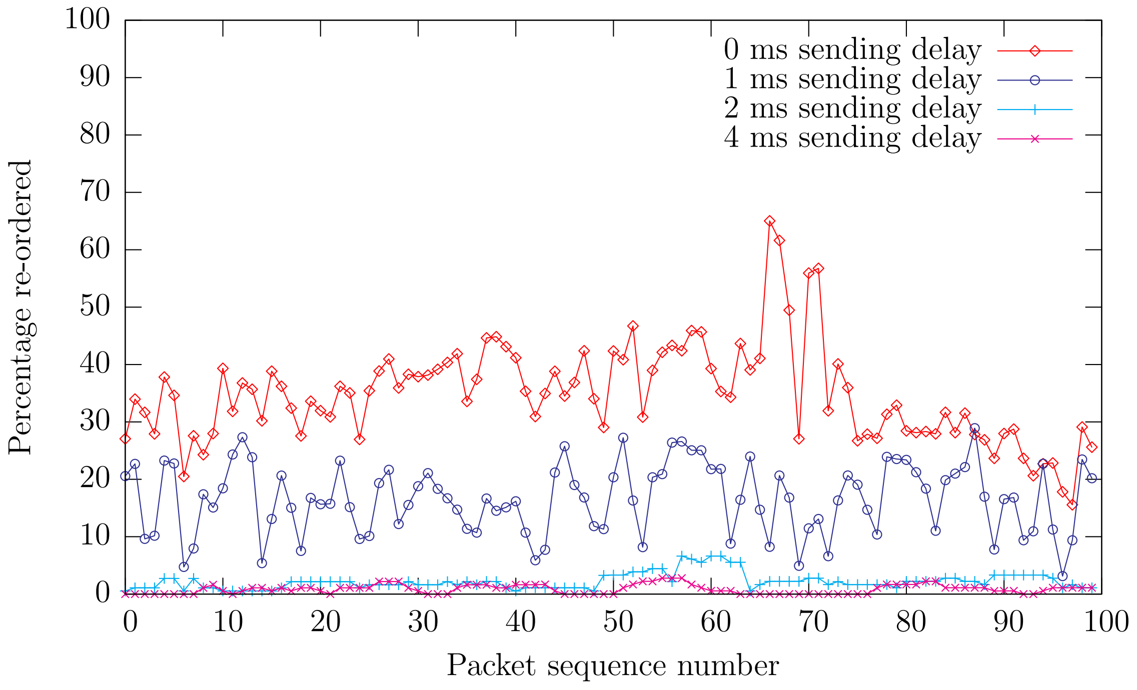


Figure A.3: Out-of-order packets using 100 byte payloads

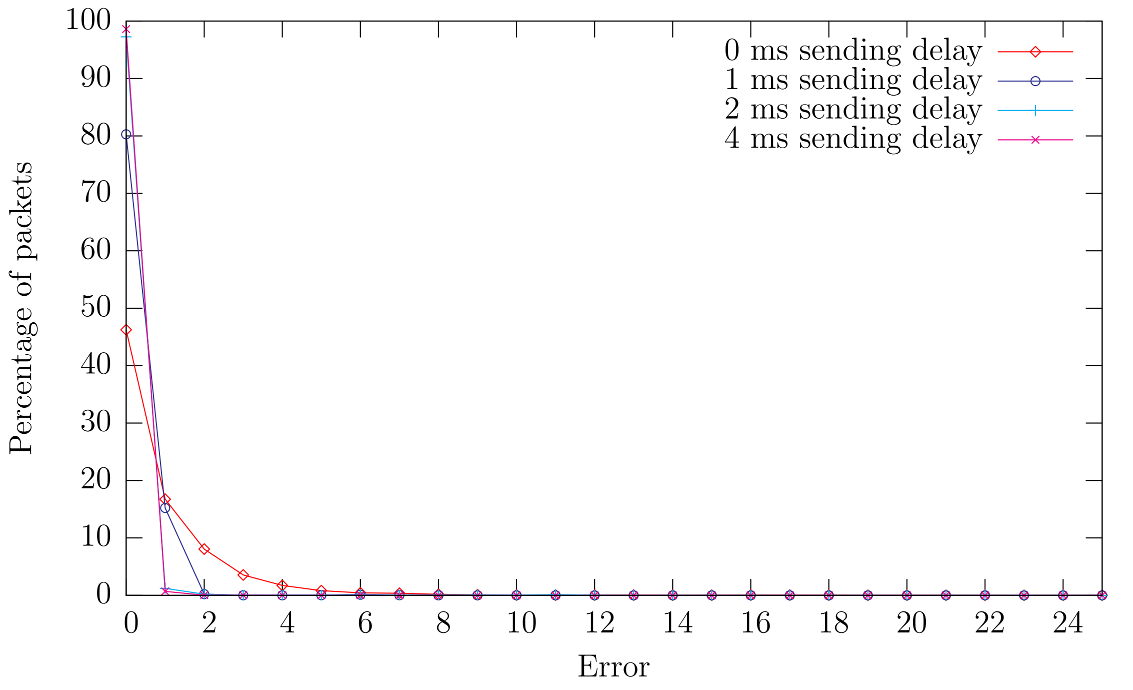


Figure A.4: Magnitude of errors using 100 byte payloads

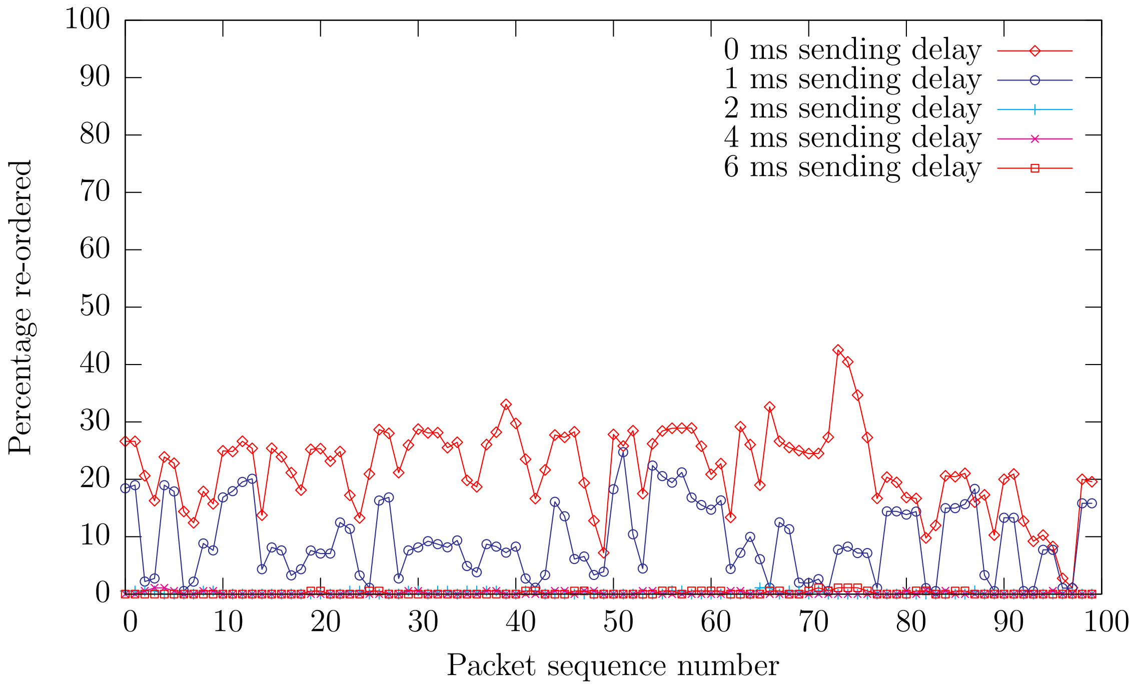


Figure A.5: Out-of-order packets using 500 byte payloads

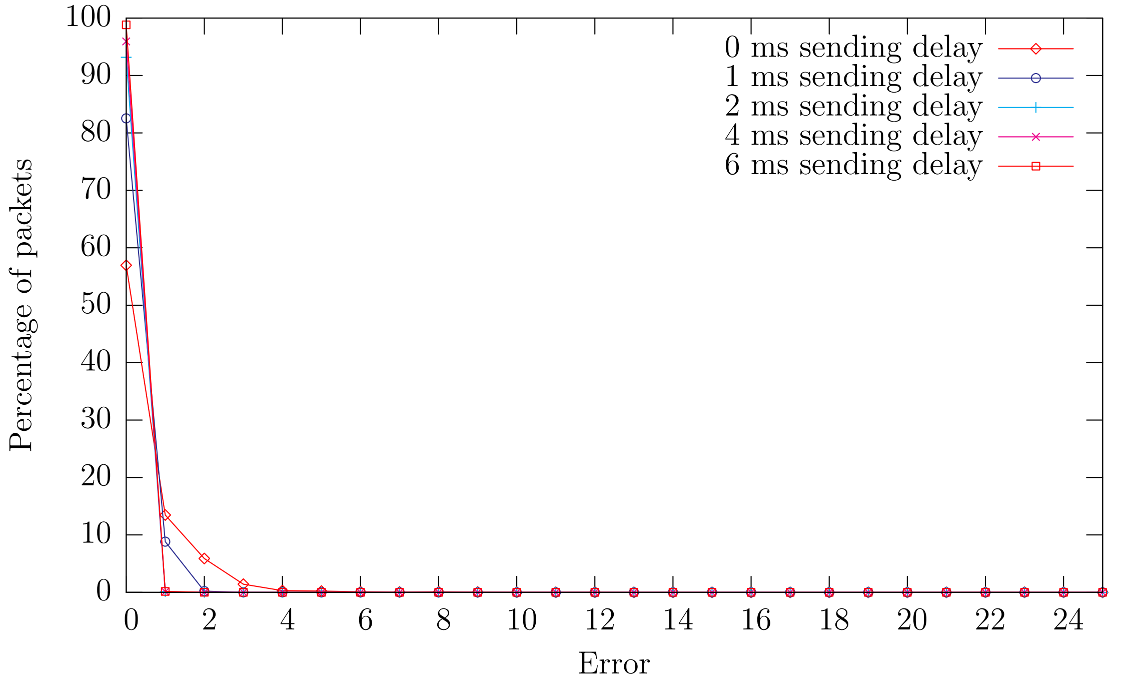


Figure A.6: Magnitude of errors using 500 byte payloads

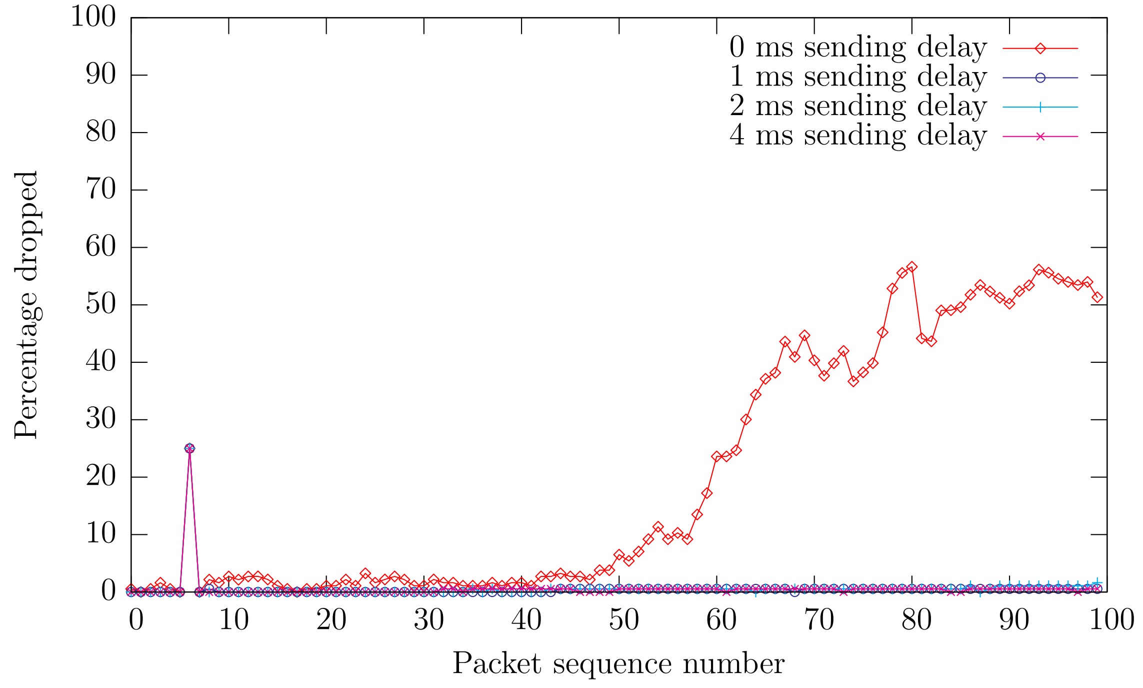


Figure A.7: Packet loss using 0 byte UDP payloads

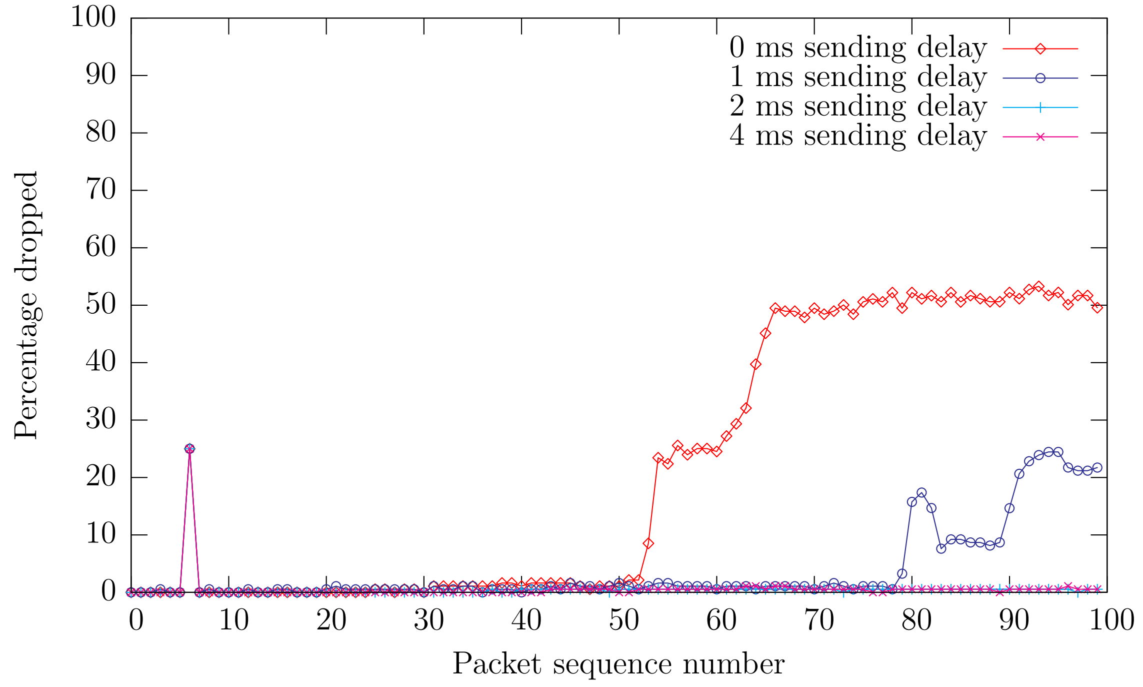


Figure A.8: Packet loss using 100 byte UDP payloads

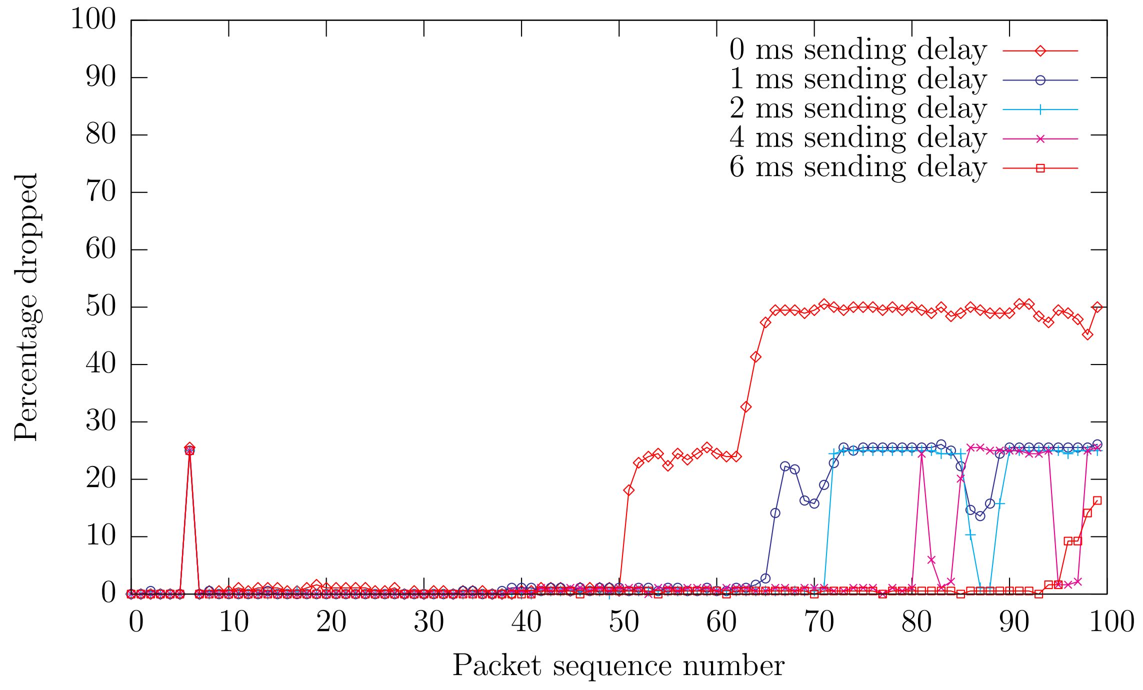


Figure A.9: Packet loss using 500 byte UDP payloads

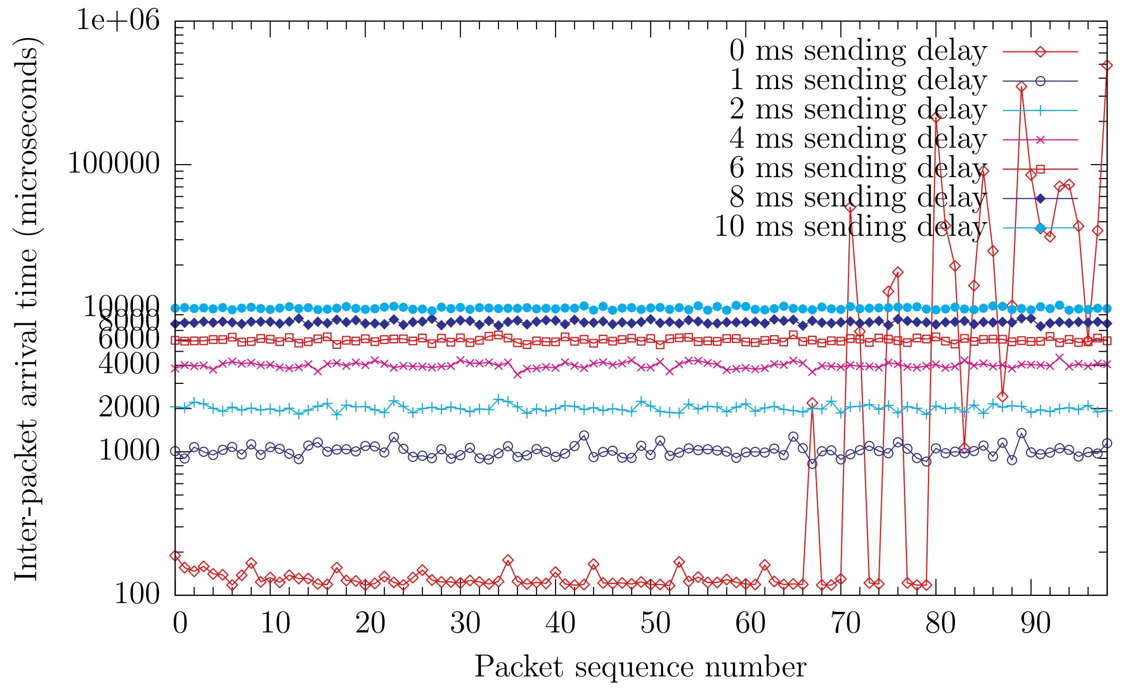


Figure A.10: Target 1: inter-packet arrival times using 0 byte UDP payloads

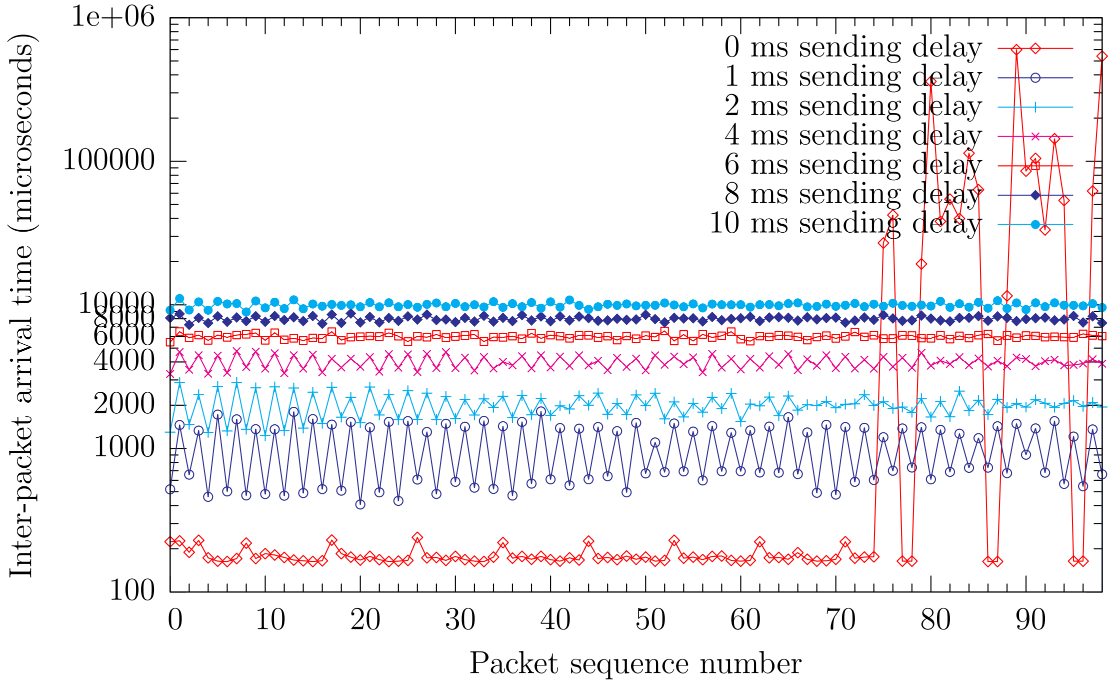


Figure A.11: Target 1: inter-packet arrival times using 100 byte payloads

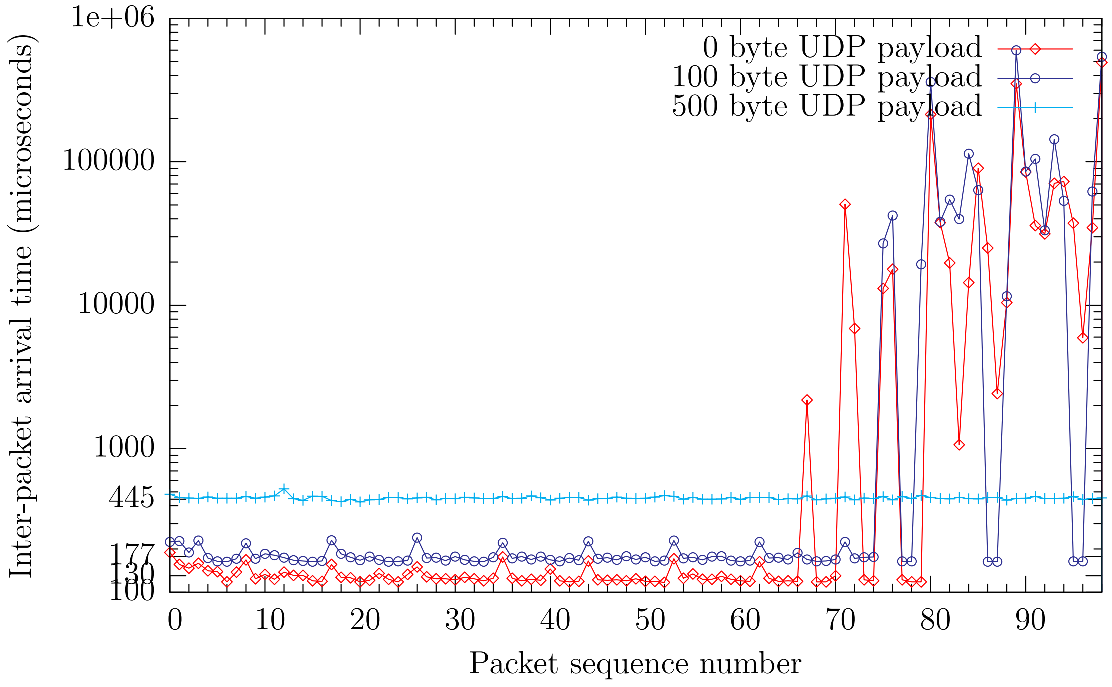


Figure A.12: Target 1: inter-packet arrival times using 0 ms inter-packet delays

Payload size	Sending inter-packet time (μs)	Arrival inter-packet time (μs)			
		Target 1	Target 2	Target 3	Target 4
0 bytes	13	130	289	1253	795
	968	989	988	1201	880
	1920	1972	1971	2047	1721
	3826	3926	3931	4005	3419
100 bytes	12	177	555	4264	528
	965	976	999	4405	889
	1920	1961	1979	4344	1684
	3823	3909	3918	4349	3402
500 bytes	11	445	1656	16777	745
	967	1000	1658	16812	894
	1919	1966	1973	16667	1707
	3825	3915	3891	16760	3405

Table A.5: Average inter-packet times when sending and receiving

Target 1 didn't drop any packets at all; rather, after about the 65th packet, the average inter-packet arrival time grew by several orders of magnitude in the 1 ms tests with 0- and 100-byte payloads, while remaining constant with 500-byte payloads and longer inter-packet delays (see Figures A.10, A.11 and A.12). The other three targets all saw roughly constant inter-packet arrival times. This suggests that the choke-point along the route to Target 1 employs a rather different buffering strategy than those on the other targets' routes. The regular spikes in all three graphs suggest that buffers are being processed in bursts.

The average inter-packet arrival times for packets early in each 0ms burst also suggest that the loss and delay effects are due to buffering. Table A.5 shows the average inter-packet sending and arrival times for the first 50 packets in the 0, 1, 2 and 4ms tests with all three packet sizes. The increases in inter-packet times between sending and receiving are most likely caused by processing delays at routers; packets that arrive faster than they can be processed must be buffered. Once routers' buffers are full, they start dropping packets. Comparing with Tables A.2, A.3 and A.4, packets were dropped most when packet inter-arrival times were significantly greater than inter-packet sending delays.

The sending times shown in Table A.5 are the averages of those measured using the `gettimeofday()` function by all four targets and differ slightly from the intended times. Due to processing, the time between packets in the 0 ms test is non-zero, while for unknown reasons, the delay timer used in the other tests tended to run a few percent fast, especially when sending to target 4. Because of this, small differences in sending and receiving inter-packet times are not significant.

A.4 Conclusions

For more accurate results, this experiment should be repeated using considerably more targets (enough that tests of statistical significance could be done). It would also be useful to gather information about the targets themselves and their activity during the experiment to determine if bursts in packet loss or re-ordering were due to bursts in unrelated network traffic or CPU use. Due to differences in the definition of out-of-order packets, it is difficult to compare these results to those of other experiments. However, the following conclusions can be made from the available data:

- The probability of packets sent at a high rate onto a high-speed network being delivered out of order is high, especially for very small packets. This probability drops off sharply when inter-packet delays are 2 ms or more.

- Of the packets that are delivered out of order in a burst, non-trivial numbers are delivered out of order by more than one or two positions, especially for very small packets.

- The probability of a packet in a burst being dropped is low near the beginning of the burst, but increases significantly farther along in the burst.

Appendix B

Proof That Permutation Knocking Is Inefficient

Permutation knocking, as introduced in Section 4.3.1, involves encoding information in a permutation of a set of port numbers, rather than in the numbers used themselves. In this section, I present a proof that for any size of message that can be sent using permutation knocking, the same message can be sent using the standard port knocking encoding, using *both* a smaller port range and fewer knocks.

As stated in Section 4.3.1, permutation knocking allows messages of up to $m = \lfloor \log_2(n!) \rfloor$ bits to be encoded into permutations of sets of n ports. Transmitting such a message to a server necessarily requires that n knocks be sent and that the server is listening on n ports. The standard encoding requires port sets with sizes that are integer powers of 2, so if n is an upper bound on the number of available ports, then $s = 2^{\lfloor \log_2(n) \rfloor}$ is the maximum number of usable ports for the standard encoding. Each port can encode $log_2(s)$ bits of information, so the total message of m bits will require $r = \frac{m}{\log_2(s)}$ knocks. By definition, $s \leq n$; this leaves open the question of how r compares to n.

Theorem 1. *For all integers $n \geq 2$, with m, n, r, and s defined as above, $r \leq n$.*

Proof of Theorem 1 requires Proposition 1.

Proposition 1. *For all $n \geq 6$, $n! \leq \left(\frac{n}{2}\right)^n$*

Proof (by induction). $6! = 720$ and $\left(\frac{6}{2}\right)^6 = 729$, so Prop. 1 is true for $n = 6$.

Assuming that $n! \leq \left(\frac{n}{2}\right)^n$ for some $n \geq 6$,

$$\begin{aligned}
(n+1)! &= (n+1)(n!) \\
&\leq (n+1)\left(\frac{n}{2}\right)^n && \text{by the inductive hypothesis} \\
&= n\left(\frac{n}{2}\right)^n + \left(\frac{n}{2}\right)^n \\
&= \frac{n^{n+1} + n^n}{2^n} \\
&= \frac{2n^{n+1} + 2n^n}{2^{n+1}} \\
&< \frac{2n^{n+1} + \frac{n+3}{2}n^n}{2^{n+1}} && \text{since } n \geq 6 \\
&= \frac{n^{n+1} + (n+1)n^n + \frac{n(n+1)}{2}n^{n-1}}{2^{n+1}} \\
&= \frac{\binom{n+1}{0}n^{n+1} + \binom{n+1}{1}n^n + \binom{n+1}{2}n^{n-1}}{2^{n+1}} \\
&< \frac{(n+1)^{n+1}}{2^{n+1}} && \text{by the binomial theorem.}
\end{aligned}$$

□

Proof of Theorem 1. Since $r = \frac{m}{\log_2(s)}$, $m = \lfloor \log_2(n!) \rfloor$, and $s = 2^{\lfloor \log_2(n) \rfloor}$, it is sufficient to prove that

$$\frac{\lfloor \log_2(n!) \rfloor}{\log_2(2^{\lfloor \log_2(n) \rfloor})} = \frac{\lfloor \log_2(n!) \rfloor}{\lfloor \log_2(n) \rfloor} \leq n$$

By Prop. 1, $\forall n \geq 6$,

$$\begin{aligned}
n! &\leq \left(\frac{n}{2}\right)^n \\
&\leq 2^{\log_2\left(\frac{n}{2}\right)^n} \\
&\leq 2^{n(\log_2(n)-1)}
\end{aligned}$$

Taking the log of both sides,

$$\log_2(n!) \leq n(\log_2(n) - 1)$$

Rearranging this gives

$$\frac{\log_2(n!)}{\log_2(n)-1} \leq n$$

Since $x-1 < \lfloor x \rfloor \leq x$,

$$\frac{\lfloor \log_2(n!) \rfloor}{\lfloor \log_2(n) \rfloor} \leq \frac{\log_2(n!)}{\log_2(n)-1} \leq n$$

This proves the theorem for all $n \geq 6$. Cases for $n = 2,3,4,5$ can be proved by substitution:

$$\begin{aligned}
\frac{\lfloor \log_2(2!) \rfloor}{\lfloor \log_2(2) \rfloor} &= 1 \quad\;\;\; \leq 2 \\
\frac{\lfloor \log_2(3!) \rfloor}{\lfloor \log_2(3) \rfloor} &< 1.64 \leq 3 \\
\frac{\lfloor \log_2(4!) \rfloor}{\lfloor \log_2(4) \rfloor} &< 2.30 \leq 4 \\
\frac{\lfloor \log_2(5!) \rfloor}{\lfloor \log_2(5) \rfloor} &< 2.98 \leq 5
\end{aligned}$$

□

Since it is impossible to encode any message over fewer than 2 ports, Theorem 1 proves that standard encoding can encode any message using a smaller (or equal) port range and fewer (or equal) knocks than permutation encoding.