Defending Systems Against Application-Level Denial of Service Attacks

Gal Badishi

Defending Systems Against Application-Level Denial of Service Attacks

Research Thesis

Submitted in Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy

Gal Badishi

Submitted to the Senate of the Technion — Israel Institute of Technology

HESHVAN 5768 HAIFA NOVEMBER 2007

The Research Thesis Was Done Under The Supervision of Prof. Idit Keidar in the Department of Electrical Engineering

Acknowledgment

First and foremost, I would like to thank my advisor, Prof. Idit Keidar, for simply being everything I could ever ask for in an advisor.

A big thanks also goes to my supporting and loving family, and to my friends and colleagues, Aran Bergman, Isaskhar (Zigi) Walter, Nadav Lavi and Zvika Guz, for their assistance and support, and for just being good friends.

Last but not least, I thank Miri Shaul for inspiring me to do things I never thought I would do.

The generous financial help of the Israeli Ministry of Science and the Technion is gratefully acknowledged

Contents

List of Figures

List of Tables

Abstract

Many systems today operate over the Internet, which is a hostile environment where many attacks are common, e.g., penetration, forgery, and *denial of service* (DoS) attacks. Thus, security measures should be taken in order to ensure the survivability of a system even when facing failures or attacks. One of the most devastating attacks is an application-level DoS attack, which aims to deplete the resources of end hosts by abusing application traffic. Dealing with such an attack is a challenge that concerns both the industry and the academic community.

Our research begins by presenting *Drum* – a gossip-based application-level multicast protocol that is resistant to application-level DoS attacks. Drum ensures correct delivery of multicast messages to all nodes in a timely fashion, w.h.p., even when a large percentage of the nodes is under DoS attacks. Drum is analyzed, simulated, and implemented, and all results show its good traits.

Our research on Drum continues by allowing each node to locally adapt its behavior to the locally-perceived state of the system. We model the multicast problem as an optimization problem, and solve it to find a solution to the adaptation problem. We show that even though nodes adapt their behavior using local knowledge only, the total expected propagation time of messages in an attacked system is improved.

Having found a DoS solution for application-level multicast, we turn to protect other applications. Obviously, there is a vast number of different applications, and tailoring a specialized solution for each and every one of them is not viable. Thus, it is important to find some general DoS solution that can be applied in a multitude of applications. We start by developing a simple and general building block – DoS-resistant two-party communication. We define a formal model of a realistic port-based rationing channel, and based on that model we develop a protocol, *ϕ-Hopper*, that is resilient to DoS attacks. We prove the protocol's resilience by rigorously analyzing its *success rate*, i.e., the number of valid messages that are sent and are correctly received at the other end. We show that existing protocols that validate communication using an unchanged secret payload are bound to eventually fail, while ϕ-Hopper uses packet fields, e.g. ports, to store its random payload, and proactively hops between field values.

Finally, we use ϕ-Hopper as one component of *Beaver* – a multi-party solution that allows a server to communicate with many clients, even in the face of application-level DoS attacks. We

design a complete system to protect legacy servers from DoS attacks, with minimal alterations to the communicating parties. Our design provides mechanisms for registration, admission, and DoS-resistant communication between the parties involved. We show that the system is robust even when DoS attacks and compromised clients are present.

List of Symbols

Abberviations and Acronyms

ADM Admission Server

CDF Cumulative Distribution Form

DoS Denial of Service

DDoS Distributed Denial of Service

FI Filtering Identifier

LAN Local Area Network

MAC Message Authentication Code

NDIS Network Driver Interface Specification

PRF Pseudorandom Function

RR Round Robin

WAN Wide Area Network

Chapter 1

Introduction and Background

The proliferation of Denial of Service (DoS) attacks in recent years [12] has increased the interest in protecting systems from such attacks [46, 43, 42, 27, 1]. In a remotely-launched DoS attack, the attacker attempts to disrupt some service by crafting special network packets. We concentrate on attackers that generate bogus requests and send them to the target, with the intention of overwhelming the target and degrading its service. Such adversaries can magnify the severity of their attack by first infiltrating many computers, and then utilizing them as "zombies" to perform a Distributed DoS (DDoS) attack [51].

The first line of defense in protecting against DoS attacks is to protect the network [46, 43]. Indeed, if the network is congested, there is little one can do to allow valid communication to take place. However, protecting the network does not solve the DoS problem for the application. Common network-protection mechanisms include filtering according to field values on packet headers (such as addresses and ports), and rate-limiting traffic. But packet headers can be spoofed by the adversary, and rate-limiting discards messages indiscriminately, throwing away valid messages as well. Moreover, it is possible to perform a DoS attack on the application without overloading the network, especially when the application invests many resources in each incoming request, as might happen, for example, when using cryptographic primitives to authenticate or encrypt/decrypt packets.

One might assume that authenticating every packet in transit, e.g. using IPSec [3], removes the problem of DoS, as bogus packets are identified and dropped, while valid packets are authenticated and delivered to the application. But while this is indeed the case, the cost of per-packet authentication creates a new attack vector – targeting the authenticator. Thus, as we show in this work, using IPSec alone to protect from DoS attacks is insufficient.

Our goal is to provide mechanisms, tools and systems to protect against DoS attacks when the network is not congested. We leverage existing cheap and simple solutions such as filtering and

rate-limiting to provide strong and robust protection against DoS. Our systems are practical and easy to implement and deploy.

We provide means to protect the following communication methods:

- (Gossip-based) Multicast [8, 14].
- Two-party communication.
- Client-server communication [27, 1].

We formally model and prove the correctness and effectiveness of our systems when under DoS attacks. We provide a framework for reasoning about different approaches and best strategies, and compare different solutions. Finally, we quantify the robustness of our systems by providing results from simulations and measurements of real implementations.

In Chapter 3 we propose a framework and methodology for quantifying the effect of DoS attacks on a distributed system. We present a systematic study of the resistance of gossip-based multicast protocols to DoS attacks. We show that even distributed and randomized gossip-based protocols, which eliminate single points of failure, do not necessarily eliminate vulnerabilities to DoS attacks. We propose Drum – a simple gossip-based multicast protocol that eliminates such vulnerabilities. Drum was implemented in Java and tested on a large cluster. We show, using closed-form mathematical analysis, simulations, and empirical tests, that Drum survives severe DoS attacks. The work presented in Chapter 3 appeared in [6, 7].

In Chapter 4 we improve the resistance of gossip-based multicast protocols to (Distributed) Denial of Service (DDoS) attacks using dynamic local adaptations at each node. Each node estimates the current state of the attack on the system, and then adapts its behavior according to this local estimation. The adaptation is achieved through modeling the problem of propagating messages under a DoS attack as an optimization problem, and solving it using linear programming, independently at each node. Simulation results show that when the system is under attack, the local decisions each node takes bring the system to a stable point, which is the solution of the linear programming problem. The adaptation leads to propagation times that are 30% faster than those of existing DoS-resistant gossip-based protocols.

In Chapter 5 we consider the problem of overcoming DDoS attacks by realistic adversaries that have knowledge of their attack's successfulness, e.g., by observing service performance degradation, or by eavesdropping on messages or parts thereof. A solution for this problem in a high-speed network environment necessitates lightweight mechanisms for differentiating between valid traffic and the attacker's packets. The main challenge in presenting such a solution is to exploit existing packet filtering mechanisms in a way that allows fast processing of packets, but is complex enough so that the attacker cannot efficiently craft packets that pass the filters. We show a protocol, ϕ-Hopper, that mitigates DoS attacks by adversaries that can eavesdrop and (with some delay) adapt

their attacks accordingly. The protocol uses only available, efficient packet filtering mechanisms based mainly on addresses and port numbers. ϕ-Hopper avoids the use of fixed ports, and instead performs 'pseudo-random port hopping'. We model the underlying packet-filtering services and define measures for the capabilities of the adversary and for the success rate of the protocol. Using these, we provide a novel rigorous analysis of the impact of DoS on an end-to-end protocol, and show that ϕ-Hopper provides effective DoS prevention for realistic attack and deployment scenarios. The work presented in Chapter 5 appeared in [4, 5].

In Chapter 6 we present two prototype implementations of ϕ-Hopper, one as part of IPSec in a Linux kernel, and a second as an NDIS hook driver on a Windows machine. Our implementations show that ϕ-Hopper is practical, and easy to deploy. We also present results of experiments, using the two implementations. Our measurements illustrate that ϕ-Hopper withstands severe DoS attacks without hampering the client-server communication. Moreover, ϕ-Hopper is simple and easy to deploy.

In Chapter 7 we present Beaver, a method and architecture to "build dams" (filters) to protect servers from DDoS attacks (floods). Beaver allows efficient filtering of DoS traffic using low-cost, high-performance, readily-available packet filtering mechanisms. Beaver improves on previous solutions by not requiring cryptographic processing per message, allowing the use of efficient routing (avoiding overlays), and establishing keys and state as needed.

Chapter 2

Methodology

We evaluate our systems using closed-form formal analysis, simulations, and experiments. We model the attacker and the system, and then perform an analysis of our solution based on that model. Such an analysis allows us to prove the correctness of our solution for all scenarios, as oppossed to just using experiments, or simulations, which show the practicability of the solution in specific settings.

Our formal analysis tries to find lower and upper bounds on metrics that are relevant to the problem at hand. For instance, in Chapter 3 we find lower and upper bounds on message propagation time in gossip-based multicast protocols that are under attack. In Chapter 5 we find a lower bound on the success rate and delivery probability, which is the percentage of valid message that reach the target under attack. We usually find the lower and upper bounds by solving optimization problems. For example, in Chapter 4 we use linear programming to solve an optimization problem for the node's best behavior under attack. In Chapter 5 we find the attacker's best strategy.

To check the effect of the simplifying assumptions in our model formal analysis, we sometimes simulate the system using MATLAB (see Chapters 3 and 4). Since our protocols use random information, we run each experiment 100 times, and each data point we use is an average of the results of these 100 experiments.

Finally, we implement our systems in C (Chapter 6) or Java (Chapter 3) and test their performance under real DoS attacks. We compare our results to the results obtained by the simulations and the results predicted by the analysis, which assumes a simplified model.

Our different techniques for analyzing the systems validate each other, and thus we can be certain that the systems are robust. The evaluation frameworks that we build allow us to compare different systems and protocols, and show that our solutions improve the resistance to DoS attacks compared to existing solutions.

Chapter 3

Drum

One of the biggest security threats faced by a distributed system is a *denial of service* (DoS) attack, in which an attacker makes a system unresponsive by forcing it to handle bogus requests that consume all available resources. In a *distributed denial of service* (DDoS) attack, the attacker utilizes multiple computers as the source of a DoS attack, in order to increase the attack strength. Since a DDoS attack is essentially a strong DoS attack, we will consider them to be the same. In 2003, approximately 42% of U.S. organizations, including government agencies, financial institutions, medical institutions and universities, were faced with DoS attacks [12]. That year, DoS attacks were the second most financially damaging attacks, only short of theft of proprietary information, and far above other attacks [12]. Therefore, coping with DoS attacks is essential when deploying services in a hostile environment such as the Internet [39].

As a first defense, one may protect a system against DoS attacks using network-level mechanisms [46, 42, 43]. These mechanisms involve rate-limiting incoming traffic, and filtering packets according to their headers. However, network-level filters cannot detect DoS attacks at the application level, when the traffic seems legitimate. Even if means are in place to protect against network-level DoS, an attack can still be performed at the application level, as the bandwidth needed to perform such an attack is usually lower. This is especially true if the application performs intensive computations for each message, as occurs, e.g., with secure protocols based on digital signatures.

As network-level DoS-mitigation solutions are increasingly available, application level DoS attacks are becoming a major concern [53]. Consequently, vendors have begun employing some measures against DoS attacks at the application layer [24, 41]. Such solutions are commonly deployed at the network/firewall level, although they are application-specific. However, these measures are usually just hard-coded validity checks for well-known protocols, and do not contain means to deal with resource exhaustion caused by the application. In this chapter, we are concerned with coping with DoS attacks in application-level multicast protocols. The basic idea is to assume simple and

general mechanisms at the network/firewall level and to exploit them at the application (multicast protocol) level.

To quantify the effects of DoS attacks, we measure their influence on the time it takes to propagate a message to all the processes in the system, as well as on the average throughput processes can receive. We do this using asymptotic analysis, simulations, and measurements.

We focus on large-scale distributed systems (e.g., 1000 processes). A DoS attack that targets every process in a large system inevitably causes performance degradation, but also requires vast resources. In order to be effective even with limited resources, attackers target vulnerable parts of the system. For example, consider a tree-based multicast protocol; by targeting a single inner node in the tree, an attacker can effectively partition the multicast group. Hence, eliminating single points of failure is an essential step in constructing protocols that are less vulnerable to DoS attacks.

We therefore focus on gossip-based (epidemic) multicast protocols [13, 8, 14, 19, 26, 31, 25], which eliminate single points of failure using redundancy and random choices. Such protocols are robust and have been shown to provide graceful degradation in the face of amounting failures [20, 32]. As in previous work, e.g., [8, 31], we assume that the gossip-based multicast system is deployed in a WAN environment, and as such, its nodes suffer from DoS attacks launched from outside the system. One may expect that such a system will not suffer from vulnerabilities to DoS attacks, since it can continue to be effective when many processes fail. Surprisingly, we show that gossip-based protocols can be extremely vulnerable to DoS attacks targeted at a small subset of the processes. This occurs because an attacker can effectively isolate a small set of processes from the rest of the group by attacking this set.

Having observed the vulnerabilities of traditional protocols, we turn to search for ways to eliminate these vulnerabilities. Specifically, our goal is to design a protocol that does not allow an attacker to increase the damage it causes by focusing on a subset of the processes. We are familiar with only one previous work, by Minsky and Schneider [38], that addresses DoS attacks on a gossip-based protocol. However, the problem they consider differs from ours in a way that renders their approach inapplicable to our setting (see Section 3.1), and moreover, they only deal with limited attack strengths.

We present *Drum* (DoS-Resistant Unforgeable Multicast), a gossip-based multicast protocol, which, using a few simple ideas, eliminates common vulnerabilities to DoS attacks: the best attack against Drum requires the attacker to target the entire system. The 3 main ideas used in Drum are:

1. Simultaneously using two gossiping techniques, *push* and *pull*.

2. Allocating separate resources for each operation.

3. Using random ports whenever possible, for each communication channel.

Mathematical analysis and simulations show that Drum indeed achieves our design goal: an attacker cannot substantially hinder Drum's performance by targeting a small subset of the processes. When an adversary has a large sending capacity, its most effective attack against Drum is an all-out attack that distributes the attacking power as broadly as possible. We concentrate on heavy attacks since they are the most damaging, and one can expect them to happen in actual scenarios [51]. Obviously, performance degradation due to a broad all-out DDoS attack is unavoidable for any multicast protocol, and indeed all the tested protocols exhibit the same performance degradation under such a broad attack. In contrast, under an attack that focuses on a strict subset of the processes, Drum's latency remains *constant* as the attack strength increases, whereas in traditional protocols, the latency grows *linearly* with the attack strength.

We have implemented Drum in Java and tested it on a cluster of workstations. Our measurements validate the analysis and simulation results, and show that Drum can withstand severe DoS attacks, where naïve protocols that do not take any measures against DoS attacks completely collapse in terms of latency and throughput.
In summary, this chapter makes the following contributions:

- It presents a new framework and methodology for quantifying the effects of DoS attacks. We are not familiar with any previously suggested metrics for DoS-resistance nor with previous attempts to quantify the effect of DoS attacks on a system.

- It uses the new methodology to conduct the first systematic study of the impact of DoS attacks on multicast protocols. This study exposes vulnerabilities in traditional fault-tolerant protocols, showing that robustness, although necessary, is not sufficient for DoS-mitigation.

- It presents Drum, a simple gossip-based multicast protocol that eliminates such vulnerabilities. We believe that the ideas used in Drum can serve to mitigate the effects of DoS attacks on other protocols as well.

- It provides closed-form asymptotic analyses as well as simulations and measurements of gossip-based multicast protocols under DoS attacks varying in strength and extent.

This chapter proceeds as follows: Section 3.1 gives background and related work. Section 3.2 presents the system model. Section 3.3 describes Drum. Section 3.4 presents our evaluation methodology and considered attack models. The following three sections evaluate Drum and compare it to traditional gossip-based protocols using various tools: Section 3.5 gives closed-form asymptotic latency bounds; Section 3.6 provides a thorough evaluation using simulations; and Section 3.7 presents latency and throughput measurements. Section 3.8 evaluates the usefulness of two specific DoS-mitigation techniques used in Drum. Finally, we provide some derivations for the analysis.

3.1 Background and Related Work

Gossip-based dissemination [13] is a leading approach in the design of scalable reliable application-level multicast protocols, e.g., [8, 14, 19, 26, 31, 25]. Our work focuses on symmetric gossip-based multicast protocols like lpbcast [14]. We consider protocols that do not rely on external mechanisms such as IP multicast.

Such protocols work roughly as follows: each process locally divides its time into *gossip rounds*; rounds are not synchronized among the processes. In each round, the process randomly selects a small number of processes to gossip with, and tries to exchange information with them. Every piece of information is gossiped for a number of rounds. It has been shown that the propagation time of gossip protocols increases logarithmically with the number of processes [44, 25]. There are two methods for information dissemination: (1) *push*, in which the process sends messages to randomly selected processes; and (2) *pull*, in which the process requests messages from randomly selected processes. We show that both methods are susceptible to DoS attacks: attacking the incoming push channels of a process may prevent it from receiving valid messages, and attacking a process's incoming pull channels may prevent it from sending messages to valid targets. Some protocols use both methods [13, 25]. Karp et al. showed that combining push and pull allows the use of fewer transmissions to ensure data arrival to all group members [25].

Drum utilizes both methods, and in addition, allocates a bounded amount of resources for each operation (push and pull), so that a DoS attack on one operation does not hamper the other. Similar resource separation was also used in COCA [62], for the sake of overcoming DoS attacks on authentication servers. Drum further utilizes randomly selected ports for data transmission, thus making it difficult for an attacker to target these ports.

Secure gossip-based dissemination protocols were previously suggested by Malkhi et al. [35, 36, 37]. However, they did not deal with DoS attacks. Follow-up work by Minsky and Schneider [38] suggested a pull-based protocol that can endure limited DoS attacks by bounding the number of accepted requests per round. However, these works solve the *diffusion* problem, in which each message simultaneously originates at more than t correct processes, where up to t processes may suffer Byzantine failures. In contrast, we consider a multicast system where a message originates at a single source. Hence, using a pull-based solution that utilizes $t + 1$ disjoint paths, as suggested in [38], does not help in withstanding DoS attacks in the multicast system we consider. Moreover, Minsky and Schneider [38] focus on load rather than on DoS attacks; they include only a brief analysis of DoS attacks, under the assumption that no more than t processes perform the attack, and that each of them generates a single message per round (the reception bound is also assumed to be one message per round). In contrast, we focus on substantially more severe attacks, and study how system performance degrades as the attack strength increases.

Drum deals with DoS attacks at the application-level, assuming network-level defenses are already in place. Network-level DoS analysis and mitigation has been extensively dealt with [48, 9, 16, 55, 10, 46], but DoS-resistance at the secure multicast service layer has gotten little attention. We note that our work is the first that we know of that conducts a systematic study of the effect of DoS attacks on message latency.

Here, we focus on DoS attacks in which the attacker sends fabricated application messages. DoS can also be caused by churn, where processes rapidly join and leave [33], thus reducing availability. In Drum, as in other gossip-based protocols, churn has little effect on availability: even when as many as half of the processes fail, such protocols can continue to deliver messages reliably and with good quality of service [32]. A DoS attack of another form can be caused by process perturbations, whereby some processes are intermittently unresponsive. The effect of perturbations is analyzed in [8], where it is shown that probabilistic protocols, e.g., gossip-based protocols, solve this problem.

3.2 System Model

Drum supports probabilistically reliable multicast [8, 14, 25] among processes that are members of a group. Each message is created by exactly one group member (its *source*). Throughout this chapter we assume that the multicast group is static. There are n members in the group, and each process p has a list of the other $n-1$ group members.

Like previous gossip protocols [8, 14], we assume that the underlying network is fully-connected. The message latency varies, but it is bounded. The link-loss probability is constant, equal for all links, and independent of any other factor. The communication channels are insecure, meaning that senders of incoming messages cannot be reliably identified in a simple manner.

An adversary can generate fabricated messages. However, this requires the adversary to utilize resources. Malicious processes can perform DoS attacks on group members. We note that authenticating messages, e.g., using digital signatures, does not solve the DoS problem, as fabricated messages must be invalidated using a costly operation.

We assume that communication can take place on ports that change on demand, and that the multicast protocol can randomly choose to process a subset of the messages that arrive to a designated port, and ignore messages that arrive to other ports. We further assume that a DoS attack that does not specifically target the designated port does not affect the reception on this port (i.e., the application-level DoS attack does not cause a network-level DoS attack as well). This can be achieved using available network-level products [46, 42, 43].

We assume that a process can choose a random port for communication that the adversary cannot predict. We assume that the adversary only attacks ports it knows of. In our protocol, the use of a random port is limited in time, and the process notifies another process of this new

communication port by sending it a message stating the port number. We assume that it takes the adversary considerable time to react to this message, so that it cannot attack this random port while it is still in use. This assumption is justified, since an attacker that has significant strength is probably employing a DDoS attack and needs to notify its subordinates whenever it wishes to change targets.

3.3 DoS-Resistant Gossip-Based Multicast Protocol

Drum is a simple gossip protocol, which achieves DoS-resistance using a combination of pull and push operations, separate resource bounds for different operations, and the use of random ports in order to reduce the chance of a port being attacked. Each process, p, locally divides its time into rounds. The rounds are not synchronized among the processes. A round is typically in the order of a second, and its duration may vary according to local random choices. Every round, p chooses two small (constant size) random sets of processes (group members), $view_{push}$ and $view_{pull}$, and gossips with them. E.g., when these views consist of two processes each, this corresponds to a combined fan-out of four. In addition, p maintains a message buffer. Process p performs the following operations in each round:

- *Pull-request* – p sends a digest of the messages it has received to the processes in its $view_{pull}$, requesting missing messages. Pull-request messages are sent to a well-known port. The pull-request specifies a randomly selected port on which p will await responses, and p spawns a thread for listening on the chosen port. This thread is terminated after a few rounds.

- *Pull-reply* – in response to pull-request messages arriving on the well-known port, p randomly selects messages that it has and are missing from the received digests, and sends them to the destinations indicated in the requests.

- *Push* – in a traditional push operation, p randomly picks messages from its buffer, and sends them to each target t in its $view_{push}$. In order to avoid wasting bandwidth on messages that t already has, p instead requests t to reply with a message digest, as follows:

  1. p sends a *push-offer* to t, along with a random port on which it waits for a push-reply.
  2. t replies with a *push-reply* to p's random port, containing a digest of the messages t has, and a random port on which t waits for data messages.
  3. If p has messages that are missing from the digest, it chooses a random subset of these, and sends them back to t's randomly chosen port.

  The target process listens on a well-known port for push-offers.

Upon receiving a new data message, either by push or in response to a pull-request, p first performs some sanity checks. If the message passes these checks, p delivers it to the application and saves it in its message buffer for a number of rounds. The sanity checks employ cryptographic mechanisms, which ensure that the attacker has negligible probability of fabricating a message that passes these checks. Consequently, bogus messages impact only their first recipient. However, the sanity checks are costly in terms of execution time (e.g., verifying digital signatures). Thus, performing sanity checks at a high rate effectively causes DoS.

Resource allocation and bounds. In each round, p sends push-offers to all the processes in its $view_{push}$ and pull-requests to all the processes in its $view_{pull}$. If the total number of push-replies and pull-requests that arrive in a round exceeds p's sending capacity, then p equally divides its capacity between sending responses to push-replies and to pull-requests. Likewise, p responds to a bounded number (typically $|view_{push}|$) of push-offers in a round, and if more data messages than it can handle arrive, then p divides its capability for processing incoming data messages equally between messages arriving in response to pull-requests and those arriving in response to push-replies. The messages are randomly chosen from the incoming message buffers.

At the end of each round, p flushes its incoming message buffers. This is important, especially in the presence of DoS attacks, as an attacker can send more messages than p can handle in a round.

Achieving DoS-resistance. We now explain how the combination of push, pull, random port selections, and resource bounds achieves resistance to targeted DoS attacks. A DoS attack can flood a port with fabricated messages. Since the number of messages accepted on each port in a round is bounded, the probability of successfully receiving a given valid message M in a given round is inversely proportional to the total number of messages arriving on the same port as M in that round. Thanks to the separate resource bounds, an attack on one port does not reduce the probability for receiving valid messages on other ports.

In order to prevent a process from *sending* its messages using a *push* operation, one must attack (flood) the push-offer targets, the ports where push-replies are awaited, or the ports where data messages are awaited. However, the push destinations are randomly chosen in each round, as are the push-reply and data ports. Thus, the attacker has no way of predicting these choices.

Similarly, in order to prevent a process from *receiving* messages during a *pull* operation, one needs to target the destination of the pull-requests or the ports on which pull-replies arrive. However, the destinations and ports are randomly chosen. Thus, using the push operation, Drum achieves resilience to targeted attacks aimed at preventing a process from *sending* messages, and using the pull operation, it withstands attacks that try to prevent a process from *receiving* messages.

3.4 Evaluation Methodology

The most important contribution of this chapter is our thorough evaluation of the impact of various DoS attacks on gossip-based multicast protocols. In addition to examining the effect of DoS on Drum, we also measure the effectiveness of the DoS-mitigating techniques employed by it. We mostly concern ourselves with the benefits of combining both the push and pull methods. We evaluate three protocols: (i) Drum, (ii) *Push*, which uses only push operations, and (iii) *Pull*, which uses only pull operations. Pull and Push are implemented the same way Drum is, with the important measures of bounding the number of messages accepted in each round and using random ports. Thus, in comparing the three protocols, we study the effectiveness of combining push and pull operations under the assumption that these other measures are used. Subsequently, Section 3.8 evaluates the effectiveness of Drum's other DoS-mitigation concepts, by contrasting Drum's performance against that of two modified versions of Drum: one without resource separation, and a second without using random ports.

We begin by evaluating the effect that a range of DoS attacks have on message latency using asymptotic mathematical analysis (in Section 3.5) and simulations (in Section 3.6). Our simulation results exhibit the trends predicted by the analysis.

For these evaluations, we make some simplifying assumptions: We assume no message is ever purged from any process's message buffer, and that all processes have some messages in their buffers (from previous multicast sessions). We also assume that when processes send a data message, they send the complete contents of their buffer in a single operation. We model the push operation as performed without push-offers (in Drum and in Push). We assume that the rounds are synchronized, and that the message-delivery latency is smaller than half the gossip period; thus, a process that sends a pull-request receives the pull-reply in the same round. All of these assumptions were made in previous analyses of gossip-based protocols, e.g., [8, 14, 35, 38].

The analysis and simulations measure latency in terms of gossip rounds: we measure the message's *propagation time*, which is the expected number of rounds it takes a given protocol to propagate a message to all (in the closed-form analysis) or to 99% (in the simulations) of the correct processes. We chose a threshold of 99% since the message may fail to reach some of the correct processes due to old-message purging or link loss. Note that correct processes can be either attacked or non-attacked. In both cases, they should be able to send and receive data messages.

We turn to measure actual performance on a cluster of workstations (in Section 3.7). Our goal for this evaluation is twofold: First, we wish to ensure that the simplifying assumptions made in the analysis and simulations have little impact on their results. E.g., in the implementation, rounds are not synchronized and the push-offer mechanism is used (in Drum and in Push). Second, we seek to measure the consequences of DoS attacks not only on actual latency (in msecs.), but also on the

throughput of a real system, where multiple messages are sent, and old messages are purged from processes' message buffers.

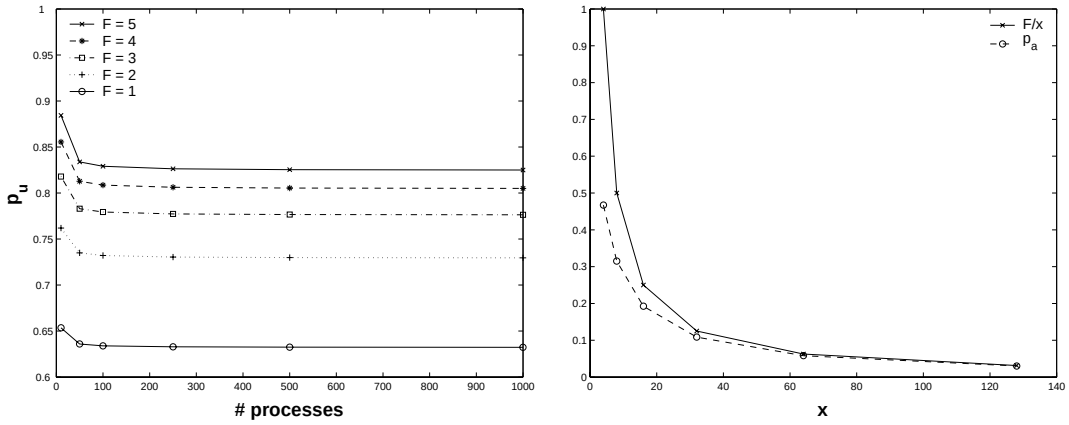
Attacks. In all of our evaluations, we stage various DoS attacks. We assume that the DoS attacks are launched from outside the system. DoS from inside the group is essentially just one source (or more) generating excessive traffic. This can happen regardless of any malicious nodes being part of the multicast group, e.g., in a heterogenous system. Consequently, this is in fact a flow-control problem, as one cannot differentiate between a malicious attack and legitimate excessive traffic. Flow control in gossip-based multicast has been dealt with in [47].

In each DoS attack, the adversary focuses on a fraction α of the processes ($0 < \alpha \leq 1$), and sends each of them x fabricated messages per round (in Drum, this means $\frac{x}{2}$ push messages and $\frac{x}{2}$ pull-requests). We note that randomly choosing the attack targets every round does not make any difference, as the communication partners are re-chosen uniformly at random each round. We denote the total attack strength by $B = x \cdot \alpha \cdot n$. We assume that the message source is being attacked (this has no impact on the results of Push). We consider attacks either of a *fixed strength*, where B is fixed and α increases (thus, x decreases); or of *increasing strength*, where either x is fixed and α increases, or vice versa (in both cases, B increases). Examining fixed strength attacks allows us to identify protocol vulnerabilities, e.g., whether an adversary can benefit from targeting a subset of the processes. Increasing strength attacks enable us to assess the protocols' performance degradation due to an increasing attack intensity.

3.5 Asymptotic Closed-Form Analysis

In this section we assume that all the processes are correct. The protocols use a constant fan-out, F. Every round, each process sends messages to F processes and accepts messages from at most F processes. In Drum, F is equally divided between push and pull, e.g., if $F = 4$, then $view_{push} = view_{pull} = 2$, and each process accepts push messages from at most 2 processes and pull-request messages from at most 2 processes in a round. We analyze Drum in Section 3.5.1, Push in Section 3.5.2, and Pull in Section 3.5.3.

We denote by p_u the probability of a non-attacked process to accept a valid incoming push or pull-request message sent to it. Similarly, we denote by p_a the probability of an attacked process to accept a valid incoming message. Obviously, p_u is independent of the attack strength. In Section 3.9, we give detailed formulas for p_a and p_u, and Lemma 8 proves that $p_u > 0.6$ for all $F \geq 3$. Numerical calculations using the formula in Section 3.9 show that $p_u > 0.6$ for all $F \geq 1$, as can be seen in Figure 3.1(a). When at least one valid message is sent, an attacked process gets at least $x + 1$ messages in a round, and accepts at most F of them. We get the following coarse bound: $p_a < \frac{F}{x}$. Figure 3.1(b) shows an example of the numerical calculation of p_a versus $\frac{F}{x}$.



(a) p_u as a function of n, for various fan-outs.

(b) p_a vs. $\frac{F}{x}$ for $F = 4$, $n = 1000$.

Figure 3.1: Actual values of p_u and p_a.

3.5.1 Drum

We begin by considering increasing strength attacks. We show that in Drum, an adversary does not gain any significant advantage by increasing its attack strength while focusing on a fixed strict subset of the processes.

Lemma 1 *Fix $\alpha < 1$ and n. Drum's expected propagation time is bounded from above by a constant independent of x.*

Proof: Since $\alpha < 1$, some processes are not attacked at all. Let us look at a two-stage propagation scheme that works as follows: At the first stage, only the source propagates the message. The expected propagation time from the source via push to all the non-attacked processes is independent of x and bounded, since n is fixed. At the next stage, the non-attacked processes constitute non-attacked sources for the rest of the group via pull. The expected propagation time of the second stage is again independent of x and bounded. Since n is fixed, this two-stage expected propagation time is constant. The two-stage propagation from the source to all of the destinations is obviously not faster than Drum's propagation. Thus, Drum's expected propagation time is bounded from above by a constant independent of x. □

Figure 3.3(a) in Section 3.6.2 illustrates this quality of Drum, using simulations.

We now consider attacks where the adversary has a fixed attacking power. Thus, the attacker can intensely attack a small group of processes, or perform a moderate attack on a large number of processes. We would like to see which strategy is more beneficial to the attacker. We denote by

$c = \frac{B}{F \cdot n} = \frac{\alpha x}{F}$ the attack strength divided by the total system capacity. We show that the adversary's best strategy against Drum is to attack as many processes as it can, i.e., increase α.

We define the *effective expected fan-in*, *I*, to be the average number of valid data messages a process successfully receives in a round. (If the same data message is received from k processes, we count this as k messages.) Likewise, the *effective expected fan-out*, *O*, is the average number of messages that a process sends and are successfully received by their targets in a round.

Let us examine the effect of a DoS attack on O and I, with respect to the push operation (O_{push} and I_{push}, resp.). The probability of an attacked process to receive a push message is p_a. The probability of a non-attacked process to receive a push message is p_u. Therefore, the effective fan-ins I^a_{push} and I^u_{push} of an attacked and non-attacked process (resp.) are:

$$I^a_{push} = F \cdot p_a \quad \text{and} \quad I^u_{push} = F \cdot p_u \tag{3.1}$$

When αn processes are attacked, the effective fan-outs are:

$$O^a_{push} = O^u_{push} = F \cdot (\alpha \cdot p_a + (1 - \alpha) \cdot p_u) \tag{3.2}$$

Similar arguments apply for the pull operation. The probability of an attacked process to receive a pull-request is p_a. The same probability for a non-attacked process is p_u. Receiving pull-requests allows a process to send data messages, and on average, each process receives F pull-requests. Due to the use of random ports, we assume that each pull-reply is actually being received, and thus, the effective fan-outs are:

$$O^a_{pull} = \quad F \cdot p_a \quad \text{and} \qquad O^u_{pull} = F \cdot p_u \tag{3.3}$$

Receiving data messages requires sending pull-requests. Each round, F pull-requests are being sent. On average, αF of them reach an attacked process and are successfully read with probability p_a, and $(1 - \alpha)F$ of those reach a non-attacked process and are successfully read with probability p_u. Due to the use of random ports, we can assume it makes no difference whether the requesting process is attacked or not. We get the following fan-ins:

$$I^a_{pull} = \quad I^u_{pull} = \quad F \cdot (\alpha \cdot p_a + (1 - \alpha) \cdot p_u) \tag{3.4}$$

In Drum, $O = \frac{1}{2}(O_{push} + O_{pull})$ and $I = \frac{1}{2}(I_{push} + I_{pull})$. Therefore:

$$O^a = I^a = \quad \tfrac{F}{2} \cdot (\alpha \cdot p_a + (1 - \alpha)p_u + p_a) \quad = F \cdot (\frac{\alpha + 1}{2} \cdot p_a + \frac{1 - \alpha}{2} \cdot p_u) \tag{3.5}$$

$$O^u = I^u = \quad \tfrac{F}{2} \cdot (\alpha \cdot p_a + (1 - \alpha)p_u + p_u) \quad = F \cdot (\frac{\alpha}{2} \cdot p_a + \frac{2 - \alpha}{2} \cdot p_u) \tag{3.6}$$

Lemma 2 *For $c > 5$, Drum's expected propagation time is monotonically increasing with α.*

Proof: We will show that all the processes' effective fan-ins and fan-outs are monotonically decreasing with α. That is, we want to prove that: $\frac{\mathrm{d}O^a}{\mathrm{d}\alpha} < 0$ and $\frac{\mathrm{d}O^u}{\mathrm{d}\alpha} < 0$. We require the following:

$$\frac{\mathrm{d}O^a}{\mathrm{d}\alpha} = \frac{\mathrm{d}I^a}{\mathrm{d}\alpha} = \frac{F}{2} \cdot \left(p_a + \alpha \frac{\mathrm{d}p_a}{\mathrm{d}\alpha} + \frac{\mathrm{d}p_a}{\mathrm{d}\alpha} - p_u \right) < 0$$
$$p_a + (\alpha + 1)\frac{\mathrm{d}p_a}{\mathrm{d}\alpha} < p_u$$

Recall that $p_a < \frac{F}{x}$. In Lemma 7 in Section 3.9 we show that $\frac{\mathrm{d}p_a}{\mathrm{d}\alpha} < \frac{F}{\alpha x}$. Bounding the left side of the inequality, we get:

$$p_a + (\alpha + 1)\frac{\mathrm{d}p_a}{\mathrm{d}\alpha} < \frac{F}{x} + (\alpha + 1)\frac{F}{\alpha x} = \frac{F}{\alpha x} \cdot (\alpha + \alpha + 1) = \frac{2\alpha + 1}{c} < \frac{3}{c}$$

Thus, our condition holds when $\frac{3}{c} < p_u$, that is, when $c > \frac{3}{p_u}$. Similarly, when applying the derivative to the second term we get the condition:

$$\frac{\mathrm{d}O^u}{\mathrm{d}\alpha} = \frac{\mathrm{d}I^u}{\mathrm{d}\alpha} = \frac{F}{2} \cdot \left(p_a + \alpha \frac{\mathrm{d}p_a}{\mathrm{d}\alpha} - p_u \right) < 0$$
$$p_a + \alpha \frac{\mathrm{d}p_a}{\mathrm{d}\alpha} < p_u$$

Bounding the left side of the inequality, we get:

$$p_a + \alpha \frac{\mathrm{d}p_a}{\mathrm{d}\alpha} < \frac{F}{x} + \alpha \frac{F}{\alpha x} = \frac{F}{\alpha x} \cdot (\alpha + \alpha) = \frac{2\alpha}{c} < \frac{2}{c}$$

Thus, we require that $\frac{2}{c} < p_u$, or that $c > \frac{2}{p_u}$. This is already inferred from our previous result. The lemma follows since $p_u > 0.6$. □

This behavior is validated in the simulations in Section 3.6.3. Moreover, the simulations show that even for much smaller values of c (ranging from 0.25 to 2), Drum's propagation time increases with α (see Figures 3.7–3.8).

3.5.2 Push

We first prove the following simple lemma.

Lemma 3 $\forall a > 0 \quad a < \frac{1}{\ln(1+\frac{1}{a})} < a + 1$.

Proof: We show that $\forall y > 0 \quad \frac{1}{y} < \frac{1}{\ln(1+y)} < \frac{1}{y} + 1$.
Define $h(y) = \ln(1 + y) - \frac{y}{1+y}$ and $g(y) = \ln(1 + y) - y$. By taking derivatives we get:

$$h'(y) = \frac{1}{1+y} - (\frac{1}{1+y} - \frac{y}{(y+1)^2}) = \frac{y}{(y+1)^2} > 0, \ \ \forall y > 0,$$
$$g'(y) = \frac{1}{1+y} - 1 < 0, \ \ \forall y > 0.$$

Since $h(0) = g(0) = 0$, $y > \ln(1 + y) > \frac{y}{(y+1)}$. Therefore, $\frac{1}{y} < \frac{1}{\ln(1+y)} < \frac{1}{y} + 1$. □

We proceed to show that Push's propagation time is linear in x.

Lemma 4 *The expected propagation time to all processes in Push is bounded from below by:*

$$\frac{\ln n - \ln\left[(1-\alpha)\, n + 1\right]}{\ln\left(1 + F\alpha p_a\right)}$$

Proof: We prove that the given bound holds even for the case where initially all the non-attacked processes have the message (denoted by *M*), in addition to the source (which is attacked). The lemma then follows immediately.

Let the random variable $M(k)$ denote the number of processes that have *M* at the beginning of round k, and let $E\left[M(k)\right]$ denote its expectation. In round k, each process having M sends it to F other processes. On average, $F\alpha$ of those are attacked, and each attacked process receives the message with probability p_a. Thus, we get the coarse recursive bound $E\left[M(k+1)\right] \leq E\left[M(k)\right] + E\left[M(k)\right] \cdot F\alpha p_a$ with the initial condition $E\left[M(0)\right] = M(0) = (1-\alpha)n + 1$. Thus, $E\left[M(k)\right] \leq \left[(1-\alpha)\, n + 1\right]\left(1 + F\alpha p_a\right)^k$. *M* reaches all the processes when $E\left[M(k)\right] \geq n$. To bound k from below we use the fact that having $\left[(1-\alpha)\, n + 1\right]\left(1 + F\alpha p_a\right)^k < n$ implies that $E\left[M(k)\right] < n$. Thus, the first round number k that may satisfy the inequality $E\left[M(k)\right] \geq n$ is the required formula. □

Corollary 1 *Fix $\alpha > 0$ and $n > \frac{1}{\alpha}$. The propagation time of Push increases at least linearly with x.*

Proof: Since α and $n > \frac{1}{\alpha}$ are fixed, the numerator in Lemma 4 is a positive constant. Consider the denominator: since $p_a < \frac{F}{x}$, it holds that $F \cdot \alpha \cdot p_a$ is $O(\frac{1}{x})$. The lemma follows since, by Lemma 3, $\frac{1}{\ln(1+\frac{1}{x})}$ is $\Theta(x)$. □

The above corollary explains the trend exhibited by Push in Figure 3.3(a).

3.5.3 Pull

We begin by proving the following lemma.

Lemma 5 $\forall b \in \mathbb{N}$ $\frac{x^b}{x^b-(x-F)^b}$ *is* $\Omega(x)$.

Proof: We first show that $\frac{a-1}{b} \leq \frac{a^b}{a^b-(a-1)^b}$ for every $a > 1$, $b \in \mathbb{N}$.

We prove by induction on b that $\frac{b}{a-1} \geq \frac{a^b-(a-1)^b}{a^b}$. For $b = 1$, $\frac{1}{a-1} \geq \frac{1}{a}$ for every $a > 1$. The inductive Step: $\frac{a^{b+1}-(a-1)^{b+1}}{a^{b+1}} = \frac{a(a)^b-(a-1)(a-1)^b}{a(a)^b} = \frac{a^b}{a(a)^b} + \frac{a-1}{a}\frac{a^b-(a-1)^b}{a^b} \leq \frac{1}{a} + \frac{a-1}{a}\frac{b}{a-1} = \frac{1}{a} + \frac{b}{a} = \frac{b+1}{a} \leq \frac{b+1}{a-1}$.

By substituting $\frac{x}{F}$ for a in the proven inequality, we get that $\frac{x-F}{bF} \leq \frac{x^b}{x^b-(x-F)^b}$ for every $x > F$. Therefore, $\frac{x^b}{x^b-(x-F)^b}$ is $\Omega(x)$. □

We define $\widetilde{p}$ as probability that the message M is propagated from the source in a round.

Lemma 6 *Fix α and n. The number of rounds it takes a message to leave the source in Pull grows at least linearly with x.*

Proof: We give a gross over-estimate of $\widetilde{p}$ by assuming that all the other $n-1$ processes choose the source every round. (When fewer processes choose the source, M is *less* likely to leave the source.) Since $p_a < \frac{F}{x}$, $\widetilde{p} < (1 - (\frac{x-F}{x})^{n-1})$. The number of rounds it takes to propagate a message beyond the message source is geometrically distributed with $\widetilde{p}$. Therefore, its expectation is $\frac{1}{\widetilde{p}} > \frac{x^{n-1}}{x^{n-1}-(x-F)^{n-1}}$. Substituting $n-1$ for b in Lemma 5, we get that $\frac{1}{\widetilde{p}}$ is $\Omega(x)$. □

Corollary 2 *Fix α and n. The propagation time of Pull grows at least linearly with x.*

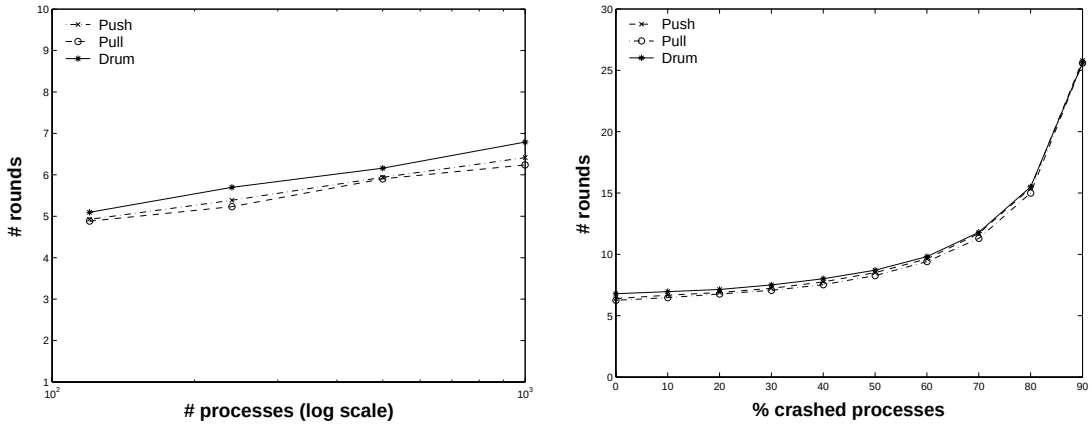
Figure 3.3(a) illustrates this behavior of Pull.

3.6 Simulation Results

This section presents MATLAB simulations of the three protocols under various DoS attack scenarios. All group members constantly have messages to send, and we track the propagation of one of these messages, M, from its source. Each process receives messages from at most $F = 4$ other processes each round (disregarding pull-replies). If more than F processes try to access this process's incoming channels, a random F-sized subset of them is chosen. We consider a link-loss probability of 0.01 on all links and a fan-out of $F = 4$. Rounds are synchronized among all processes. Each data point is averaged over 1000 runs, where in each run the number of rounds it takes the message to reach 99% of the processes is measured.

In Section 3.6.1 we consider situations with no DoS attack (either no failures or only crash failures), and validate known results about gossip protocols. We continue in Sections 3.6.2 and 3.6.3 by measuring the effect of DoS attacks on the system. In these studies, we assume that 10% of the processes have crashed when the system started (we assume that no failure detectors are being used), and that the DoS attack is launched from outside the system. Since we do not assume that nodes can detect that their gossip partners are down, assuming that nodes crash right when the system starts has no special effect on the results. If nodes crash later on, the system will operate as usual until the processes crash. After that, the system will operate as analyzed with processes that have crashed right from the start.

We measure the propagation times to the correct processes, both attacked and non-attacked. In Section 3.6.2 we measure the impact of targeted DoS attacks, and in Section 3.6.3 we examine fixed strength attacks and adversary strategies.



(a) Failure-free operation.

(b) Operation with crashed processes, $n = 1000$.

Figure 3.2: Runs without DoS attack: Average propagation time to 99% of the correct processes (simulations).

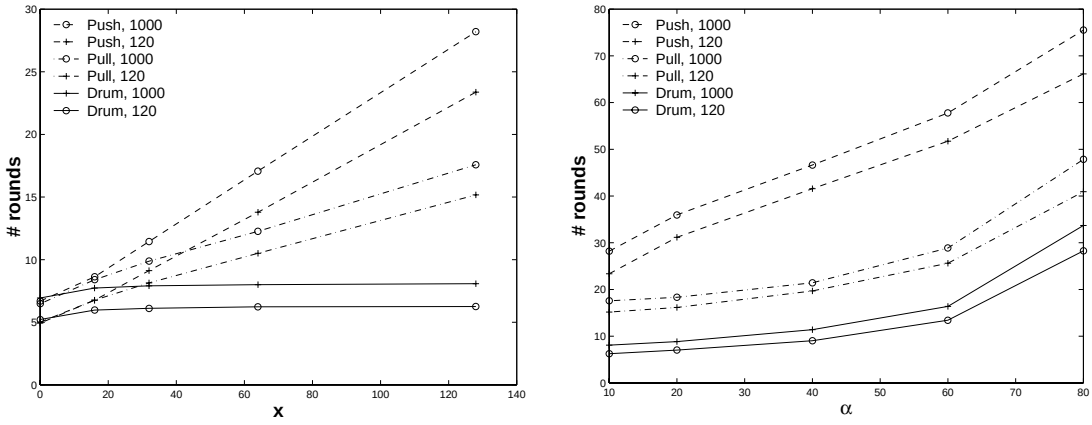
3.6.1 Validating Known Results

We begin by evaluating the three protocols in a failure-free scenario, and in situations where crash failures occur. We assume that the crashes occur before M is generated, and that the source does not crash. We also assume that the crashes are not detected by the correct processes, i.e., they try to gossip with crashed processes as well.

Our aim is to validate two known results: (1) the propagation time of gossip-based multicast protocols is $O(\log n)$ [44, 25], as can be seen in Figure 3.2(a), with a logarithmic x-axis; and (2) the performance of such protocols degrades gracefully as crash failures amount [20, 32], as depicted in Figure 3.2(b)). We can see that Push and Pull slightly outperform Drum in these experiments. This is due to the fact that the bounds on the pull and push channels in Drum are strict, i.e., even if in a specific round no messages have arrived via the push channels, only requests from at most two distinct processes will be handled, although the process is capable of handling four such requests. Conversely, Push and Pull have only one bound, which guarantees that messages won't be discarded if they can be processed. The ability to perform well even when many processes crash stems from the random choice of communication partners each round.

3.6.2 Targeted DoS Attacks

In this section we consider targeted attacks, where a subset of size αn of the processes is attacked. Figure 3.3 compares the time it takes *M* to reach 99% of the correct processes for the three protocols under various DoS attacks, with 120 and 1000 processes. Figure 3.3(a) shows that when 10% of



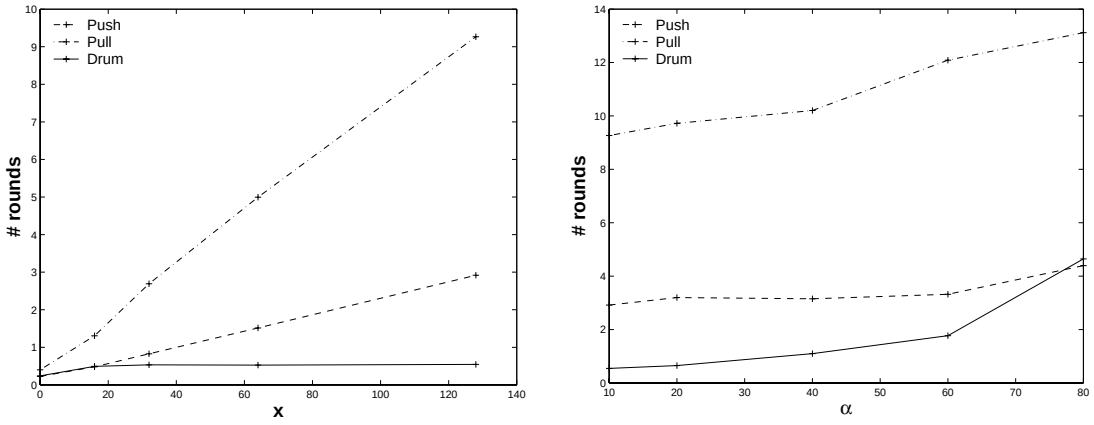
(a) $\alpha = 10\%$. (b) $x = 128$.

Figure 3.3: Increasing attack strength: Average propagation time to 99% of the correct processes, $n = 120, 1000$ (simulations).

the processes are attacked, the propagation time of both Push and Pull increases linearly with the severity of the attack, while Drum's propagation time is unaffected by the attack strength. This is consistent with the prediction of Lemma 1 and Corollaries 1 and 2. Moreover, the three protocols perform virtually the same without DoS attacks (see the leftmost data point). Figure 3.3(b) illustrates the propagation time as the percentage of attacked processes (and thus B) increases. The rightmost data point in this figure matches a scenario where only 10% of the processes are both correct non-attacked. Although the protocols exhibit similar trends, Drum propagates messages much faster than Push and Pull.

Figure 3.4 illustrates the *standard deviation* (STD) of the propagation times presented in Figure 3.3 for $n = 1000$. It shows that for a fixed α, Drum's STD is not affected by the attack strength, whereas the other protocols' STD increases linearly. Furthermore, both Drum and Push exhibit a small STD compared to Pull. E.g., for $\alpha = 10\%$ and $x = 128$, the STDs of Drum and Push are 0.5 and 2.9 rounds (resp.), whereas Pull's STD is 9.3 rounds. Therefore, the behavior of Drum and Push is more predictable. The high STD of Pull's propagation time is mainly due to the large STD of the number of rounds it takes to propagate *M* beyond the source. The number of rounds it takes to propagate *M* beyond the source is geometrically distributed with $\widetilde{p}$, where $\widetilde{p}$ is the probability to propagate *M* beyond the source in a round. Thus, the STD of the number of rounds it takes to propagate *M* beyond the source is $\frac{\sqrt{1-\widetilde{p}}}{\widetilde{p}}$. A numerical calculation of $\widetilde{p}$ according to the formula in Section 3.10, with $F = 4$ and $x = 128$, yields an STD of 8.17 rounds, which explains Pull's measured STD of 9.3 rounds mentioned above.

Figure 3.5 illustrates the cumulative distribution function (CDF) of the percentage of correct



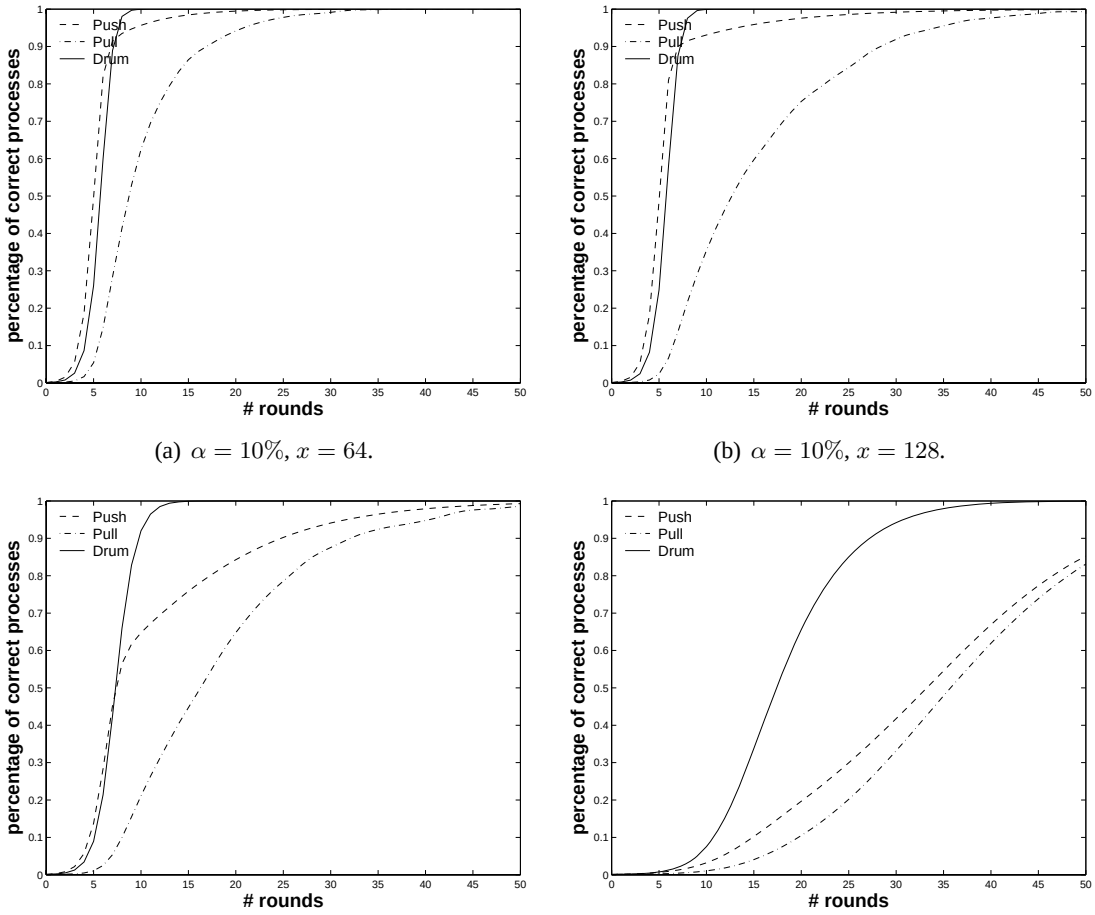
(a) $\alpha = 10\%$. (b) $x = 128$.

Figure 3.4: Increasing attack strength: STD of the propagation time to 99% of the correct processes, $n = 1000$ (simulations).

processes that receive *M* by a given round, under different DoS attacks. As expected, Push propagates *M* to the non-attacked processes very quickly, but takes much longer to propagate it to the attacked processes. Again, we see that Drum significantly outperforms both Push and Pull when a strict subset of the system is attacked.

Interestingly, on average, Push propagates *M* to more processes per round than Pull does (see Figure 3.5), although the average number of rounds Pull takes to propagate M to 99% of the correct processes is smaller than that of Push (see Figure 3.3). This paradox occurs since, with Pull, there is a non-negligible probability that *M* is delayed at the source for a long time. With $F = 4$ and $x = 128$, the probability of *M* not being propagated beyond the source in 5, 10, and 15 rounds is 0.54, 0.3, and 0.16 resp. (as computed using the formula for $\widetilde{p}$ in Section 3.10). Once M reaches one non-attacked process, it quickly propagates to the rest of the processes. Therefore, even if by a certain round k, in most runs, a large percentage of the processes have M, there is still a non-negligible number of runs in which Pull does not reach *any* process (other than the source) by round k. This large difference in the percentage of processes reached has a significant impact on the average depicted in Figure 3.5. In contrast, Push, which reaches all the non-attacked processes quickly in all runs, does not have runs with such low percentages factoring into this average. Nevertheless, Push's average propagation time to 99% of the correct processes is much higher than Pull's, because Push has to propagate M to *all* the attacked processes, whereas Pull has to propagate M only out of one attacked process.

Figure 3.6 illustrates this behavior: Figure 3.6(a) shows that Push propagates M much faster than Pull to the non-attacked processes, while Figure 3.6(b) indicates that Push and Pull take the



(a) $\alpha = 10\%$, $x = 64$.

(b) $\alpha = 10\%$, $x = 128$.

(c) $\alpha = 40\%$, $x = 128$.

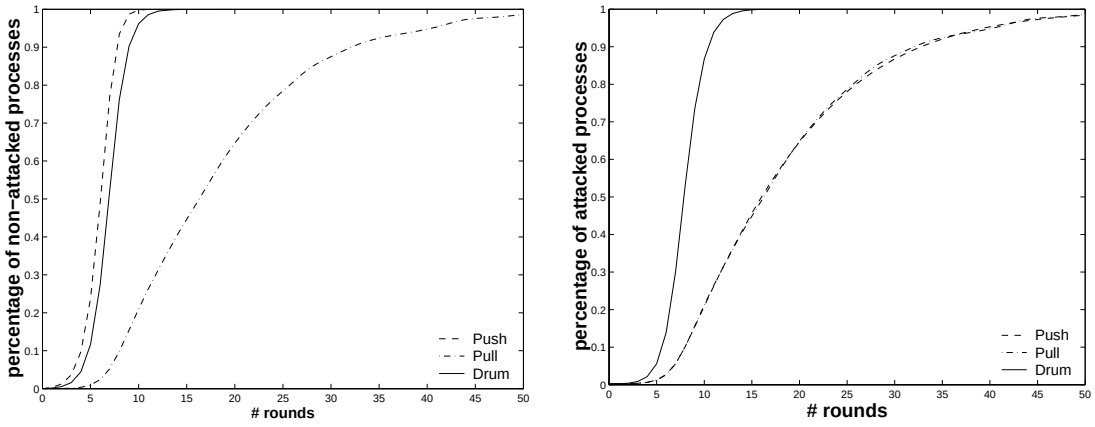
(d) $\alpha = 80\%$, $x = 128$.

Figure 3.5: Targeted DoS attacks: CDF: Average percentage of correct processes that receive *M*, $n = 1000$ (simulations).

same time to propagate M to the attacked processes. Conversely, Drum exhibits fast propagation times both to attacked and non-attacked processes.

3.6.3 Adversary Strategies

We now evaluate the protocols under a range of attacks with fixed adversary strengths. First, we consider severe attacks with $B = 7.2n$ and $B = 36n$ (corresponding to $c = 2$ and $c = 10$, resp.) fabricated messages per round. If the adversary chooses to attack all correct processes, it can send 8 (resp., 40) fabricated messages to each of them in each round, because 90% of the processes are correct. If the adversary instead focuses on 10% of the processes, it can send 72 (resp., 360)



(a) Non-attacked processes. (b) Attacked processes.

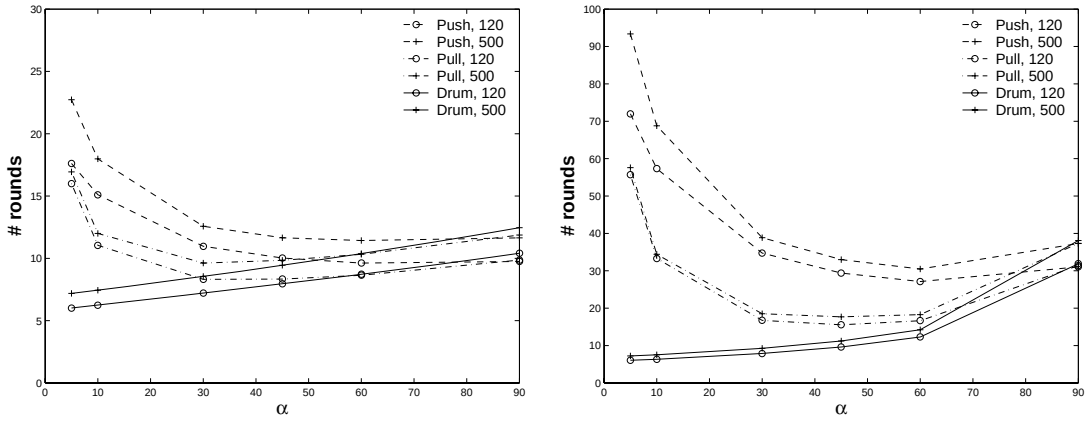
Figure 3.6: Propagation to attacked vs. non-attacked processes: CDF: Average percentage of attacked versus non-attacked processes that receive *M*, $n = 1000$, $\alpha = 40\%$, $x = 128$ (simulations).

fabricated messages per round to each of them. Figure 3.7 illustrates the protocols' propagation times with different percentages of attacked processes, for system sizes of 120 and 500. It validates the prediction of Lemma 2, and shows that the most damaging adversary strategy against Drum is to attack all the correct processes. That is, an adversary cannot "benefit" from focusing its capacity on a small subset of the processes. In contrast, the performance of Push and Pull is seriously hampered when a small subset of the processes is targeted. Not surprisingly, the three protocols perform equally when all correct processes are targeted (see the rightmost data point).

Next, we evaluate Drum under attacks with relatively small adversary powers of $B = 0.9n$, $B = 1.8n$ and $B = 3.6n$ ($c = 0.25$, $c = 0.5$, and $c = 1$, resp.) and also without an attack (as a baseline). As Figure 3.8 shows, such attacks have little impact on Drum's propagation time.

3.7 Implementation and Measurements

We have implemented Drum, Push, and Pull in Java. The implementations are multithreaded. The operations that occur in a round are not synchronized, e.g., one process might send messages before trying to receive messages in that round, while another might first receive a new message, and then propagate it. We run our experiments on 50 machines at the Emulab testbed [59], on a 100Mbit LAN, where a single process is run on each machine (i.e., $n = 50$). As in the simulations, 10% of the processes have crashed when the system started (these crashes go undetected), and the DoS attack is launched from outside the system. Since we do not have a router/firewall that randomly



(a) $B = 7.2n$ ($c = 2$). (b) $B = 36n$ ($c = 10$).

Figure 3.7: Strong fixed strength attacks: Average propagation time to 99% of the correct processes (simulations).

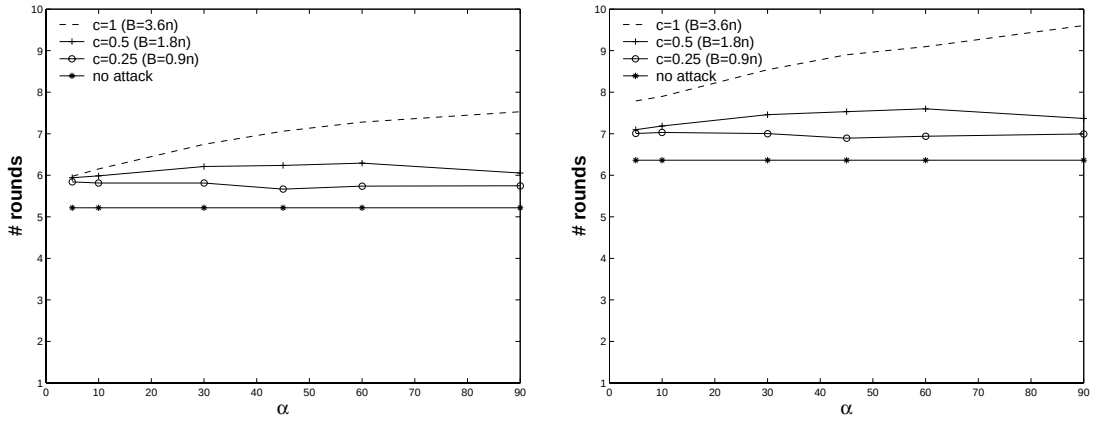
selects messages according to the protocol's needs, we have implemented the selection of messages by sequentially reading messages from the port at random times within the round, and discarding all messages at the end of the round. Since rounds are locally controlled and randomly vary in duration, the attacker cannot "aim" its messages for the beginning of a round.

3.7.1 Validating the Simulation Methodology

Our first goal for these experiments is to validate the simulation methodology. To this end, we experiment with the same settings that were tested in Section 3.6, first for increasing values of x and $\alpha = 10\%$, and then for $x = 128$ and increasing values of α. As in the simulations, every process has messages to send, and we track the propagation of one of those messages. Each data point is averaged over 1000 runs, again, as in the simulations.

Due to the lack of synchronization, messages can be propagated multiple hops in a single round in some situations. We use the following method to count the number of rounds it takes to propagate a message: when a message is created, a round counter is attached to it and initialized to 0. The message source logs the value 0, and immediately increases the round counter to 1. Whenever a process receives a new message, it logs the message's current round counter. Every round, each process increments the round counters of all the messages in its local buffer.

Figure 3.9 depicts the results of these experiments, and compares them with the corresponding simulation results. It shows that the experimental results are consistent with the simulation results, indicating that the simplifying assumptions made in the analysis and simulations have negligible effect on the results.



(a) $n = 120$. (b) $n = 500$.

Figure 3.8: Weak fixed strength attacks: Drum, average propagation time to 99% of the correct processes (simulations).

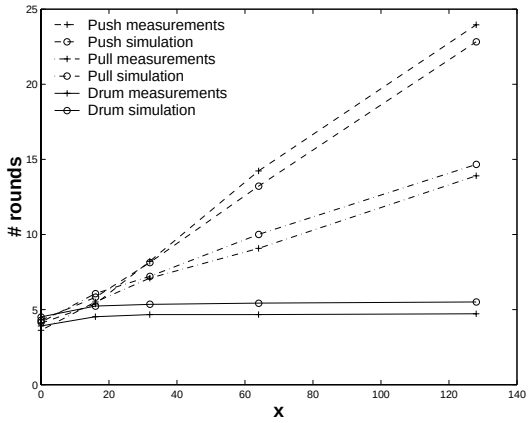
3.7.2 High Throughput Experiments

We proceed to evaluate the protocols in a realistic setting, where multiple messages are sent, and old messages are purged from processes' buffers. By running on a real network, we can faithfully evaluate latency in milliseconds (instead of rounds), as well as throughput.

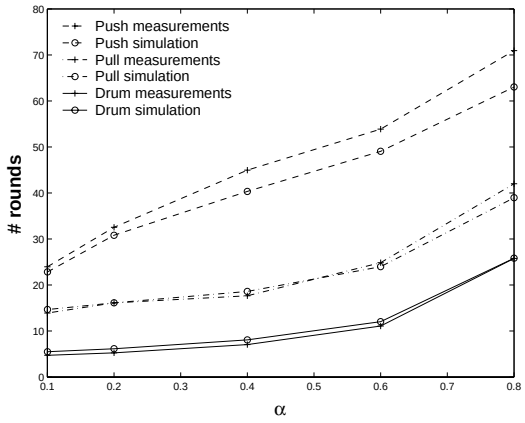
In each experiment scenario, a total of $10{,}000$ messages are sent by a single source, at a rate of 40 messages per second. The average received throughput and latency are measured at the remaining 44 correct processes (recall that 5 of the 50 processes are faulty). The average throughput is calculated ignoring the first and last 5% of the time of each experiment. The round duration is 1 second. Data messages are 50 bytes long. (The evaluation in [14] used a similar transmission rate and similar message sizes.)

In a practical system, messages cannot reside in local buffers forever, nor can a process send all the messages it ever received in a single round. In our experiments, messages are purged from processes' buffers after 10 rounds, and each process sends at most 80 randomly chosen *new* messages to each of its gossip partners in a round. These are roughly twice the buffer size and sending rate required for the throughput of 40 messages per round in an ideal attack-free setting, since the propagation time in the absence of an attack is about 5 rounds. Due to purging, some messages may fail to reach all the processes. Since we measure throughput at the receiving end, this is reflected by an average throughput lower than the transmission rate (of 40 messages per second).

Figure 3.10 shows the throughput at the receiving processes for Drum, Push, and Pull, under the same DoS attack scenarios staged above. Figure 3.10(a) indicates that, as for latency, Drum's

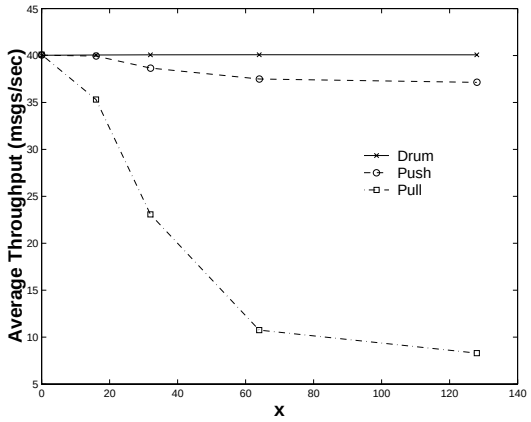


(a) $\alpha = 10\%$.

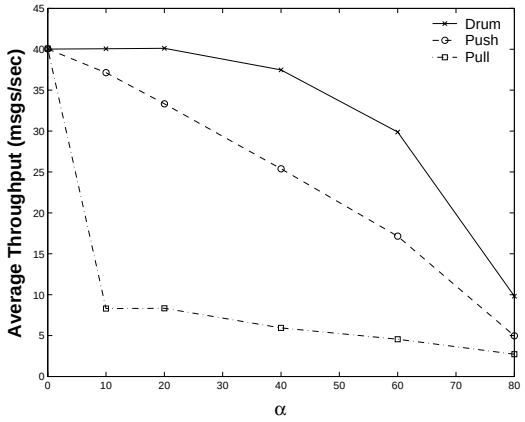


(b) $x = 128$.

Figure 3.9: Simulations vs. measurements: Average propagation time to 99% of the correct processes, $n = 50$.

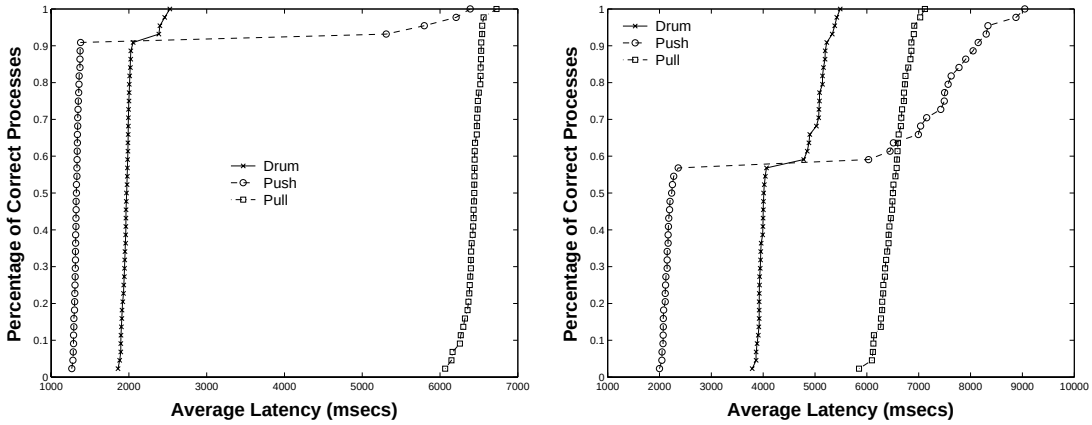


(a) $\alpha = 10\%$.



(b) $x = 128$.

Figure 3.10: Increasing attack strength: Average received throughput (measurements).



(a) $\alpha = 10\%$, $x = 128$. (b) $\alpha = 40\%$, $x = 128$.

Figure 3.11: CDF: average latency of received messages (measurements).

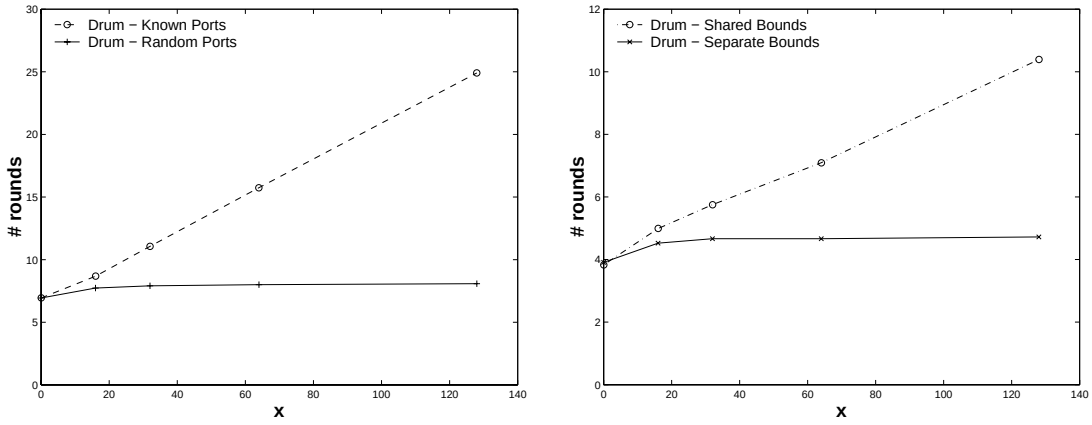
throughput is also unaffected by increasing x, while Push shows a slight degradation of throughput, and Pull's throughput decreases dramatically. Figure 3.10(b) shows that Drum's throughput gracefully degrades as α increases, while Push exhibits a linear degradation, and Pull's throughput is drastically affected for every $\alpha > 0$.

Figure 3.11 depicts the CDF of the average latency of *successfully received* messages in two scenarios. Each data point shows, for a given latency l, the percentage of correct processes for which the average latency does not exceed l. We observe that Push is the fastest in delivering messages to non-attacked processes, but suffers from substantial variation in delivery latency, as messages take a long time to reach the attacked processes. E.g., Figure 3.11(a) shows that the 4 attacked processes (other than the source) measure an average latency 4 times longer than non-attacked processes. While Pull exhibits almost the same average latency for all the processes, this latency is very long. Drum combines the best of Push and Pull: it delivers messages almost as fast as Push, while maintaining a small variation between attacked and non-attacked processes.

3.8 Other DoS-Mitigation Methods

Until now, we have evaluated the advantage of combining both the push and pull techniques as a way to mitigate DoS attacks, in the context of a protocol that also employs resource bounds and random ports. We now turn to examine the importance of using the other two techniques: utilizing random ports whenever possible, and allocating separate resources for orthogonal operations.

In order to evaluate the effectiveness of random ports, we simulate Drum as described in Section 3.6, with the difference that pull-replies are sent to a well-known port instead of to a random



(a) Random ports, $n = 1000$ (simulations). (b) Separate bounds, $n = 50$ (measurements).

Figure 3.12: The effect of random ports and separate bounds on Drum's performance, $\alpha = 10\%$.

one. The adversary attacks this port by equally dividing its attack strength for the pull channels between the pull-request port and the pull-reply port (i.e., each pull port is attacked with a quarter of the total attack strength). Figure 3.12(a) presents simulation results comparing Drum's performance with and without the use of random ports, when 10% of the processes are attacked. The results show a linear increase in propagation time for the well-known ports variation of Drum, as the rate of bogus messages each attacked process receives in a round increases. This is in contrast to the propagation time of Drum using random ports, which is bounded by a constant.

When solely using well-known ports, the adversary can attack both pull ports, as well as the push port. A process under attack experiences difficulty receiving messages both via push and through the pull channels, since the push and pull-reply ports are attacked. The same process's ability to send messages is only partly hampered. Although the pull-request port is attacked, the adversary cannot directly affect the process's outgoing push channels.

Next, we measure the effect of resource separation on Drum's performance. To this end, we change Drum's implementation detailed in Section 3.7. Resources are now combined (i.e., a joint bound on the maximum number of processed messages per round is used) for receiving control messages: pull-requests, push-offers, and push-replies. We do not include the reception of data messages in this bound, since this bound may differ greatly from the bound on control messages in actual scenarios. Figure 3.12(b) contrasts the measurements of Drum's propagation time with shared bounds against those with separate bounds, when 10% of the processes are attacked. The results indicate a linear degradation of performance as the attack rate increases, when bounds are shared. On the other hand, the unmodified version of Drum is virtually indifferent to the increase in attack strength.

Shared bounds degrade Drum's performance under a DoS attack, since the fabricated control messages sent by the adversary to the well-known push-offer and pull-request ports consume resources that should be used for reading pull-requests, push-offers, and push-replies. The valid control messages are then discarded when resources are exhausted, and the attacked process becomes less responsive.

We conclude that random ports and separate resource bounds are crucial to Drum's ability to cope with DoS attacks.

3.9 Calculating p_u and p_a

Suppose process p_i sends a message to process p_j, we want to calculate the probability that process p_j accepts this message. Denote the event "process p_i sends a message to process p_j" by S_{ij}. Assume $n > F$, and define q as the probability that process p_j appears in process p_i's *view*, then:

$$q = 1 - \frac{n-2}{n-1} \cdot \frac{n-3}{n-2} \cdots \frac{n-1-F}{n-F} = 1 - \frac{n-1-F}{n-1} = \frac{F}{n-1}$$

Let Y be the number of valid messages received by p_j in a single round, then:

$$\begin{array}{rrcl} & \Pr(Y \leq 0 \mid S_{ij}) & = & \Pr(Y \geq n \mid S_{ij}) = 0 \\ 0 < y < n & \Pr(Y = y \mid S_{ij}) & = & \binom{n-2}{y-1} q^{y-1} (1-q)^{n-1-y} \end{array}$$

Let p_Y be the probability that a non-attacked process, p_j, discards the message sent by p_i, given S_{ij}, then:

$$p_Y = \begin{cases} 0 & Y \leq F \\ \frac{Y-1}{Y} \cdot \frac{Y-2}{Y-1} \cdots \frac{Y-F}{Y-F+1} = \frac{Y-F}{Y} & Y > F \end{cases}$$

Calculating p_u gives:

$$\begin{aligned} p_u &= 1 - \sum_{y=-\infty}^{\infty} p_y \cdot Pr(Y = y \mid S_{ij}) = \\ & 1 - \sum_{y=F+1}^{n-1} \frac{y-F}{y} \cdot \binom{n-2}{y-1} \left(\frac{F}{n-1}\right)^{y-1} \left(\frac{n-1-F}{n-1}\right)^{n-1-y} = \\ & \sum_{y=1}^{F} \binom{n-2}{y-1} \left(\frac{F}{n-1}\right)^{y-1} \left(\frac{n-1-F}{n-1}\right)^{n-1-y} + \\ & \sum_{y=F+1}^{n-1} \frac{F}{y} \cdot \binom{n-2}{y-1} \left(\frac{F}{n-1}\right)^{y-1} \left(\frac{n-1-F}{n-1}\right)^{n-1-y} \end{aligned} \tag{3.7}$$

If p_j is attacked with $x \geq F$ messages, we get:

$$p_Y = \frac{Y+x-1}{Y+x} \cdot \frac{Y+x-2}{Y+x-1} \cdots \frac{Y+x-F}{Y+x-F+1} = \frac{Y+x-F}{Y+x}$$

And thus:

$$\begin{aligned} p_a &= 1 - \sum_{y=-\infty}^{\infty} p_y \cdot Pr(Y = y \mid S_{ij}) = \\ & 1 - \sum_{y=1}^{n-1} \frac{y+x-F}{y+x} \cdot \binom{n-2}{y-1} \left(\frac{F}{n-1}\right)^{y-1} \left(\frac{n-1-F}{n-1}\right)^{n-1-y} = \\ & \sum_{y=1}^{n-1} \frac{F}{y+x} \cdot \binom{n-2}{y-1} \left(\frac{F}{n-1}\right)^{y-1} \left(\frac{n-1-F}{n-1}\right)^{n-1-y} < \\ & \sum_{y=1}^{n-1} \frac{F}{x} \cdot \binom{n-2}{y-1} \left(\frac{F}{n-1}\right)^{y-1} \left(\frac{n-1-F}{n-1}\right)^{n-1-y} = \frac{F}{x} \end{aligned}$$

Lemma 7 $\frac{\mathrm{d}p_a}{\mathrm{d}\alpha} < \frac{F}{\alpha x}$.

Proof: Calculating the derivatives, we get:

$$\begin{aligned} \frac{\mathrm{d}p_a}{\mathrm{d}x} &= \sum_{y=1}^{n-1} \frac{\mathrm{d}\frac{F}{y+x}}{\mathrm{d}x} \cdot \binom{n-2}{y-1} \left(\frac{F}{n-1}\right)^{y-1} \left(\frac{n-1-F}{n-1}\right)^{n-1-y} = \\ & \sum_{y=1}^{n-1} \frac{-F}{(y+x)^2} \cdot \binom{n-2}{y-1} \left(\frac{F}{n-1}\right)^{y-1} \left(\frac{n-1-F}{n-1}\right)^{n-1-y} \\ \frac{\mathrm{d}x}{\mathrm{d}\alpha} &= \frac{\mathrm{d}\frac{B}{\alpha n}}{\mathrm{d}\alpha} = \frac{-B}{\alpha^2 n} \\ \frac{\mathrm{d}p_a}{\mathrm{d}\alpha} &= \frac{\mathrm{d}p_a}{\mathrm{d}x} \cdot \frac{\mathrm{d}x}{\mathrm{d}\alpha} = \\ & \sum_{y=1}^{n-1} \frac{FB}{\alpha^2 n(y+x)^2} \cdot \binom{n-2}{y-1} \left(\frac{F}{n-1}\right)^{y-1} \left(\frac{n-1-F}{n-1}\right)^{n-1-y} = \\ & \sum_{y=1}^{n-1} \frac{Fx}{\alpha(y+x)^2} \cdot \binom{n-2}{y-1} \left(\frac{F}{n-1}\right)^{y-1} \left(\frac{n-1-F}{n-1}\right)^{n-1-y} < \\ & \sum_{y=1}^{n-1} \frac{Fx}{\alpha x^2} \cdot \binom{n-2}{y-1} \left(\frac{F}{n-1}\right)^{y-1} \left(\frac{n-1-F}{n-1}\right)^{n-1-y} = \frac{F}{\alpha x} \end{aligned}$$

□

We now give a bound on p_u.

Lemma 8 $p_u > 0.6$.

Proof: Define:

$$\mu \triangleq E[Y \mid S_{ij}] = \sum_{y=1}^{n-1} y \cdot \binom{n-2}{y-1} q^{y-1}(1-q)^{n-1-y} = \frac{n-2}{n-1} \cdot F + 1$$
$$E[Y^2 \mid S_{ij}] = \sum_{y=1}^{n-1} y^2 \cdot \binom{n-2}{y-1} q^{y-1}(1-q)^{n-1-y} = \frac{(n-2)(n-3)}{(n-1)^2} \cdot F^2 + 3 \cdot \frac{n-2}{n-1} \cdot F + 1$$
$$\sigma^2 \triangleq Var(Y \mid S_{ij}) = \frac{(n-2)(n-3)}{(n-1)^2} \cdot F^2 + 3 \cdot \frac{n-2}{n-1} \cdot F + 1 - \left(\frac{n-2}{n-1} \cdot F + 1\right)^2 = \frac{n-2}{n-1} \cdot F - \frac{n-2}{(n-1)^2} \cdot F^2$$

By [58], for $n \gg 1$ we get that Y given S_{ij} can be approximated using a normal distribution function, with $\mu = F + 1$ and $\sigma^2 = F$. The cumulative distribution function $D(x)$ is thus:

$$D(x) = \frac{1}{2} \cdot \left(1 + \mathrm{erf}\left(\frac{x-\mu}{\sqrt{2}\sigma}\right)\right) = \frac{1}{2} \cdot \left(1 + \mathrm{erf}\left(\frac{x-F-1}{\sqrt{2F}}\right)\right) \quad \text{where} \quad \mathrm{erf}(z) = 1 - \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-t^2} \mathrm{d}t$$

From [58] we get the following:

$$\frac{1}{x+\sqrt{x^2+2}} < e^{x^2} \int_x^\infty e^{-t^2} \mathrm{d}t < \frac{1}{x+\sqrt{x^2+\frac{4}{\pi}}}$$

Concluding that:

$$\mathrm{erf}(z) = 1 - \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-t^2} \mathrm{d}t > 1 - \frac{2}{\sqrt{\pi}} \cdot \frac{e^{-z^2}}{z + \sqrt{z^2 + \frac{4}{\pi}}}$$

The first sum in formula 3.7 is approximated by $D(F)$. Calculating $D(F)$ gives:

$$D(F) = \frac{1}{2} \cdot \left(1 + \mathrm{erf}\left(\frac{-1}{\sqrt{2F}}\right)\right) > \frac{1}{2} + \frac{1}{2} \cdot \left(1 - \frac{2}{\sqrt{\pi}} \cdot \frac{e^{-\frac{1}{2F}}}{\sqrt{\frac{1}{2F} + \frac{4}{\pi}} - \frac{1}{\sqrt{2F}}}\right) =$$
$$1 - \frac{1}{\sqrt{\pi}} \cdot \frac{e^{-\frac{1}{2F}}}{\frac{\sqrt{\pi+8F}}{\sqrt{2\pi F}} - \frac{1}{\sqrt{2F}}} = 1 - \sqrt{2} \cdot \frac{\sqrt{F} \cdot e^{-\frac{1}{2F}}}{\sqrt{\pi + 8F} - \sqrt{\pi}}$$

Define:

$$g(F) = \frac{\sqrt{F} \cdot e^{-\frac{1}{2F}}}{\sqrt{\pi + 8F} + \sqrt{\pi}}$$

We want to bound D(x) from above by finding for which values of F, $g(F) < 0$. The denominator of $g'(F)$ is always positive, so we ignore it when calculating the derivative:

$$\left(\frac{e^{-\frac{1}{2F}}}{2\sqrt{F}} + \frac{\sqrt{F}e^{-\frac{1}{2F}}}{2F^2}\right)\left(\sqrt{\pi + 8F} - \sqrt{\pi}\right) - \frac{8\sqrt{F}e^{-\frac{1}{2F}}}{2\sqrt{\pi+8F}} < 0$$
$$\frac{F^{\frac{3}{2}} + F^{\frac{1}{2}}}{2F^2}\left(\sqrt{\pi + 8F} - \sqrt{\pi}\right) - \frac{8\sqrt{F}e^{-\frac{1}{2F}}}{2\sqrt{\pi+8F}} < 0$$
$$\frac{\left(F^{\frac{3}{2}} + F^{\frac{1}{2}}\right)\left(\sqrt{\pi+8F} - \sqrt{\pi}\right)\sqrt{\pi+8F} - 8F^{\frac{5}{2}}}{2F^2\sqrt{\pi+8F}} < 0$$

Once again, the denominator is positive, and we get:

$$\left(F^{\frac{3}{2}}+F^{\frac{1}{2}}\right)\left(\sqrt{\pi+8F}-\sqrt{\pi}\right)\sqrt{\pi+8F}-8F^{\frac{5}{2}}<0$$
$$\pi+8F-\sqrt{\pi^2+8\pi F}-8F\cdot\left(1-\tfrac{1}{F+1}\right)<0$$
$$\tfrac{8F}{\sqrt{\pi}(F+1)}<\sqrt{\pi+8F}-\sqrt{\pi}$$

Taking derivatives we get:

$$\frac{8}{\sqrt{\pi}(F+1)^2} \overset{?}{<} \frac{8}{2\sqrt{\pi+8F}}$$
$$2\sqrt{\pi+8F} \overset{?}{<} \sqrt{\pi}(F+1)^2$$

Clearly, $(F+1)^2$ grows faster than $2\sqrt{\pi+8F}$. Numerically solving for $F=1$ shows that the inequality holds. Thus, it holds for every $F\in\mathbb{N}$. Consequently, we only need to find the first F for which:

$$\frac{8F}{\sqrt{\pi}(F+1)}<\sqrt{\pi+8F}-\sqrt{\pi}$$

A numerical solution for this inequality shows that it first holds for $F=3$. Thus, for $F\geq 3$ we get that $g'(F)<0$, and thus $D(F+1)>D(F)$. Assigning $F=3$ in our previous bound for D(F), we get that for all $F\geq 3,\ \ D(F)\geq D(3)>0.3968\approx 0.4$. Assuming $F\geq 3$, we get:

$$\sum_{y=1}^{F}\binom{n-2}{y-1}\left(\frac{F}{n-1}\right)^{y-1}\left(\frac{n-1-F}{n-1}\right)^{n-1-y}>0.4$$

Since $D(x)$ is maximal at $x=\mu=F+1$ and symmetric around it, we get the approximation:

$$\sum_{y=F+1}^{2F}\binom{n-2}{y-1}\left(\frac{F}{n-1}\right)^{y-1}\left(\frac{n-1-F}{n-1}\right)^{n-1-y}>\sum_{y=1}^{F}\binom{n-2}{y-1}\left(\frac{F}{n-1}\right)^{y-1}\left(\frac{n-1-F}{n-1}\right)^{n-1-y}$$

And finally, we conclude that:

$$\begin{aligned}
p_u &= \sum_{y=1}^{F}\binom{n-2}{y-1}\left(\frac{F}{n-1}\right)^{y-1}\left(\frac{n-1-F}{n-1}\right)^{n-1-y}+\\
&\sum_{y=F+1}^{n-1}\frac{F}{y}\cdot\binom{n-2}{y-1}\left(\frac{F}{n-1}\right)^{y-1}\left(\frac{n-1-F}{n-1}\right)^{n-1-y}>\\
&\frac{2}{5}+\sum_{y=F+1}^{2F}\frac{F}{2F}\cdot\binom{n-2}{y-1}\left(\frac{F}{n-1}\right)^{y-1}\left(\frac{n-1-F}{n-1}\right)^{n-1-y}>\\
&\frac{2}{5}+\frac{1}{2}\cdot\sum_{y=1}^{F}\binom{n-2}{y-1}\left(\frac{F}{n-1}\right)^{y-1}\left(\frac{n-1-F}{n-1}\right)^{n-1-y}>\\
&\frac{2}{5}+\frac{1}{2}\cdot\frac{2}{5}=\frac{3}{5}
\end{aligned}$$

□

3.10 Calculating $\widetilde{p}$

We now compute $\widetilde{p}$, the probability that M is propagated from the source in a round in Pull. Assume $n > F$, and define q as the probability that process p_2 appears in process p_1's $view_{pull}$, then $q = \frac{F}{n-1}$. Let Y be the number of valid pull-requests received in a single round, then:

$$\begin{array}{rrcl} & \Pr(Y < 0) & = & \Pr(Y \geq n) = 0 \\ 0 \leq y < n & \Pr(Y = y) & = & \binom{n-1}{y} q^y (1-q)^{n-1-y} \end{array}$$

Assume $x \geq F$, and define p_Y as the probability that a valid pull-request is read from the buffer, then:

$$p_Y = 1 - \left(1 - \frac{Y}{Y+x}\right)\left(1 - \frac{Y}{Y+x-1}\right)\cdots\left(1 - \frac{Y}{Y+x-F+1}\right) = 1 - \frac{x! \cdot (Y+x-F)!}{(x-F)! \cdot (Y+x)!}$$

The probability $\widetilde{p}$ that a valid pull-request is read from the buffer, independent of Y, is:

$$\widetilde{p} = \sum_{y=-\infty}^{\infty} p_y \cdot \Pr(Y = y) = \sum_{y=0}^{n-1} \left(1 - \frac{x! \cdot (y+x-F)!}{(x-F)! \cdot (y+x)!}\right) \binom{n-1}{y} \left(\frac{F}{n-1}\right)^y \left(\frac{n-1-F}{n-1}\right)^{n-1-y}$$

Chapter 4

Adaptive Drum

Denial of service (DoS) attacks are attacks that usually aim to exhaust resources by overloading an entity with large amounts of bogus messages. The use of armies of infiltrated machines ("zombies") leads to distributed DoS (DDoS), in which the attacker utilizes its set of compromised machines to launch a coordinated attack with massive strength. In this chapter, we consider application-level DoS attacks, in which the application is overwhelmed with messages to process even when the network is not congested. This situation is common is applications that require extensive processing for each incoming request, e.g., cryptographic authentication.

As DoS attacks can cause severe damage and are fairly easy to deploy, it is important to design communication protocols with DoS-mitigation mechanisms in place. However, designing a protocol that performs well under a certain DoS attack does not mean that it is still performs well as the attack changes. We believe that a protocol that adapts its parameters to the actual attack taking place can perform better than a static protocol that behaves the same under all DoS attacks. To illustrate this point, we focus on gossip-based multicast protocols as a case study.

Chapter 3 presented Drum, which is a gossip-based protocol that is designed to cope with DoS attacks by equally dividing the available resources between push and pull. However, this allocation is static, and does not consider the actual attack on the system. For example, even if only the push channels are attacked, every node in Drum still allocates half of its resource to push.

We assume the adversary does not know the identity of all nodes in the multicast group (which may be very large), and is thus restricted to attacking just a portion of the group. The attacker uses zombies to leverage its attack, and must communicate with them to update them on the attack strategy. Realizing that the system has adapted itself to the attack, devising a new attack plan and updating all zombies take time. We can therefore assume that by the time the attacker reacts to our adaptation, the system has completely reached its optimized point.

We present a novel approach that adapts a gossip-based protocol to the attack currently launched

on the system. The adaptation is modeled as an optimization problem, which is solved using linear programming. Every round, each node locally estimates the current state of the attack on the system and feeds it to the linear programming algorithm, which presents the node with the new resource distribution to use. We show, using simulations, that the local resource distribution each node independently calculates improves the global propagation time in the system, i.e., the number of rounds it takes a message generated at the source, to reach all nodes with high probability. Our propagation times are better than Drum's by up to 34%.

Adaptive Drum, like Drum, uses the important measures of combining push and pull and communicating using random ports. We concentrate on dynamic resource allocation using local decisions to achieve better message propagation times than Drum. Adaptation in gossip-based protocols has been explored before. For example, Rodrigues et. al [47] study adaptation in a gossip-protocol using flow-control to avoid congestion. Kyasanur et al. [29] study adaptive gossip in sensor networks, where the sensors wish to limit their transmission to conserve power consumption. However, we are the first the we know of that provide adaptation to DoS attacks.

The chapter proceeds as follows: Section 4.1 presents the assumptions we use for the adversary. Section 4.2 details the adaptation mechanism and discusses local estimations of the system's state. Section 4.3 gives simulation results that show the effectiveness of the adaptation and estimation.

4.1 Adversary Assumptions

We assume an external adversary that can cause messages to be dropped by overloading the multicast nodes with bogus requests. The adversary is not part of the multicast system, and does not participate in the gossip protocol. We assume that all nodes in the system are correct, follow the gossip protocol, and can differentiate between valid and bogus requests, perhaps at the cost of additional work, i.e., authentication. The adversary causes waste of resources by requiring the nodes to verify incoming requests, and supplying many bogus requests.

The attacker has bounded capacity for sending messages in a single round. When mounting an attack, the adversary chooses the nodes and ports to attack, out of the ones it knows, and the number of invalid messages it wishes to send to each attacked node each round. We denote by α the percentage of nodes being attacked, and for simplicity assume that all attacked nodes are attacked with C_{push} bogus push messages and C_{pull} bogus pull messages per round, i.e., every round C_{push} and C_{pull} bogus messages are sent to each attacked node's push and pull ports, respectively.

4.2 Adaptation

Each node locally adapts its behavior according to its view of the current state of the attack on the system. To perform a useful adaptation, there are two challenges to consider:

1. How to reliably estimate the current state of the attack.

2. Given the current state of the attack, what is the best strategy to employ.

To tackle these challenges, we start by assuming that all nodes know the exact state of the attack, and find a strategy that accommodates the attack and improves propagation time (Section 4.2.1). We then provide means to estimate the state of the attack (Section 4.2.2).

4.2.1 Finding the Target Strategy

The only communication elements a node controls are its push and pull channels, whether incoming or outgoing. The distribution of the node's limited resources among these channels constitutes the node's strategy. We want to find the best strategy each node should use to optimize the global propagation time when the system is under a DoS attack. Since gossip-based multicast protocols choose communication partners uniformly at random each round, and since all attacked nodes are attacked in the same manner, it is clear that all attacked nodes should exhibit the same behavior, and all unattacked nodes should use the same strategy. The strategies of the attacked and unattacked nodes will likely not be the same.

Recall that α is the percentage of attacked nodes, and every round the attacker sends each of these nodes C_{push} and C_{pull} bogus messages to their incoming push and pull channels, respectively. For simplicity of analysis, we transform C_{push} and C_{pull} to the concrete damage that they make, and define:

- p_s – the probability of a push message being dropped due to the attack on the push channels (depends on C_{push}).

- p_l – the probability of a pull message being dropped due to the attack on the push channels (depends on C_{pull}).

We use the following notations for node strategies:

- ASO is an attacked node's push fan-out, i.e., the number of nodes randomly-chosen each round as targets for outgoing push messages. A successful reception of an outgoing push message sent from node A to node B results in transferring data messages from node A to node B.

- ASI is an attacked node's push fan-in, i.e., the maximum number of randomly selected incoming push messages (valid or not) that will be processed in a single round.

- ALO and ALI are the same as ASO and ASI (respectively), but for pull. Unlike push, a successful reception of an outgoing pull message sent from node A to node B results in transferring data messages in the *opposite* direction – from node B to node A. That is, ALO is responsible for outgoing pull messages, *but incoming data messages*. ALI is responsible for incoming pull messages, *but outgoing data messages*.

- USO, USI, ULO and ULI are the same as ASO, ASI, ALO and ALI (respectively), but for an unattacked node.

By definition, all fan-ins and fan-outs are non-negative integers. For example: In Drum, all fan-ins and fan-outs are equal to F, where F is some positive integer, e.g., 4. In a push protocol, all push fan-ins and fan-outs are equal to $2F$, and all pull fan-ins and fan-outs are equal to 0. The value of F is bounded from above due to the limited resources the node can allocate for the communication.

We now move on to finding the nodes' best strategy when under attack. We do that by solving an optimization problem. We start by describing a set of constraints that each node must adhere to. All constraints are normalized by F, our basic unit of reference:

Constraint 1 $ALI + ASO = 2$
$ULI + USO = 2$

Reasoning. Receiving pull messages and sending push messages provide the same functionality – sending data messages from the node to the nodes it communicates with. The resources are thus bound by 2 units, as we are essentially bounding two communication channels (push and pull) together.

Constraint 2 $ASI + ALO = 2$
$USI + ULO = 2$

Reasoning. Both receiving push messages and sending pull message allow the node to receive data messages from nodes it communicates with. Once again, the total amount of resources allocated for this purpose is 2 units.
Additionally, we have some constraints on the system as a whole:

Constraint 3 $\alpha ASI + (1-\alpha)USI = \alpha ASO + (1-\alpha)USO$
$\alpha ALI + (1-\alpha)ULI = \alpha ALO + (1-\alpha)ULO$

Reasoning. The total amount of resources allocated for outgoing push messages should be equivalent to the total amount of resources allocated for incoming push messages, otherwise resources are wasted. This is true for pull as well.

Constraint 4 $\alpha ASO + (1-\alpha)USO = 2 \cdot \left(1 - \frac{p_s}{p_s+p_l}\right)$
$\alpha ALO + (1-\alpha)ULO = 2 \cdot \left(1 - \frac{p_l}{p_s+p_l}\right)$

Reasoning. It is important to have both the push and pull operations. The push operation allows an attacked source to propagate its message quickly via its outgoing push channel. It has been proven that it takes a time linear in C_{pull} to retrieve a message from an attacked source, when exclusively using the pull protocol (see Chapter 3). Pull allows an attacked node to receive data messages easily from an unattacked node, through the outgoing pull channel. Using push alone to deliver messages to attacked nodes takes a time linear in C_{push}.

Obviously, the more a channel is attacked, the less we want to use it – hence the ratio. Note that in case only one channel is attacked, it is closed and the attack has no influence. Obviously, this is the best strategy for such a case. Additionally, when both channels are attacked at the same strength, it is clear that the amount of resources allocated for push and pull should be equal (from symmetry).
Finally, we have the boundary conditions:

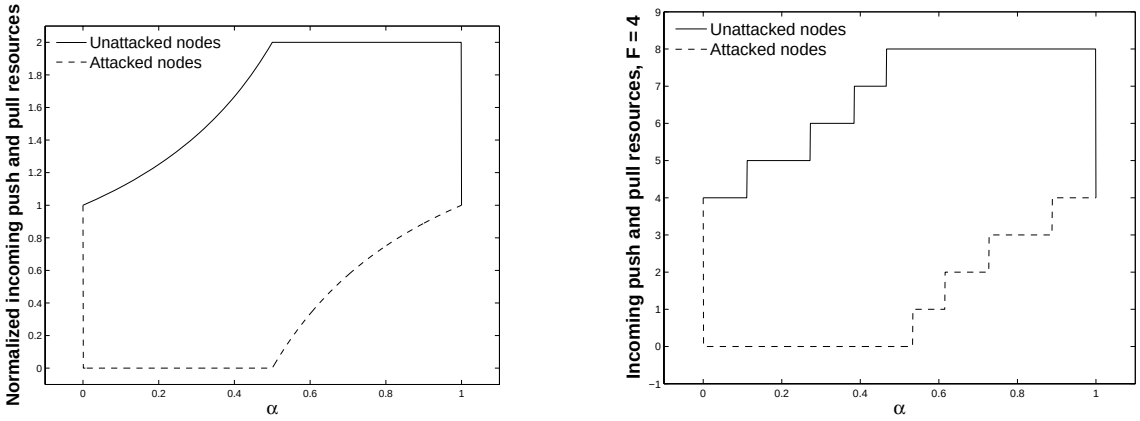
Constraint 5 $0 \leq ASO,\ ASI,\ ALO,\ ALI,\ USO,\ USI,\ ULO,\ ULI \leq 2$

To complete the optimization-problem statement, we still need to define the cost function to minimize. We want to minimize losses in the system, so that more messages can be processed by nodes, and thus data messages will be transferred faster. All messages lost due to the attack are dropped at the incoming channels of the attacked nodes. Assuming that attacked nodes are sent at least ASI and ALI valid messages for their incoming push and pull ports, respectively, the attack-induced losses in the system are defined by the following function:

$$f(\text{fan-outs and fan-ins}) = \alpha p_s ASI + \alpha p_l ALI$$

We have completed the definition of the optimization problem, and can now turn to solving it. Since we assume that we know α, p_s and p_l, we get a set of linear equations and inequalities in 8 variables (the fan-outs and fan-ins). The function to minimize, f, is also linear. Thus, we can solve this optimization problem using linear programming.

Figures 4.1, 4.2 and 4.3 show some solutions to the optimization problem for different scenarios, as calculated using MATLAB.



(a) Fan-ins computed using linear programming.

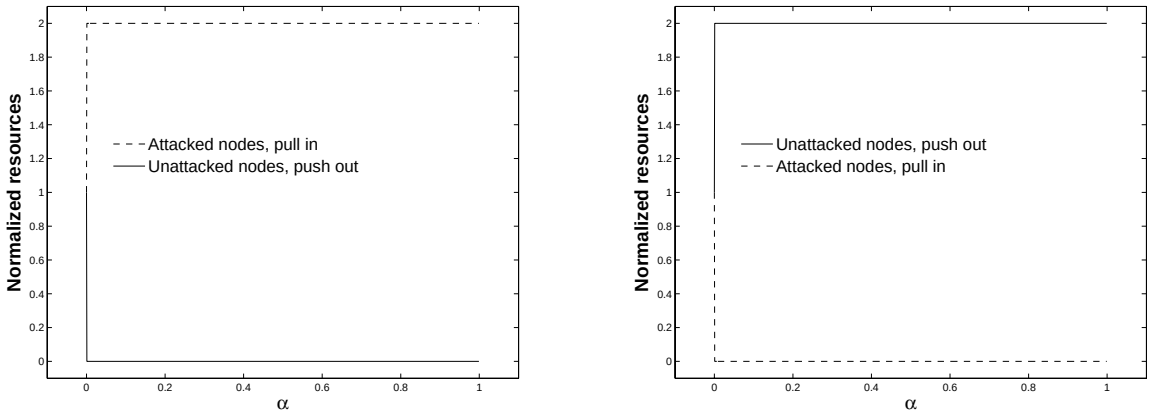
(b) Concrete fan-ins for $F = 4$.

Figure 4.1: Target strategies for $p_s = p_l > 0$, as a function of α.

Figure 4.1(a) shows the change in the resources allocated for the incoming push channels, ASI and USI, as a function of the percentage of the attacked nodes, α. The system is attacked on both push and pull channels, and the probability of a valid message being dropped due to the attack is greater than 0 and equal for both channels, i.e., $p_s = p_l > 0$. Due to this symmetry, exactly the same amount of resources is allocated for the incoming pull channels, i.e., ALI and ULI. The actual values of p_s and p_l do not matter, as long as they are equal and positive. We can see that as soon as the attack begins ($\alpha > 0$), the attacked nodes deallocate all resources used for the incoming push channels, which minimizes our cost function f. From Constraint 2 we can tell that these resources are diverted to the outgoing pull channels (not shown on figure). This is a good adaptation, since the attacked nodes experience problems receiving data messages via their incoming push channels due to the attack, and it is best if they concentrate more resources on receiving data messages using their outgoing pull channels, which do not directly suffer from the attack. Figure 4.1(b) shows the actual fan-ins the nodes should use (whole numbers), for $F = 4$.

From Figure 4.1(a) and Constraints 1 and 2, we get that as more nodes are attacked (up to 50% of the nodes), the total amount of resources allocated by attacked nodes for outgoing channels increases, since each attacked node directs all its resources to its outgoing channels. To accommodate this increase in the total amount of resources allocated for outgoing channels, the unattacked nodes increase the amount of resources allocated for their incoming channels. This conforms to Constraint 3. When more than 50% of the nodes are attacked, the unattacked nodes can no longer compensate for the increase in the incoming-channels' resources, as they have already exhausted all their available resources. Consequently, the attacked nodes change their behavior and direct some

resources from the outgoing channels to the incoming channels. Finally, a node's strategy in a system where all nodes are attacked, and both push and pull channels are attacked at the same strength, is equal to the node's strategy in a system in which no node is attacked at all. This is a consequence of all nodes experiencing the same environment, and the equal allocation of resources to the push and pull channels, as per Constraint 4, since both of them exhibit the same loss rate.

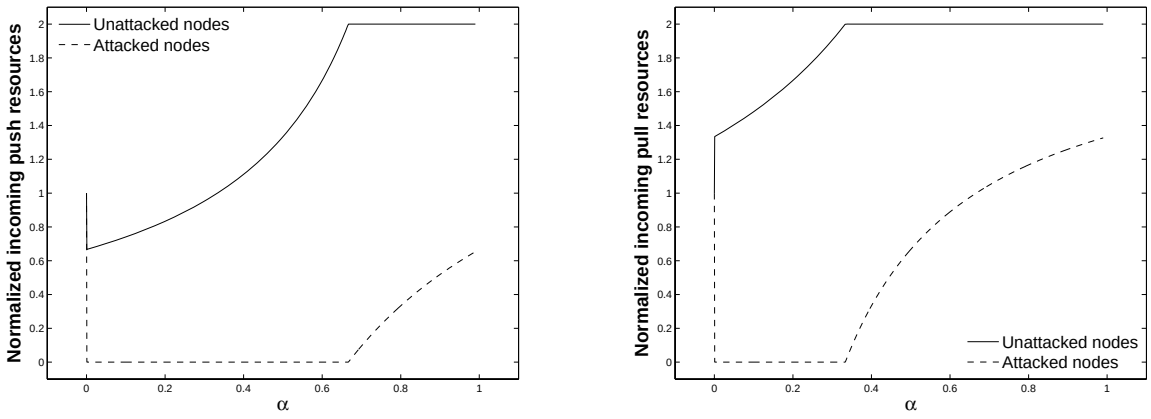


(a) $p_s = 1, p_l = 0$. (b) $p_s = 0, p_l = 1$.

Figure 4.2: Target strategies when only one of the channels is attacked, as a function of α.

Figures 4.2(a) and 4.2(b) show the nodes' behavior when only push or only pull channels are attacked. Figure 4.2 shows that when only push is attacked, the attacked nodes invest all their resources for outgoing data messages in the incoming pull channels, and thus, by Constraint 1, do not use outgoing push at all. We can see that the unattacked nodes do the same thing, as the resources they allocate for outgoing push messages immediately drop to 0 when the attack commences. It is easy to see that Constraint 3 means that no resources are allocated for incoming push messages as well, and all resources are diverted to outgoing pull messages (by Constraint 2). The resulting strategy is the exclusive use of pull in the system. Figures 4.2(b) shows the dual case, in which only pull is attacked. Similarly, the system adapts itself to using push alone. These results are intuitively appealing, as it is clear that if one channel is attacked but the other is not, we would not want to waste our resources on the attacked channel when we can get better results by using the unattacked channel.

Figures 4.3(a) and 4.3(b) show the nodes' behavior when the attack on push is stronger than the attack on pull, such that the loss probability for push, p_s, is 1, and the loss probability for pull, p_l, is 0.5. From Constraint 4 we get that the system will try to divide the total amount of resources allocated in the system for outgoing channels to $\frac{2}{3}$ for push, and $\frac{4}{3}$ for pull (out of a combined total of 2 normalized resources). Figure 4.3(a) shows that the attacked nodes immediately cease to use



(a) Push resources. (b) Pull resources.

Figure 4.3: Target strategies for $p_s = 1$, $p_l = 0.5$, as a function of α.

the incoming push channels as the attack begins (to minimize the cost function f), and shift the deallocated resources to the outgoing pull channels (Constraint 2). Similarly, Figure 4.3(b) shows that the attacked nodes also reduce to 0 the resources they allocate for the incoming pull channels, which means that their outgoing push channels are at full capacity (Constraint 1).

Back to Figure 4.3(a), the unattacked nodes continue to use push, to support the outgoing push channels of the attacked nodes, but reduce the resource allocation to $\frac{2}{3}$ of the basic unit. Similarly, in Figure 4.3(b) we can see the unattacked nodes using pull with an allocation of $\frac{4}{3}$ of the basic unit. As the percentage of attacked nodes increases, the unattacked nodes need to compensate for the increase in the total amount of resources allocated for the outgoing channels in the system, so they start increasing the resource allocation for their incoming channels, at the expense of their outgoing channels (cf. Figure 4.3). Once the unattacked nodes cannot allocate more resources for the incoming channels, the attacked nodes start allocating resources for the incoming channels, to support Constraint 3. This happens earlier for pull than for push.

4.2.2 Attack Estimation

Now that each node knows what strategy to employ based on the system's state, we need to devise a way for a node to estimate that state through local observations. According to our adaptation algorithm, a state is completely defined by α, p_s and p_l, so we need to find a way to estimate those variables.

A node's perception of the system comes from its interaction with other nodes. Each round the node performs push and pull communication with a different random subset of nodes, and can use this communication to evaluate the system's state.

An attacked node knows that it is being attacked, as it receives many bogus messages each round. Each time an attacked node communicates with other nodes, it informs them that it is being attacked. The estimation of α is based on the this information gained from communicating with attacked nodes. Additionally, p_s and p_l are estimated based on percentage of outgoing push and pull messages that were not replied to. This factor also contributes to the calculation of α, as we assume that messages that were not replied to were dropped due to an attack. This method of estimating p_s and p_l works well as long as the outgoing channels of the node performing the estimation and the incoming channels of the nodes being estimated have fan-outs and fan-ins (respectively) greater than 0. Otherwise, no meaningful data will be gathered.

To ensure that estimation is performed regardless of the fan-ins and fan-outs, we add a special *probe* message. Each node allocates static resources for incoming and outgoing probe messages. These messages are sent to the push and pull channels of other nodes, much like the push and pull messages. However, a probe message is like a ping message – a node that receives a probe message, simply replies with an empty message to indicate that it is able to receive messages. This mechanism is light-weight, and does not impose any limitations on the nodes. Specifically, it does not nearly consume as much resources as push and pull messages do, since no validation is performed, and no data messages are sent. The transmission rate of the probe messages to the incoming push and pull channels allows the nodes to evaluate p_s and p_l. The additional static resources allocated for these channels are used solely for answering probe messages, and not for answering push/pull messages.

We use the following notations when considering some node A's outgoing communication in a single round:

- SO, LO – the number of nodes A sent messages (including probes) to via the push or pull channels, respectively.
- SOA, LOA – the number of nodes A sent messages (including probes) to via the push or pull channels, respectively, and the nodes replied and indicated that they were attacked.
- SOD, LOD – the number of nodes A sent messages (including probes) to via the push or pull channels, respectively, and got no reply back. These nodes are also presumed to be under attack.

Each round r, every node performs its local estimations as follows:

$$\alpha(r) = \frac{SOA + SOD + LOA + LOD}{SO + LO}$$

$$p_s(r) = \frac{SOD}{SOA + SOD}$$

$$p_l(r) = \frac{LOD}{LOA + LOD}$$

Estimations are subject to fluctuations, since the choice of nodes to communicate with is random. In order to prevent the nodes from constantly changing their strategies even when the system's state remains intact, the nodes do not use single-round estimations in the calculation of their strategies, but rather use the average of the last k estimations, e.g., $AVERAGE(\alpha(r-k+1),\ \alpha(r-k+2),\ \ldots,\ \alpha(r-1),\ \alpha(r))$. The last k estimations are set to 0 when a node first joins the system. When an estimation cannot be performed, because of the denominator being 0 in some round, that round's estimation is chosen to be the average of the last k estimations. Choosing a small k means that nodes are able to respond more rapidly to a change in the system state (a change in the attack strength/distribution). Choosing a large k means that there are no fluctuations in the fan-ins and fan-outs as long as the system's state does not change.

4.3 Simulation Results

We test our adaptation mechanism through MATLAB simulations. Our system consists of a $1{,}000$ nodes, communicating using a gossip-based push/pull multicast protocol. The simulation progresses in synchronous rounds. In each round, all nodes send push/pull messages to randomly-selected nodes. Push/pull messages that do not get dropped due to limited incoming resources or due to an attack get answered, and then data messages are transferred. Finally, the nodes perform any calculations they may have for that round. All rounds begin and end at the same time in all nodes. A round is finished when all operations for that round end at all the nodes. In all experiments, $F = 4$.

Section 4.3.1 tests the effectiveness of the strategies computed in Section 4.2.1. α, p_s and p_l are assumed to be known, and the propagation time of our adaptive protocol is compared with 3 other protocols. Section 4.3.2 evaluates the estimation procedure described in Section 4.2.2. The nodes constantly estimate the state of the attack and change their strategies according to the solution to the minimization problem with the perceived α, p_s and p_l.

4.3.1 Strategy Evaluation

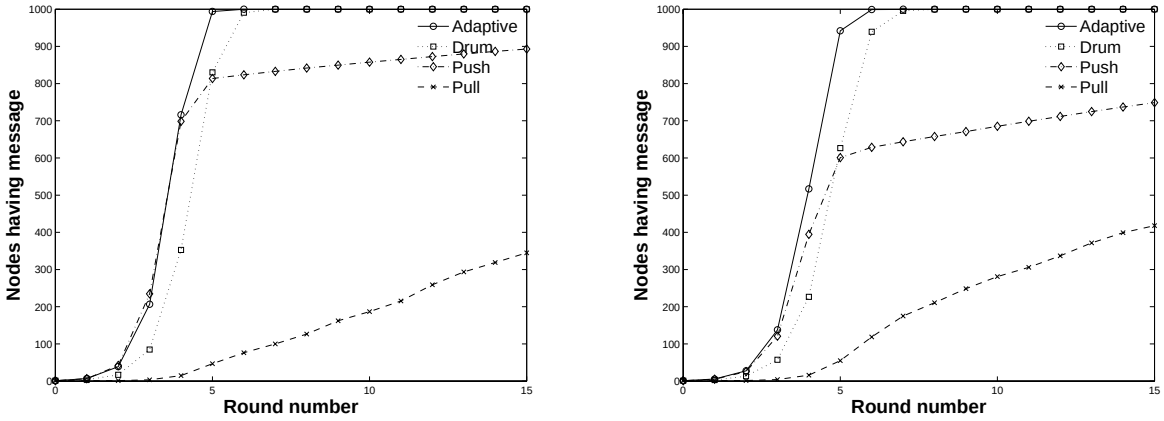
We start by evaluating our solution to the adaptation problem, as described in Section 4.2.1. We assume that α, p_s and p_l are known in advance to all nodes that use adaptation, and compare 4 gossip-based multicast protocols:

- Push – only uses the push channels.
- Pull – only uses the pull channels.
- Drum – divides its resources equally between push and pull.

- Adaptive Drum – divides its resources according to the adaptation strategy described in Section 4.2.1.

To determine the exact strategy Adaptive Drum uses, we need to determine the exact values of p_s and p_l before running the simulation. p_s depends on C_{push} and on ASI. Similarly, p_l depends on C_{pull} and ALI. We first assume that all fan-ins and fan-outs equal to F (as in Drum), and calculate p_s and p_l. Then, we solve the optimization problem and get the adapted ASI and ALI. This might change the values of p_s and p_l, so we recalculate them, and so on. When we are finished, we have the values of p_s and p_l after stabilization, and the proper strategies for all nodes. These are the strategies we use for Adaptive Drum.

We assume that new data messages are constantly generated in the system, and examine the propagation of one of those messages, i.e., the number of nodes that have the message as the rounds progress. The message originates at an attacked node in round 0. For simplicity, we assume that whenever nodes send data messages to one another, they send all the data messages they know of. This is consistent with the assumptions used in [8, 14] and in Chapter 3. Due to the random nature of gossip-based multicast protocols, each data point represents an average of 100 independent experiments.

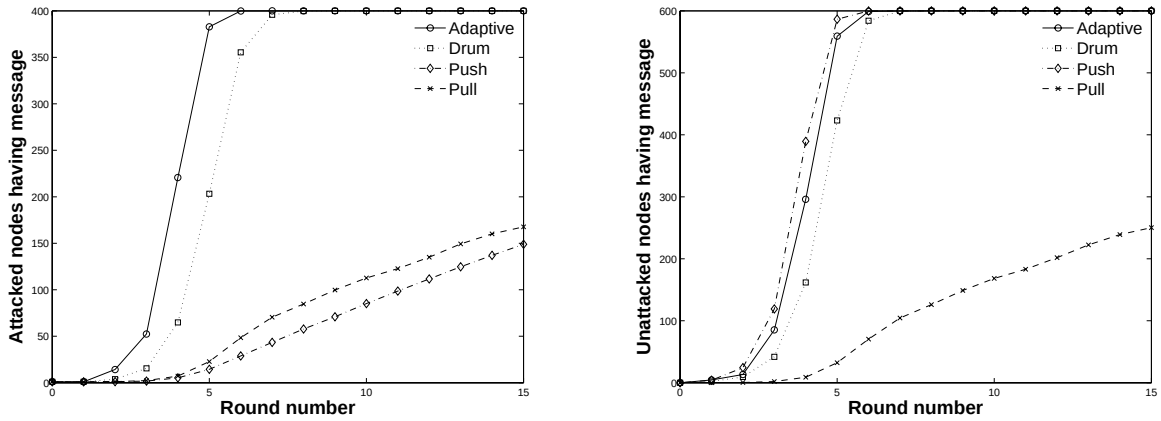


(a) $\alpha = 20\%$. (b) $\alpha = 40\%$.

Figure 4.4: Message propagation, $C_{push} = C_{pull} = 1{,}000$.

Figure 4.4 shows the message propagation for all 4 protocols, when both the attacked nodes' push and the pull incoming channels are attacked with $1{,}000$ bogus messages per channel per round ($p_s = p_l \approx 1$). The optimized strategy for this scenario was shown in Figure 4.1. Figure 4.4(a) shows the case where 20% of the nodes are under attack, and Figure 4.4(b) depicts the message propagation when $\alpha = 0.4$. In both cases, we can see that Adaptive Drum propagates the message to all nodes faster than the rest of the protocols. Push quickly propagates the message to the unattacked

nodes, but then takes time to deliver it to the attacked nodes. Pull experiences problems getting the message out of its source, since the source is attacked. Drum starts propagating the message slower than Push, since it also uses the pull channels, but then continue to propagate the message faster than Push, as Drum has little trouble to propagate the message to the attacked nodes. The decision of the attacked nodes to allocate all their resources to the outgoing channels means that Adaptive Drum's propagation times are similar to Push's propagation times, at the beginning of the dissemination. However, Adaptive Drum's robustness is soon realized, as it continues to propagate the message at the same good pace, while Push's propagation speed is significantly slowed when it is time to deliver the message to the attacked processes.



(a) Attacked nodes. (b) Unattacked nodes.

Figure 4.5: Breakdown of message propagation, $C_{push} = C_{pull} = 1{,}000$, $\alpha = 40\%$.

Figure 4.5 is a decomposition of Figure 4.4(b) to attacked nodes (Figure 4.5(a)) and unattacked nodes (Figure 4.5(b)). We can see that it is hard to deliver the message to the attacked nodes when using Push, and there is also much difficulty to extract the message from its (always) attacked source when using Pull. These observations were made in Chapter 3. Other than that, we can see that Adaptive Drum exhibits better propagation times than Drum right from the start, both for attacked and unattacked nodes, because it wastes less resources on messages that get dropped due to the attack.

Figure 4.6 compares the propagation times for Drum and Adaptive Drum. The figure shows the number of rounds it takes a message to reach all the nodes in the system for the worst experiment. That is, each data point is the minimal round number for which in all 100 experiments all nodes had the message. We can see that Adaptive Drum is constantly better than Drum, when there is an attack (recall that when there is no attack present, Adaptive Drum and Drum are exactly the same). Adaptive Drum improves the propagation time by 13% to 34% compared to Drum. Also,

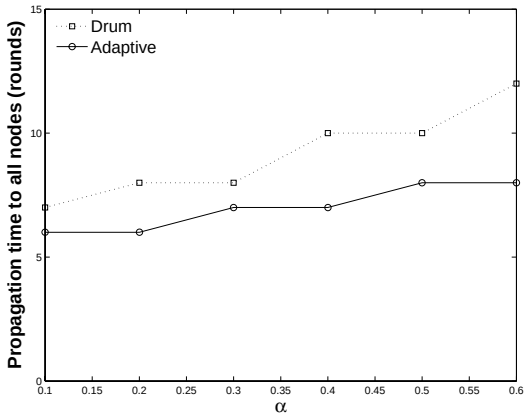
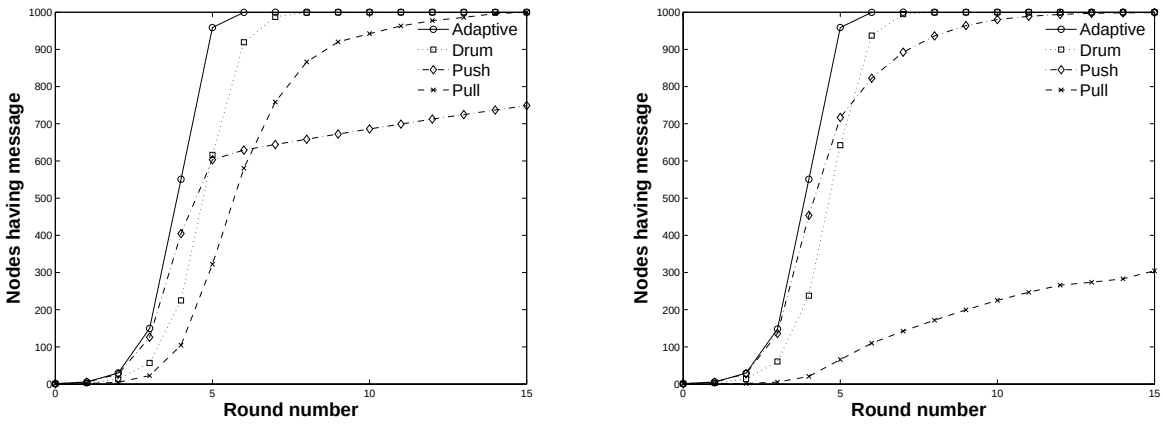


Figure 4.6: Propagation times of Drum vs. Adaptive Drum, $C_{push} = C_{pull} = 1{,}000$.

the improvement becomes more significant as the percentage of attacked nodes increases.



(a) $C_{push} = 1{,}000$, $C_{pull} = 100$. (b) $C_{push} = 100$, $C_{pull} = 1{,}000$.

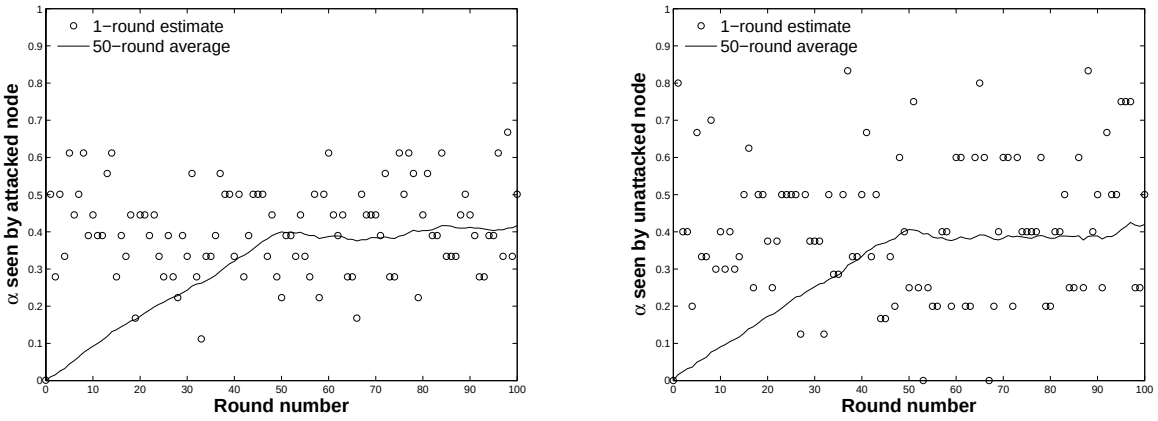
Figure 4.7: Message propagation under uneven attacks, $\alpha = 40\%$.

Finally, Figure 4.7 shows propagation times when the attack is uneven on the push channels and the pull channels. Although the attack is uneven, we still get that both p_s and p_l are very close to 1, due to the adaptation of the attacked nodes. Figure 4.7(a) depicts a scenario where push is attacked in a 10-times stronger attack than pull. We can see that indeed Pull performs better than Push, but still, Adaptive Drum provides the fastest propagation time. Figure 4.7(b) shows the opposite case, where the pull channels are attacked more severely than the push channels. In this case, we can see that the propagation time of Push improves. Nevertheless, Adaptive Drum still achieves the best propagation time. These results stem from the fact that when both channels are attacked, relying on

just one of them is not enough when it comes to delivering messages to attacked nodes (push) or receiving messages from attacked nodes (pull).

4.3.2 Estimation Evaluation

We proceed to evaluate the estimation mechanisms. In this set of experiments, nodes estimate α, p_s and p_l each round, and adjust their fan-ins and fan-outs for the next round according to the average of the last 50 estimations. The adversary attacks 40% of the nodes, with $C_{push} = C_{pull} = 1{,}000$. The attack begins in round 0. It is reasonable to assume that only a portion of the nodes is attacked, as the adversary may not even know about most of the nodes. All results presented are of a single experiment, for two nodes chosen at random. 100 experiments and several nodes were tested, and all provided similar results.

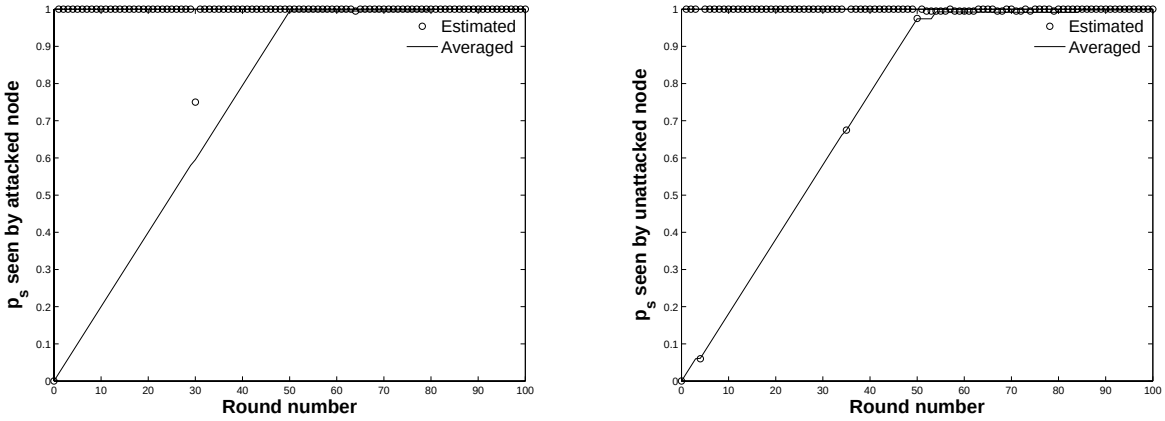


(a) Attacked node. (b) Unattacked node.

Figure 4.8: Estimation of α, $C_{push} = C_{pull} = 1{,}000$, $\alpha = 0.4$.

Figure 4.8 shows the estimation of α as performed by a randomly-selected attacked node (Figure 4.8(a)), and a randomly-selected unattacked node (Figure 4.8(b)). We can see that in both cases the nodes get a very close average estimation of α. Once 50 rounds pass and there are 50 estimations of the attack, the average estimation virtually stays the same.

Figure 4.9 shows the estimation of p_s as performed by a randomly-selected attacked node (Figure 4.9(a)), and a randomly-selected unattacked node (Figure 4.9(b)). Both nodes reach the same conclusion, that $p_s \approx 1$, which fits the calculation of p_s performed in Section 4.3.1 for Figure 4.4(b). In Figure 4.9(b) we have several estimation circles on the average estimation line, meaning that the unattacked node could not calculate a new estimation for that round (no data was available, due to the randomness in selecting the communication partners), and chose the average estimation as its estimation. The results for p_l are similar, and thus we do not show them here.



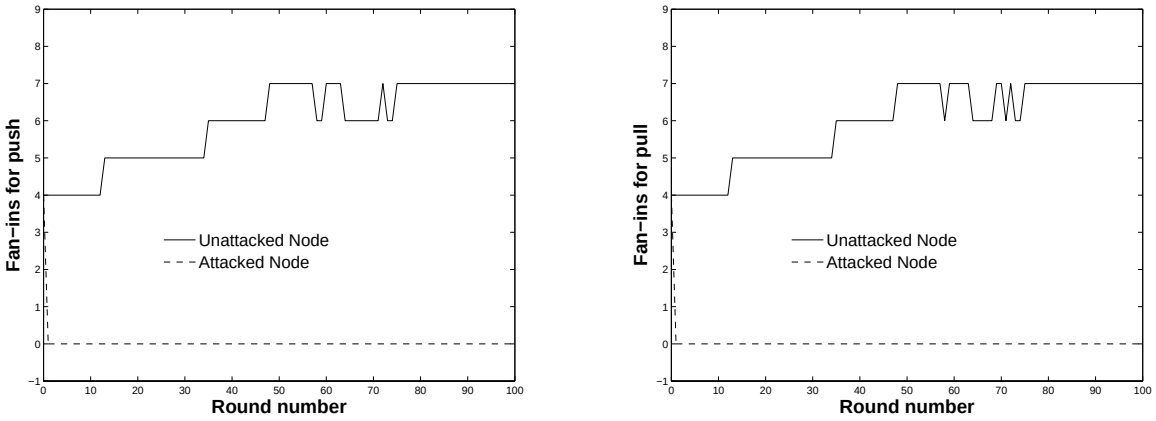
(a) Attacked node. (b) Unattacked node.

Figure 4.9: Estimation of p_s, $C_{push} = C_{pull} = 1{,}000$, $\alpha = 0.4$.

The application of the average estimations shown in Figure 4.8 and Figure 4.9 is presented in Figure 4.10. Figure 4.10(a) shows the push fan-ins resulting from solving the optimization problem using the average estimations. Figure 4.10(b) shows the pull fan-out for the same solution. The fan-ins presented are used by a randomly-selected attacked node, and a randomly-selected unattacked node. Since the average estimations were fairly accurate, the resulting fan-ins are the same ones used when the attack is known (cf. data point for $\alpha = 0.4$ in Figure 4.1(b)). Thus, we get that using local decisions and incomplete knowledge at each node, the whole system adapts itself to using the fan-ins (and thus also the fan-outs) that solve the optimization problem when the attack parameters are fully known.

Since the adversary takes time to realize that the system has adapted its behavior and inform all the zombies to change the attack strategy, our protocol should resist even attackers that change their attack strategy. We chose to measure the average for the last 50 rounds, and indeed, after 50 rounds the system stabilizes. Essentially, parts of the system may stabilize before others do, e.g., the attacked nodes reach their final fan-ins immediately, since as soon as they sense an attack they drop the allocated resources for their incoming channels to 0 (see in Figure 4.1). Thus, the propagation time can be improved even before 50 rounds pass. Averaging can also be made on less than 50 rounds, to reach the final strategy faster. Either way, rounds are short in nature (may be less than a second), and 50 rounds only take several seconds.

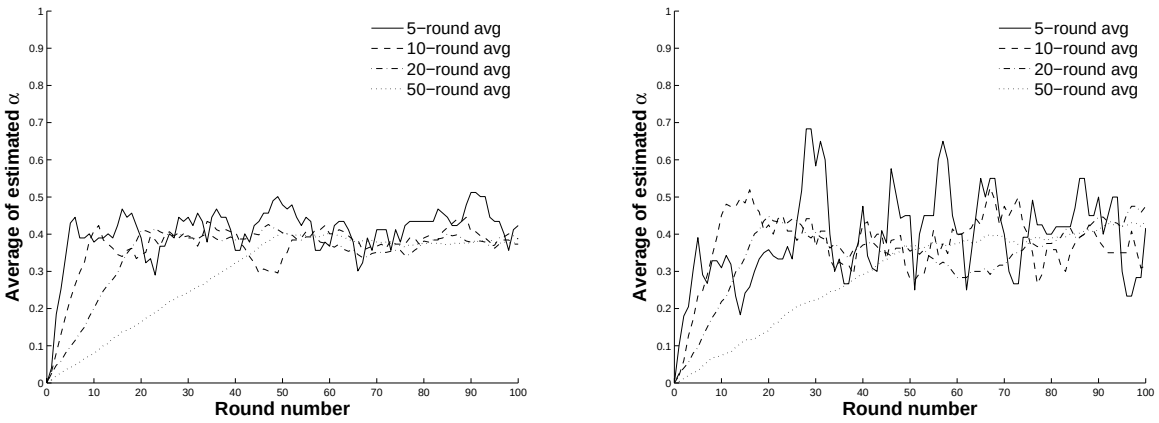
Figure 4.11 examines the use of various averages for the estimation of α. The figure shows the average estimated value of α for different numbers of data points per average. Figure 4.11(a) shows the averages as calculate for a randomly-selected attacked node, and Figure 4.11(b) shows the calculated averages for an unattacked node. These figures correspond to Figure 4.8. We can



(a) Push fan-ins. (b) Pull fan-ins.

Figure 4.10: Adaptation of fan-ins, $C_{push} = C_{pull} = 1{,}000$, $\alpha = 0.4$.

see that the smaller the length of the average, the more it fluctuates, although it reaches the area of $\alpha = 0.4$ more rapidly. The fluctuations are less evident for the attacked node, since the attacked node has its outgoing channels at full capacity, and thus gets more samples for the estimation. In contrast, the unattacked node is mainly focusing on the incoming channels, so the little resources it uses for the outgoing channels provide him with little information for the estimation.



(a) Attacked node.

(b) Unattacked node.

Figure 4.11: Estimation of α using various averages, $C_{push} = C_{pull} = 1{,}000$, $\alpha = 0.4$.

Chapter 5

ϕ-Hopper

Denial of service (DoS) attacks have proliferated in recent years, causing severe service disruptions [12]. The most devastating attacks stem from *distributed denial of service (DDoS)*, where an attacker utilizes multiple machines (often thousands) to generate excessive traffic [39]. Due to the acuteness of such attacks, various commercial solutions and off-the-shelf products addressing this problem have emerged. The main goal of all solutions is to provide lightweight packet-filtering mechanisms that are adequate for use in high-speed networks, where per-packet analysis must be efficient.

The most common solution uses an existing firewall/router (or protocol stack) to perform *rate-limiting* of traffic, and to *filter* messages according to header fields like address and port number. Such mechanisms are cheap and readily available, and are therefore very appealing. Nevertheless, rate-limiting indiscriminately discards messages, and it is easy to spoof (fake) headers that match the filtering criteria: an attacker can often generate spoofed packets containing correct source and destination IP addresses, and arbitrarily chosen values for almost all fields used for filtering.[1] Therefore, the only hope in using such efficient filtering mechanisms to overcome DoS attacks lies in choosing values that are *unknown to the adversary*. E.g., TCP's use of a random initial sequence number is a simple version of this approach, but is inadequate if the attacker has some (even limited) eavesdropping capability.

More effective DoS solutions are provided by expensive commercial devices that perform stateful filtering [42, 43, 45]. These solutions specialize in protecting a handful of commonly-used stateful protocols, e.g., TCP; they are less effective for stateless traffic such as UDP [45]. Such expensive solutions are not suitable for all organizations.

Finally, the most effective way to filter out offending traffic is using secure source authentication

[1] An exception is the TTL field of IP packets, which is automatically decremented by each router. This is used by some filtering mechanisms, e.g. BGP routers that receive only packets with maximal TTL value (255) to ensure the packets were sent by a neighboring router, and the Hop Counter Filtering proposal [22].

with message authentication codes (MAC), as in IPSec [3]. However, this requires computing a MAC for every packet, which can induce significant overhead, and thus, this approach may be even more vulnerable to DoS attacks. Specifically, it is inadequate for use in high-speed networks with high volumes of traffic.

Our goal is to address DoS attacks on end hosts, e.g., in corporate networks, assuming the network leading to the hosts is functional. (A complementary solution protecting the end network can be deployed at the ISP.) In this chapter, we focus on fortifying the basic building block of two-party communication. Specifically, we develop a DoS-resistant datagram protocol, similar to UDP or raw IP. Our protocol has promising properties, especially in overcoming realistic attack scenarios where attackers can discover some of the control information included in protocol packets, as also described in [1]. We assume that a realistic adversary can detect whether its attack is successful or not, and adjust its behavior accordingly. However, this adjustment takes some time, as it involves gathering information from the system, processing it to decide on the proper adjustment, and then notifying all the attacking nodes (massive attacks employ many nodes). We believe that our ideas, with some practical adjustments, have the potential to find their way into future DoS protection systems. E.g., these ideas can be integrated into IPSec [3]. Our formal analysis proves the effectiveness of our ideas, and thus shows that their realization into a working system is highly beneficial.

The key to exploiting lightweight mechanisms that can filter high-speed traffic is using a *dual-layer approach*: On the one hand, we exploit cheap, simple, and readily-available measures at the network layer. On the other hand, we leverage these network mechanisms to provide sophisticated defense at the application layer. The latter allows for more complex algorithms as it has to deal with significantly fewer packets than the network layer, and may have closer interaction with the application. The higher layer dynamically changes the filtering criteria used by the underlying layer, e.g., by closing certain ports and opening others for communication. It is important to note that the use of dynamically changing ports instead of a single well-known port does not increase the chance of a security breach, as a *single* application is listening on *all* open ports.

The main contribution of our work is in presenting a *formal framework* for understanding and analyzing the effects of proposed solutions to the DoS problem. The main challenges in attempting to formalize DoS-resistance for the first time are: coming up with appropriate models for the attacker and the environment, modeling the functionality that can be provided by underlying mechanisms such as firewalls, and defining meaningful metrics for evaluating suggested solutions. We capture the functionality of a simple network-level DoS-mitigation solution by introducing the abstraction of a *port-based rationing channel*. It is important to note that our use of ports just serves as an example. In fact, any field that appears on all packets can be used as the filtering criterion, and our analysis and suggested protocol apply to all such fields. For simplicity, we henceforth use the term 'port' to refer to any filtering criterion that can be dynamically changed by the application level.

Our primary metric of an end-to-end communication protocol's resistance to DoS attacks is *success rate*, which is the worst-case expected portion of valid application messages that successfully reach their destination, under a defined adversary class.

Having defined our model and metrics, we proceed to give a generic analysis of the communication success rate over a port-based rationing channel in different attack scenarios. We distinguish between *directed* attacks, where the adversary knows the port used, and *blind* attacks, in which the adversary does not know the port. Not surprisingly, we show that directed attacks are extremely harmful: with as little as 100 machines (or a sending capacity 100 times that of the protocol) the success rate is virtually zero. On the other hand, the worst-case success rate that an attacker can cause in blind attacks in realistic scenarios is well over 90% even with $10{,}000$ machines.

Our goal is therefore to "keep the attacker in the dark", so that it will have to resort to blind attacks. Our basic idea is to change the filtering criteria (i.e., ports) in a manner that cannot be predicted by the attacker. This *port-hopping* approach mimics the technique of frequency hopping spread spectrum in radio communication [49]. We assume that the communicating parties share a secret key unknown to the attacker; they apply a pseudo-random function [18] to this key in order to select the sequence of ports they will use. Note that such port-hopping has negligible effect on the communication overhead for realistic intervals between hops, and thus can be used even in high-speed networks. The remaining challenge is synchronizing the processes, so that the recipient opens the port currently used by the sender. We present a protocol for doing so in a realistic partially synchronous model, where processes are equipped with bounded-drift bounded-skew clocks, and message latency is bounded.

The chapter proceeds as follows: Section 5.1 details related work. Section 5.2 details our models for the communication channel and the adversary. Section 5.3 provides generic DoS analysis. Section 5.4 describes our port-hopping protocol and analyzes its effectiveness.

5.1 Related Work

Our work continues the line of research on prevention of Distributed Denial of Service attacks, which focuses on filtering mechanisms to block and discard the offending traffic. Our work is unique in providing rigorous model and analysis, which constitute the first step in formally modeling and evaluating the effectiveness of possible filtering and rate limiting mechanisms. Since our formal framework is not restricted to port-based filtering, but rather operates with any filtering based on per-packet fields, our model and analysis can be used in evaluating future protocols, and may assist in examining and comparing the solutions that exist now.

Most closely related is the work on SOS [27], followed by the work on Mayday [1]. Both propose realistic and efficient mechanisms that do not require global adoption, yet allow a server

to provide services immune to DDoS attacks. These solutions, like ours, utilize efficient packet-filtering mechanisms between the server and predefined, trusted 'access point' hosts. The basic ideas of filtering based on ports or other simple identifiers ('keys'), and even of changing them, already appear in [1, 27], but without analysis and details. Additionally, [1] provides a discussion of attack types and limitations, justifying much of our model, including the assumption that the exposure of the identifier (port) number may be possible but not immediate. Furthermore, [1] mentions blind and targeted attacks (where blind attacks are attacks in which the adversary does not know the valid identifier), and asserts that the damage to the system is much more severe when targeted attacks are launched. We prove that this is indeed the case, and give exact quantities for the maximum performance degradation in both attack scenarios. Both SOS and Mayday require the setup of an overlay network consisting of several nodes, and use several levels of indirection to obscure the identity of the nodes that may prove to be a promising attack target. These levels of indirection may increase latency by a factor of 5 or even 10 [27]. In contrast, our solution does not require additional hosts, preserves communication characteristics, and is simple to construct and maintain.

Additional work [52] employs an overlay network similar to SOS, which uses spread-spectrum-like path diversity to counter DoS attacks. The system also uses secret keys to authenticate valid messages. Like SOS, it requires additional nodes to construct the overlay network, and the additional overhead has an impact on message throughput and latency.

There are other several proposed methods to filter offending DoS traffic. Some proposals, e.g., by Krishnamurthy et al. [28, 23], filter according to the source IP address. This is convenient and efficient, allowing implementation in existing packet filtering routers. However, IP addresses are subject to spoofing; furthermore, using a white-list of source addresses of legitimate clients/peers is difficult, since many hosts may have dynamic IP addresses due to the use of NAT, DHCP and mobile-IP. Some proposals try to detect spoofed senders, using new routing mechanisms such as 'path markers' supported by some or all of the routers en route, as in Pi [60], SIFF [61], AITF [2], and Pushback [34], but global router modification is difficult to achieve. Few proposals try to detect spoofed senders using only existing mechanisms, such as the hop count (TTL), as in HCF [22]. However, empirical evaluation of these approaches show rather disappointing results [11].

A different approach is to perform application-specific filtering for pre-defined protocols [24, 41]. Such protection schemes are cumbersome, only work for a handful of well-known protocols, and are usually restricted to attackers that transmit invalid protocol packets.

IPSec [3] performs filtering at the IP layer, by authenticating messages using message authentication codes (MACs), based on shared secret keys. IPSec ensures that higher-level protocols only receive valid messages. However, the work required to authenticate each message is invested for each incoming packet that has a valid SPI. Once the SPI, which is sent in the clear, is known, an attacker can perform a DoS attack by overloading IPSec with many bogus packets to authenticate.

In contrast, our solution ensures that the authentication phase is reached only for packets that are valid w.h.p., by constantly changing the cleartext filtering identifier, e.g., the SPI.

In Chapter 3, we have presented Drum – a gossip-based multicast protocol resistant to DoS attacks. Drum does not use pseudo-random port-hopping, and it heavily relies on well-known ports that can be easily attacked. Therefore, Drum is far less resistant to DoS attacks than the protocol we present here. Finally, Drum focuses on multicast only, and as a gossip-based protocol, it relies on a high level of redundancy, whereas the protocol presented herein sends very little redundant information.

Independently of our work, Lee and Thing [30] examined the use of port-hopping to mitigate the effect of DoS attacks. However, they concentrated more on implementation and empirical results, providing only a very brief analysis of their method. Even so, their empirical results do not state the strategy the attacker employs for its attack, and it is not clear whether the adversary cannot launch a better attack against their protocol. Conversely, we provide a thorough formal analysis of the environment and our protocol. We formally model the communication channel and the adversary, and provide rigorous proofs for the correctness and effectiveness of our protocol under the best attack the adversary can possibly launch.

Wang, Liu and Chien [54] provide simulation results for various DDoS attacks on general proxy networks, and the applications protected by them. However, they do not provide any theoretical analysis, and only deal with general proxy networks.

5.2 Model and Definitions

5.2.1 Overview

We consider a realistic *semi-synchronous* model, where processes have continuously-increasing local clocks with bounded drift Φ from real time. Each party may schedule events to occur when its local clock reaches a specific value (time). There is a bound Δ on the transmission delay, i.e., every packet sent either arrives within Δ time units, or is considered lost. Notice that while we assume messages *always* arrive within Δ time, this is only a simplification, and our results are valid even if a few messages arrive later than that; therefore, Δ should really be thought of as the typical maximal round trip time, and not as an absolute bound on a message's lifetime (e.g., a second rather than 60 seconds).

Our goal is to send messages from a sender A to a recipient B, in spite of attempts to disrupt this communication by an adversary. The basic technique available to the adversary is to clog the recipient by sending many packets. The standard defense deployed by most corporations is to rate-limit and filter packets, typically by a firewall. We capture this type of defense mechanism using a *port-based rationing channel* machine, which models the FIFO communication channel between

A and B as well as the filtering mechanism. To send a message, A invokes a *ch_send(m)* event, a message is received by the channel in a *net_recv(m)* event, and B receives messages via *ch_recv(m)* events. We assume that the adversary cannot clog the communication to the channel, and that there is no message loss other than in the channel. The channel discards messages when it performs rate-limiting and filtering.

The channel machine is formally defined in Subsection 5.2.2. We now provide an intuitive description of its functionality. Since we assume that the attacker can spoof packets with valid addresses, we cannot use these addresses for filtering. Instead, the channel filters packets using port numbers, allowing deployment using existing, efficient filtering mechanisms. Specifically, let the set Ψ of port numbers be $\{1, \ldots, \psi\}$. Our solutions can be used with larger values of ψ, however this may require modified filtering mechanisms. The buffer space of the channel is a critical resource. The channel's interface includes the *alloc* action, which allows B to break the total buffer space of R messages into a separate allocation of R_i messages per port $i \in \Psi$, as long as $R \geq \sum_{i=1}^{\psi} R_i$. For simplicity, we assume that the buffers are read and cleared together in a single *deliver* event, which occurs exactly once on every integer time unit. If the number of packets sent to port i since the last *deliver* exceeds R_i, a uniformly distributed random subset of R_i of them is delivered.

We define several parameters that constrain the adversary's strength. The most important parameter is the *attack strength*, C, which is the maximal number of messages that the adversary may inject to the channel between two *deliver* events.

As shown in [1], attackers can utilize different techniques to try to learn the ports numbers expected by the filters (and used in packets sent by the sender). However, these techniques usually require considerable communication and time. To simplify, we allow the adversary to eavesdrop by exposing messages, but we assume that the adversary can expose packets no earlier than $\mathcal{E}$ time after they are sent, where $\mathcal{E}$ is the *exposure delay* parameter. The exposure delay reflects the time it takes an attacker to expose the relevant information, as well as to distribute it to the (many) attacking nodes, possibly using very limited bandwidth (e.g., if sending from a firewalled network). Our protocol works well with as little as $\mathcal{E} > 5\Delta$.

Since the adversary may control some behavior of the parties, we take a conservative approach and let the adversary schedule the *app_send(m)* events in which the application (at A) asks to send m to B. To prevent the adversary from abusing these abilities by simply invoking too many *app_send* events before a *deliver* event, we define the *throughput*, $T \geq 1$, as the maximal number of *app_send* events in a single time unit. We further assume that $R \geq \Delta \cdot T$, i.e., that the capacity of the channel is sufficient to handle the maximal rate of *app_send* events.

Since we focus on connectionless communication such as UDP, our main metric for resiliency to DoS attacks is its *success rate*, namely the probability that a message sent by A is received by B.

Definition 1 (Success rate μ) *Let E be any execution of a given two-party protocol operating over*

a given port-based rationing channel with parameters $\Psi, R, C, \Phi, \Delta, \mathcal{E}$ and T, with adversary ADV. Let $end(E)$ be the time of the last deliver *event in E. Let $sent(E)$ ($recv(E)$) be the number of messages sent (resp., received) by the application, in app_send (resp., app_recv) events during E, prior to $end(E) - \Delta$ (resp, $end(E)$). The* success rate *μ of E is defined as $\mu(E) = \frac{recv(E)}{sent(E)}$. The success rate of adversary ADV is the average success rate over all executions of ADV. The success rate of the protocol, denoted $\mu(\Psi, R, C, \Phi, \Delta, \mathcal{E}, T)$, is the worst success rate over all adversaries ADV.*

Finally, a protocol can increase its success rate by sending redundant information, e.g., multiple copies or error-correcting codes. We therefore also consider a system's *message* (*bit*) *complexity*, which is the number of messages (resp. redundant bits) sent on the channel per each application message.

5.2.2 Formal Model and Specifications

VARIABLES:	$rcvd$, initially 0	// Number of last received message (for FIFO)
	$m(i)_{i\in\mathbb{N}}$, initially $\emptyset$	// i^{th} message sent
	$port(i)_{i\in\mathbb{N}}$, initially $\emptyset$	// port of i^{th} message
	$t(i)_{i\in\mathbb{N}}$, initially $\emptyset$	// time when i^{th} message was sent
	$time$, initially 0	// Current time
	$\{\ In(port)\ \}_{port\in\{1,\ldots,\psi\}}$, initially $\emptyset$	// Buffer of messages to processor q, per port
	$\{\ R_{port}\ \}_{port\in\{1,\ldots,\psi\}}$	// Ration of each port, set by recipient
	$sent$, initially 0	// Count of messages sent
	inj, initially 0	// Count of messages injected since last deliver
HANDLING OF EVENTS:		
On $ch\_send(m, port)$:	$m(++sent) \leftarrow m,\ \ port(sent) \leftarrow port,\ \ t(sent) \leftarrow time$	
On net_recv:	add $m(++rcvd)$ to $In(port(rcvd))$	
On $alloc(port, r)$:	**if** $(R \geq r - R_{port} + \sum_{i=1}^{\psi} R_i)$ **then** $R_{port} \leftarrow r$	
On $deliver$:	$inj \leftarrow 0$	
	for $port \in \{1, \ldots, \psi\}$ **do**:	// Deliver up to R_{port} messages from $In(port)$
	let M be random R_{port} messages from $In(port)$	
	for $m \in M$ **do:** $ch\_recv(m, port)$	
	$In(port) \leftarrow \emptyset$	// Clear buffer
On $inj(m, port)$:	**if** $inj{+}{+} \leq C$ **then** add m to $In(port)$	
On $expose(i)$:	**if** $time \geq t(i) + \mathcal{E}$ **then** return $\langle m(i), port(i)\rangle$	
On $advance(\delta)$:	$time \leftarrow sent > rcvd\ \ ?\ \min\{time + \lvert\delta\rvert, t(rcvd + 1) + \Delta\} : time + \lvert\delta\rvert$	

Figure 5.1: Port-based rationing channel for given $\Psi, R, C, \Phi, \Delta, \mathcal{E}$.

We model the system as a collection of interacting state machines. Each state machine is defined by its state (variables), set of possible initial states, and deterministic state transitions associated with input and output events. To allow machines to make random choices, initial states include random tapes.

We model the adversary as one of the deterministic state machines of which the system is composed. The adversary controls, among other things, the scheduling of events. That is, it defines the next event that will occur in any system state, as well as the progress of time (via the *advance* event). Thus, an *execution* of the system is completely defined by its initial state and number of steps.[2] The possible choices of random tapes define a probability space on executions.

A *port-based rationing channel* models a FIFO-ordered rate-limited communication channel with port-based message filtering. Figure 5.1 provides specifications for a channel from A to B; we assume an equivalent channel is used from B to A. The *net_recv* event models the arrival of the next message from A (in FIFO order) to the channel's buffer, allowing the adversary control of network latency (up to Δ).

The recipient uses the *alloc* operation to designate ration values R_i for ports $i \in \Psi = \{1, \ldots, \psi\}$. If $R_i > 0$ we say that port i is *open*. We use $In(i)$ to denote the set of messages in the input buffer designated with port i. The channel delivers all messages from $In(i)$ if $|In(i)| \leq R_i$, and a random subset of R_i messages from $In(i)$ if $|In(i)| > R_i$.

The adversary can inject messages directly into the buffer using *inj* events, and can snoop on the contents of messages using *expose* events, under the restrictions above.

5.3 Analyzing the Success Rate in a Single Slot with a Single Port

This section provides generic analysis of the probability of successfully communicating over a port-based rationing channel under different attacks, when messages are sent to a single open port, p. This analysis is independent of the timing model and the particular protocol using the channel, and can therefore serve to analyze different protocols that use such channels, e.g., the one we present in the ensuing section. We focus on a single $deliver$ event, and analyze the *channel's delivery probability*, which is the probability for a valid message in the channel's buffer to be delivered, in that event. Since every *ch_send(m)* event eventually results in m being added to the channel's buffer, we can use the channel's delivery probability to analyze the success rates of higher level protocols.

Let R_p denote the ration allocated to port p in the last *alloc* event, and let $In(p)$ be the contents of the channel's buffer for port p (see Subsection 5.2.2 for more details). Consider a $deliver$ event of a channel from A to B, when A sends messages only to port p. We introduce some notations[3]:

$R_p = R$ is the value of the channel's R_p when $deliver$ occurs.

$a_p = a$ is the number of messages whose source is A in the channel's $In(p)$ when $deliver$ occurs. We assume $a \leq R$. If $a_p < R_p$ (i.e., $a < R$), we say that there is *over-provisioning* on port p.

[2] We encapsulate all non-determinism and randomness in the choice of random tapes.

[3] Note that *for simplicity of notation only*, we remove the $_p$ subscript from R_p and a_p. All results are valid with the subscripts in place as well.

c_p is the number of messages whose source is *not* A in $In(p)$ when $deliver$ occurs.

Assume that $1 \leq a \leq R$. If $c_p < R - a + 1$ then B receives A's messages, and the attack does not affect the communication from A to B on port p. Let us now examine what happens when $c_p \geq R - a + 1$.

Lemma 9 *If $c_p \geq R - a + 1$, then the channel's delivery probability is $\frac{R}{c_p+a}$.*

Proof: The channel delivers $m \in In(p)$ if it is part of the R messages read uniformly at random from the $c_p + a$ available messages. Thus, the delivery probability is $\frac{R}{c_p+a}$. □

If the attacker knows that B has opened port p, it can direct all of its power to that port, i.e., $c_p = C$, where we assume $C \geq R - a + 1$. We call this a *directed attack.*

Corollary 3 *In a directed attack at rate C on B's port p, the delivery probability for messages sent to the attacked port p is $\frac{R}{C+a}$, assuming $1 \leq a \leq R$ and $C \geq R - a + 1$.*

Lemma 10 *For fixed R and c_p such that $1 \leq a \leq R$ and $c_p \geq R - a + 1$, the probability of B receiving only invalid messages on port p decreases as a increases.*

Proof: The channel delivers only invalid messages, if no message of the a valid messages is read. The corresponding probability is: $\frac{c_p}{c_p+a} \cdot \frac{c_p-1}{c_p+a-1} \cdots \frac{c_p-R+1}{c_p+a-R+1}$, which clearly decreases as a increases. □

5.3.1 Blind Attack

We define a *blind attack* as a scenario where A sends messages to a single open port, p, and the adversary cannot distinguish this port from a random one. We now analyze the worst-case delivery probability under a blind attack.

In general, an adversary's strategy is composed both of timing decisions and injected messages. The timing decisions affect a, the number of messages from A that are in the channel at a given delivery slot. Given that a is already decided, we define the set of all strategies of an attacker with sending rate C as:

$$S(C) \triangleq \left\{ \{c_i\}_{i \in \Psi} \mid \forall i \in \Psi : c_i \in \mathbb{N} \cup \{0\} \wedge \sum_{i=1}^{\psi} c_i = C \right\}$$

Each strategy $s \in S$ is composed of the number of messages the attacker sends to each port. Note that since the adversary wishes to minimize the delivery probability, we restrict the discussion to the set of attacks that fully utilize the attacker's capacity for sending messages.

Consider some fixed a, C, and R. We define $\mu_B(a, C, R, s)$ as the channel's delivery probability under attack strategy $s \in S$. Since S is a finite set, μ_B has at least one minimum point, and we define the delivery probability to be that minimum:

$$\mu_B(a, C, R) \triangleq \min_{s \in S(C)} \mu_B(a, C, R, s)$$

We sometimes use μ_B instead of $\mu_B(a, C, R)$ when a, C, and R are clear from context. We want to find lower bounds on μ_B, depending on the attacker's strength. We say that port p_i is attacked in strategy s if $c_{p_i} > 0$. We partition $S(C)$ according to the number of ports being attacked, as follows:

$$S_k \triangleq \{s \in S(C) \mid \text{Exactly } k \text{ ports are being attacked in } s\}$$

Consider a fixed $s_k \in S_k$, and denote by $p_1, p_2, \ldots, p_k$ the ports that the attacker attacks under strategy s_k at rates of $c_{p_1}, c_{p_2}, \ldots, c_{p_k}$ messages, respectively, where $\sum_{i=1}^{k} c_{p_i} = C$, $c_{p_i} > 0$. Then we assume that $\forall i\ c_{p_i} \geq R - a + 1$ (otherwise, even if $p_i = p$, the probability of B receiving A's messages is exactly 1).

We now find a lower bound on μ_B as follows: We first derive a lower bound on $\{\mu_B(a, C, R, s_k) | s_k \in S_k\}$; this lower bound is given as a function of k in Corollary 4. Incidently, the worst degradation occurs when the attacker divides its power equally among the attacked ports, i.e., when it sends $\frac{C}{k}$ messages to each attacked port (this is proven in Lemma 11). Then, we show lower bounds on $\mu_B(a, C, R)$ by finding the k that yields the minimum value.

Proposition 1 *Consider some fixed k, a, C, R, and $s_k \in S_k$, and denote the ports attacked under s_k by $p_1, p_2, \ldots, p_k$ with attacking rates of $c_{p_1}, c_{p_2}, \ldots, c_{p_k}$, respectively. Then $\mu_B(a, C, R, s_k) = \frac{\psi - k}{\psi} + \frac{1}{\psi} \sum_{i=1}^{k} \frac{R}{c_{p_i} + a}$.*

Proof: The probability that B does *not* deliver A's message is: $\sum_{i=1}^{k} Pr[p_i = p] \cdot \left(1 - \frac{R}{c_{p_i}+a}\right) = \frac{1}{\psi} \sum_{i=1}^{k} \left(1 - \frac{R}{c_{p_i}+a}\right) = \frac{k}{\psi} - \frac{1}{\psi} \sum_{i=1}^{k} \frac{R}{c_{p_i}+a}$. Thus, the delivery probability is $\frac{\psi - k}{\psi} + \frac{1}{\psi} \sum_{i=1}^{k} \frac{R}{c_{p_i}+a}$. □

The proofs of the following lemmas appear in Section 5.5.

Lemma 11 *Consider some fixed k, a, C, R, and $s_k \in S_k$, and denote the ports attacked under s_k by $p_1, p_2, \ldots, p_k$ with attacking rates of $c_{p_1}, c_{p_2}, \ldots, c_{p_k}$, respectively. Then under a blind attack with strategy s_k, the worst (i.e., minimal) expected delivery probability of the system is achieved when $\forall i\ c_{p_i} = \frac{C}{k}$.*

From Proposition 1 and Lemma 11 we get:

Corollary 4 *Under a blind attack, if k, a, C, and R are fixed, then the expected delivery probability for $s_k \in S_k$ is bounded from below as follows:* $\mu_B(a,C,R,s_k) \geq \frac{\psi-k}{\psi} + \frac{1}{\psi} \cdot \sum_{i=1}^{k} \frac{R}{\frac{C}{k}+a} = \frac{\psi-k}{\psi} + \frac{1}{\psi} \cdot \frac{kR}{\frac{C}{k}+a} = \frac{\psi-k}{\psi} + \frac{k^2R}{\psi(C+ka)}$.

We now define $\mu_B(k) \triangleq \min_{s_k \in S_k} \mu_B(s_k)$. We get that for each k:

$$\mu_B(k) = \frac{\psi-k}{\psi} + \frac{R}{\psi} \cdot \frac{k^2}{C+ka}$$

To find a lower bound, we continue this analysis as if k is continuous. The derivative of $\mu_B(k)$ is then:

$$\mu'_B(k) = \frac{-1}{\psi} + \frac{R}{\psi} \cdot \frac{2k(C+ka)-k^2a}{(C+ka)^2} = \frac{R}{\psi} \cdot \frac{2kC+k^2a}{(C+ka)^2} - \frac{1}{\psi} = \frac{R-1}{\psi} - \frac{R}{\psi} \cdot \frac{C^2+(2kC+k^2a)(a-1)}{(C+ka)^2}$$

We now state two lemmas that show that $\mu_B(a,C,R)$ is bounded from below by the function $f(a,C,R)$ presented in Equation 5.1 below.

Lemma 12 *Let $R = a$, then an adversary with $C \geq \psi$ cannot decrease the expected delivery probability lower than $\frac{\psi a}{C+\psi a}$, and an adversary with $C \leq \psi$ cannot decrease the expected delivery probability lower than $1 - \frac{C}{\psi(1+a)}$.*

Lemma 13 *Let $a < R$. Then an adversary with $C \geq \frac{\psi a}{\sqrt{\frac{R}{R-a}}-1}$ cannot decrease the expected delivery probability lower than $\frac{\psi R}{C+\psi a}$, and an adversary with $C \leq \frac{\psi a}{\sqrt{\frac{R}{R-a}}-1}$ cannot decrease the expected delivery probability lower than $\frac{\psi a - C\left(\sqrt{\frac{R}{R-a}}-1\right)}{\psi a} + \frac{R}{\psi} \cdot \frac{C\left(\sqrt{\frac{R}{R-a}}-1\right)^2}{a^2\sqrt{\frac{R}{R-a}}}$.*

We conclude the following corollary:

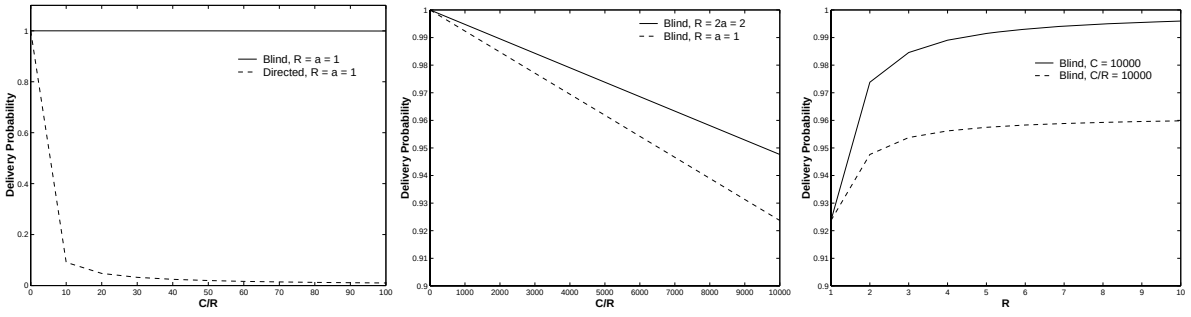
Corollary 5 *$\mu_B(a,C,R)$ is bounded from below by the following function $f(a,C,R)$:*

$$f(a,C,R) = \begin{cases} \frac{\psi a}{C+\psi a} & \text{if } R = a \text{ and } C \geq \psi \\ 1 - \frac{C}{\psi(1+a)} & \text{if } R = a \text{ and } C < \psi \\ \frac{\psi R}{C+\psi a} & \text{if } R > a \text{ and } C \geq \frac{\psi a}{\sqrt{\frac{R}{R-a}}-1} \\ \frac{\psi a - C\left(\sqrt{\frac{R}{R-a}}-1\right)}{\psi a} + \frac{R}{\psi} \cdot \frac{C\left(\sqrt{\frac{R}{R-a}}-1\right)^2}{a^2\sqrt{\frac{R}{R-a}}} & \text{if } R > a \text{ and } C < \frac{\psi a}{\sqrt{\frac{R}{R-a}}-1} \\ 0 & \text{otherwise} \end{cases} \quad (5.1)$$

Corollary 5 provides us with some insights of the adversary's best strategy and of the expected degradation in delivery probability. If no over-provisioning is used (i.e., $R = a$), then the adversary's best strategy is to attack as many ports as possible. This is due to the fact that even a single bogus message to the correct port degrades the expected delivery probability. When the adversary has enough power to target all of the available ports with at least one message, it can attack with more messages per attacked port, and the delivery probability asymptotically degrades much like the function $\frac{1}{C}$. When not all ports are attacked, the adversary would like to use its remaining resources to attack more ports rather than target a strict subset of the ports with more than one bogus message per port. The degradation of the expected delivery probability is then linear as the attacker's strength increases.

When over-provisioning is used ($R > a$), it affects the attack and its result in two ways. First, the attacker's best strategy may not be to attack as many ports as it can, since a single bogus message per port does not do any harm now. Second, for an adversary with a given strength, the degradation in delivery probability is lower when over-provisioning is used than when it is not employed. We can see in Equation 5.1 that if the attacker has enough power to attack all the ports, the over-provisioning ratio $\frac{R}{a}$ is also the ratio by which the delivery probability is increased, compared to the case where $R = a$.

5.3.2 Actual Values



(a) Blind vs. Directed, $R = a = 1$. (b) Blind, $a = 1$. (c) Blind, $a = 1$.

Figure 5.2: Delivery probability per slot in various attack scenarios on a single port, $\psi = 65536$.

Figure 5.2 shows the expected worst-case delivery probabilities for various attack scenarios on a single port. For directed attacks, we show the actual delivery probability, and for blind attacks, the lower bound $f(a, C, R)$ is shown. We chose $\psi = 65536$, the number of ports in common Internet protocols, e.g., UDP. Figure 5.2(a) illustrates the major difference between a directed attack and a blind one: even for a relatively weak attacker ($C \leq 100$), the delivery probability under a directed

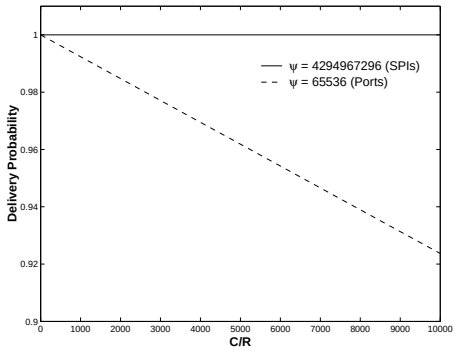


Figure 5.3: Blind mode delivery probability per slot for different values of ψ, $R = a = 1$.

attack approaches 0, whereas under a blind attack, it virtually remains 1.

Figure 5.2(b) examines blind attacks by much stronger adversaries (with C up to $10{,}000$ for $R = 1$, and up to $20{,}000$ for $R = 2$). We see that the delivery probability gradually degrades down to a low of 92.5% when $R = 1$. If we use an over-provisioned channel, i.e., have $a = 1$ (one message from A) when $R = 2$, the delivery probability improves to almost 95% for $C = 20{,}000$. (The ratio $\frac{C}{R}$ is the same for both curves). Figure 5.2(c) shows the effect of larger over-provisioning. We see that the cost-effectiveness of over-provisioning diminishes as $\frac{R}{a}$ increases.

The idea of hopping can essentially be applied to any changeable header field. For instance, other than the port numbers used in the TCP and UDP protocol, one may decide to use the SPI field of IPSec, which consists of 32 bits, or the Key field of GRE, as suggested in WebSOS [40]. Figure 5.3 shows the effect of hopping using IPSec's SPI field instead of using TCP/UDP ports. We can see that doubling the number of bits used for the filtering index has a substantial effect on the delivery probability. Using IPSec also has the added bonus of protecting all higher-level protocols, e.g., ICMP, TCP, UDP, etc.

5.4 DoS-Resistant Communication

We now describe a protocol that allows for DoS-resistant communication in a partially-synchronous environment. The protocol's main component is an ack-based protocol. B sends acknowledgments (acks) for messages it receives from A, and these acks allow the parties to hop through ports together. However, although the ack-based protocol works well as long as the adversary fails to attack the correct port, once the adversary identifies the port used, it can perform a directed attack that renders the protocol useless. By attacking the found data port, or simultaneously attacking the found data and ack ports, the adversary can effectively drop the success rate to 0, and no port hopping will occur. To solve this matter, there is a time-based proactive reinitialization of the ports used for the

ack-based protocol, independent of any messages passed in the system.

5.4.1 Ack-Based Port Hopping

We present an *ack-based port-hopping* protocol, which uses two port-based rationing channels, from B to A (with ration R_{BA}) and vice versa (with ration R_{AB}). For simplicity we assume $R_{AB} = 2R_{BA} = 2R$. B always keeps two open ports for data reception from A, and A keeps one port open for acks from B. The protocol hops ports upon a successful round-trip on the most recent port used, using a *pseudo-random function*, PRF^*[4]. In order to avoid hopping upon adversary messages, all protocol messages carry authentication information, using a second pseudo-random function, PRF, on $\{0,1\}^\kappa$. (We assume that PRF and PRF^* use different parts of A and B's shared secret key.)

The protocol's pseudocode appears in Figure 5.4. Both A and B hold a port counter P, initialized to some $seed$ (e.g., 1). Each party uses its counter P in order to determine which ports should be open, and which ports to send messages to. B opens port p_{old} using the $(P-1)^{th}$ element in the pseudo-random sequence, and p_{new}, using P. A sends data messages to the P^{th} port in the sequence, and opens the P^{th} port in a second pseudo-random sequence designated for acks. When B receives a valid data message from A on port p_{old}, it sends an ack to the old ack port. When it receives a valid message on port p_{new}, it sends an ack to the P^{th} ack port, and then increases P. When A receives a valid ack on port p_{ack}, it increases P. We note that several data messages may be in transit before a port hop takes place, since it takes at least one round-trip time for a port hop to take effect, and in a high-speed network, multiple messages are sent within this time span.

The proof of the next theorem is given in Section 5.6:

Theorem 1 *When using the ack-based protocol, the probability that a data message that A sends to port p arrives when p is open is 1 up to a polynomially-negligible factor*[5].

In order to compute the throughput that the protocol can support in the absence of a DoS attack (i.e., when $C = 0$), we need to take latency variations into consideration. Since messages sent up to Δ time apart can arrive in the same delivery slot, a throughput $T \leq R/\Delta$ ensures $a \leq R$. Since the protocol uses 2 incoming ports with the same rations, we require $T \leq \frac{R}{2\Delta}$, i.e., $a \leq \frac{R}{2}$.

We now analyze the protocol's success rate under DoS attacks. We say that the adversary is in *blind mode* if it cannot distinguish the ports used by the protocol from random ports. We first give a lower bound on the success rate in blind mode, and then give a lower bound on the probability

[4] Intuitively, we say that $f_{key}(data)$ is *pseudo-random function (PRF^*)* if for inputs of sufficient length, it cannot be distinguished efficiently from a truly random function r over the same domain and range, by a PPT adversary which can receive $g(x)$ for any values of x, where $g = r$ with probability half and $g = f$ with probability half. For definition and construction, see [18].

[5] Namely, for every polynomial $g > 0$, there is some κ_g s.t. when $\kappa \geq \kappa_g$, then the probability $\geq 1 - g(\kappa)$.

PROTOCOL FOR SENDER A:

On $ack\_init(seed)$**:**
$P = seed$
$p_{ack} = PRF^*_{S_{AB}}(P|\text{“}ack\text{”})$
$alloc(p_{ack}, R_{BA})$

On $app\_send(data)$**:**
$m = data|PRF_{S_{AB}}(P|\text{“}data\text{”})$
$ch\_send(m, PRF^*_{S_{AB}}(P|\text{“}data\text{”}))$

On $ch\_recv(ack, p_{ack})$**:**
if $ack.auth = PRF_{S_{AB}}(P|\text{“}ack\text{”})$ **then**
$alloc(p_{ack}, 0)$
$p_{ack} = PRF^*_{S_{AB}}(P+1|\text{“}ack\text{”})$
$alloc(p_{ack}, R_{BA})$
$P = P + 1$

PROTOCOL FOR RECIPIENT B:

On $ack\_init(seed)$**:**
$P = seed$
$p_{old} = PRF^*_{S_{AB}}(P-1|\text{“}data\text{”})$
$p_{new} = PRF^*_{S_{AB}}(P|\text{“}data\text{”})$
$alloc(p_{old}, R_{AB}/2)$
$alloc(p_{new}, R_{AB}/2)$

On $ch\_recv(m, p_{old})$**:**
if $m.auth = PRF_{S_{AB}}(P-1|\text{“}data\text{”})$ **then**
$app\_recv(m.data)$
$ack = PRF_{S_{AB}}(P-1|\text{“}ack\text{”})$
$ch\_send(ack, PRF^*_{S_{AB}}(P-1|\text{“}ack\text{”}))$

On $ch\_recv(m, p_{new})$**:**
if $m.auth = PRF_{S_{AB}}(P|\text{“}data\text{”})$ **then**
$app\_recv(m.data)$
$alloc(p_{old}, 0)$
$p_{old} = p_{new}$
$p_{new} = PRF^*_{S_{AB}}(P+1|\text{“}data\text{”})$
$alloc(p_{new}, R_{AB}/2)$
$ack = PRF_{S_{AB}}(P|\text{“}ack\text{”})$
$ch\_send(ack, PRF^*_{S_{AB}}(P|\text{“}ack\text{”}))$
$P = P + 1$

Figure 5.4: Two-party ack-based port-hopping.

to be in blind mode at a given time t. Finally, μ is bounded by the probability to be in blind mode throughout the execution of the protocol, times the success rate in blind mode.

Suppose B opens port p with reception rate R_p, and that $a \leq R_p$ messages from A are waiting in its channel, along with c_p messages from the adversary ($c_p \geq 0$). By Lemma 9, the success rate monotonically non-increases with a. Since the adversary can control a by varying the network delays, it can set a as high as possible for a delivery slot. Therefore, the worst case occurs when $a = T\Delta$. Using Equation 5.1, we get that the success rate in blind mode is bounded from below by $f(T\Delta, C, R)$.

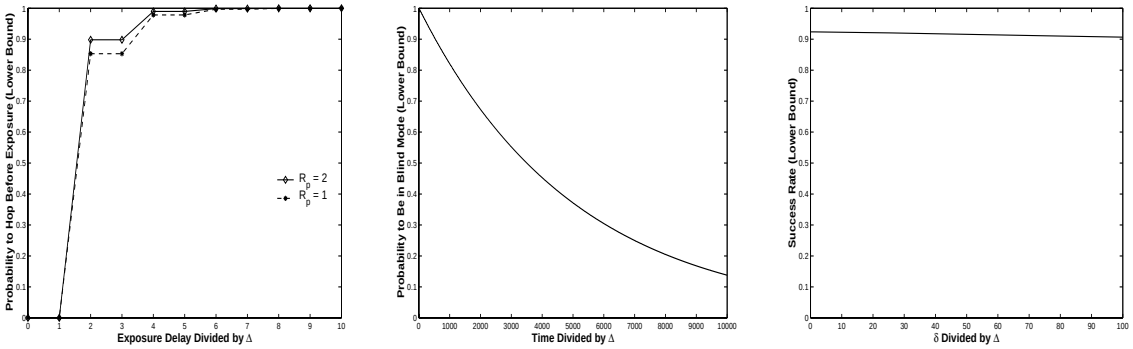
Note that the protocol begins in blind mode. We now analyze the probability that the protocol keeps the adversary in blind mode. The only way the adversary can learn of a port used by the protocol is using an *expose* event $\mathcal{E}$ time after a message is sent to that port. This information is only useful for an attack if the port is still in use. Let us trace the periodic sequence of events that causes the data port to change (once it changes, acks for the old port are useless). Assume A continuously sends messages $m_1, m_2, \ldots$ to B starting at time 0, and consider an execution without an attack: (1) By time Δ, B receives a valid message from the channel and sends an ack to A; (2) By time 2Δ, A receives the ack and changes the sending port; (3) B gets the last message destined for the old port at most at time 3Δ.

If $\mathcal{E} \geq 3\Delta$, the adversary remains in blind mode. Now let us examine what happens under attack. In order to prevent the port from changing, the adversary must either prevent B from getting valid data messages or prevent A from receiving acks. By Lemma 10, the probability that all valid messages are dropped decreases when a increases. Thus, (as opposed to the previous analysis), in order to increase the probability that all valid messages are dropped, the adversary would like to decrease a to its minimum. Obviously, the attacker would like to get out of blind mode, and for that, it needs A to send at least one message to B to expose the port being used, and so $a = 1$. We get that the lower bound on the probability of a single message to be received on a single port, as given in Section 5.3.1, is $\mu_B = f(1, C, \frac{R}{2})$.

Lemma 14 *If $\mathcal{E} = 2k\Delta$ for $k > 0$, and A sends messages to B at least every 2Δ time units, then the probability that the port changes while the attacker is still blind is at least $1 - (1 - \mu_B^2)^k$.*

Proof: The probability that the port does not change in a single round-trip is at most $1 - \mu_B^2$. Since A sends messages to B every 2Δ time units, at the conclusion of each maximal time round-trip, there is at least one new message on its own round-trip. In order for the port not to change while the adversary is still blind, every round-trip needs to fail. Since the attacker can react only after $2k\Delta$ time, there is time for k round-trips in which the attacker is blind, even if none of them succeed. The probability that all of them fail is less than $(1 - \mu_B^2)^k$. If one succeeds, the port changes. And so, the probability that the port changes is at least $1 - (1 - \mu_B^2)^k$. $\square$

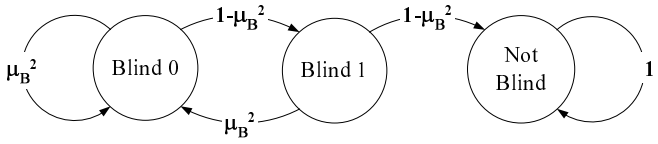
The lower bound above is illustrated in Figure 5.5(a).



(a) Bound on prob. to hop before exposure, $\frac{C}{R_p} = 10{,}000$.

(b) Bound (g) on prob. to stay in blind mode, $R_p = 1, C = 10{,}000, \mathcal{E} = 4\Delta$.

(c) Bound on protocol's success rate, $\Delta = T = R_p = 1, C = 10{,}000, \mathcal{E} = 5\Delta$



(d) Markov chain for computing the lower bound in Figure 5.5(b).

Figure 5.5: The effect of $\mathcal{E}$ on the ack-based protocol, $\psi = 65536$.

We now bound the probability to be in blind mode at time t, by assuming that once the attacker leaves the blind mode it never returns to it. The bound is computed using a Markov chain, where each state is the number of round-trips that have failed since the last port change. In the last state, all round-trips have failed before the exposure, and thus the attacker is no longer blind. The Markov chain for $\mathcal{E} = 4\Delta$ is shown in Figure 5.5(d). We use the chain's transition matrix to compute the probability $g(t, \mathcal{E}, C, R)$ for remaining in blind mode at time t. Figure 5.5(b) shows values of g for $\mathcal{E} = 4\Delta$. We can see that the protocol works well only for a limited time.

Finally, we note that the protocol's message complexity is 2, since it sends an ack for each message, and its bit complexity is constant: $\log_2(\psi)$ bits for the port plus κ bits for the authentication code.

5.4.2 Adding Proactive Reinitializations

We now introduce a proactive reinitialization mechanism that allows choosing new seeds for the ack-based protocol depending on time and not on the messages passed in the system. We denote by $t_A(t)$ and $t_B(t)$ the local clocks of A and B, resp., where t is the real time. From Subsection 5.2.1 we get that $0 \leq |t_A(t) - t| \leq \Phi$, $0 \leq |t_B(t) - t| \leq \Phi$. We also assume $t_A, t_B \geq 0$.

PROTOCOL ADD-ON FOR SENDER A:

Whenever $t_A(t) \in \{0, \delta, 2\delta, \ldots\}$**:**
$ack\_init(t_A(t)/\delta)$

PROTOCOL ADD-ON FOR RECIPIENT B:

When $t_B(t) = 0$**:**
Create the first ack-based protocol instance
For that instance, $ack\_init(0)$

PROTOCOL ADD-ON FOR RECIPIENT B (CONTINUED):

Whenever $(t_B(t) + 2\Phi) \in \{\delta, 2\delta, 3\delta, \ldots\}$**:**
Create a new ack-based protocol instance
For that instance, $ack\_init((t_B(t) + 2\Phi)/\delta)$

$4\Phi + \Delta$ time after creating a new ack-based protocol instance
or Δ time after receiving the first msg for this new instance**:**
Terminate all older protocol instances

Figure 5.6: Proactive reinitialization of the ack-based protocol.

If A reinitializes the ack-based protocol and then sends a message to B at time $t_A(t_0)$, this message can reach B anywhere in the real time interval $(t_0, t_0 + \Delta]$. Therefore, the port used by A at $t_A(t_0)$ must be open by B at least throughout this interval. To handle the extreme case where A sends a message at the moment of reinitialization, B must use the appropriate port starting at time $t_B(t_0) - \Phi$. (We note that t_0 may also be Φ time units apart from $t_A(t_0)$.) We define δ as the number of time units between reinitializations of the protocol, and assume for simplicity and effectiveness of resource consumption that $\delta > 4\Phi + \Delta$ (see Figure 5.6 for more details).

Every δ time units, A feeds a new seed to the ack-based protocol, and B anticipates it by creating a new instance of the protocol, which waits on the new expected ports. Once communication is established using the new protocol instance, or once it is clear that the old instance is not going to be used anymore, the old instance is terminated. The pseudocode for the proactive reinitialization mechanism can be found in Figure 5.6. We do not detail the change in port rations at the recipient's side as protocol instances are created or terminated. We also note that there is a negligible probability that more than one ack-based protocol instance will share the same port. Even if this happens, differentiating between instances can be easily done by adding the instance number (i.e., the total number of times a reinitialization was performed) to each message.

The proof of the next theorem is given in Section 5.7:

Theorem 2 *When using the ack-based protocol with proactive reinitializations, the probability that a data message that A sends to port p arrives when p is open is 1 up to a polynomially-negligible factor.*

Proactive reinitialization every δ time units allows us to limit the expected degradation in success rate for a single ack-based protocol instance. Choosing δ is therefore an important part of the combined protocol. A small δ allows us to maintain high success rate in the ack-based protocol, but increases the average number of ports that are open in every time unit (due to running several protocol instances in parallel). When several ports are used the ration for each one of them is decreased,

and so might the success rate. On the other hand, choosing a high δ entails lower success rate between reinitializations. We conclude the discussion above and the results presented in Section 5.4.1 with the following theorem:

Theorem 3 *Assume that if A sends a message to B in a single reinitialization period, then A keeps sending messages to B at least every 2Δ time units, or until that period ends. Then the success rate of the proactively reinitialized ack-based protocol with reinitialization periods of length δ is bounded from below by: $g(\delta+\Delta,\mathcal{E},C,R)\cdot f(T\Delta,C,R)$ up to a polynomially-negligible factor.*

Figure 5.5(c) shows the value of $g(\delta+1,\mathcal{E},10000,1)\cdot f(1,10000,1,1)$. We can see that the proactively reinitialized protocol's success rate stays over 90% even for $\delta=100\Delta$, i.e., even for relatively long periods between reinitializations.

5.4.3 Feasibility Discussion

A router/firewall that has IPSec support can be easily modified to support our hopping protocols. Such a router/firewall already has properties we can use: it is able to filter packets according to their SPI field, it has integrated authentication and hash functions (that can be used as PRFs), and it supports secret, shared keys. The only thing that is left to do is to perform SPI hopping. Thus, combining our hopping protocols with IPSec allows for ease of implementation, while providing IPSec's strong authentication capabilities for higher-level protocols, along with our robustness to DoS attacks, since hopping ensures that only packets that are valid w.h.p. go through the expensive authentication stage. We therefore believe that an integration of our hopping protocols with IPSec is an attractive choice.

The two-party communication protocols we presented use a shared secret, known only to the two parties. Each pair of communicating parties shares a different secret. An integration of our protocols with IPSec in tunnel mode on a gateway, means that the gateway might have to deal with several parties. The number of secrets that are stored on the gateway is thus linear in the number of parties. However, using a hash table, every SPI lookup takes $O(1)$, and so filtering is done at $O(1)$ per packet. All packets that do not contain the correct SPI are dropped at this filtering stage.

5.5 Channel Delivery Probability Analysis – Proofs of Lemmas

We now prove the lemmas from Section 5.3. Since a, C, and R are constants, denote $\mu_B(s_k)=\mu_B(a,C,R,s_k)$.

Lemma 11. *Fix k, a, C, R, and $s_k\in S_k$, and denote the ports attacked under s_k by $p_1,p_2,\ldots,p_k$ with attacking rates of $c_{p_1},c_{p_2},\ldots,c_{p_k}$, respectively. Then under a blind attack with strategy s_k, the worst (i.e., minimal) expected delivery probability of the system is achieved when $\forall i\ c_{p_i}=\frac{C}{k}$.*

Proof: By Proposition 1, $\mu_B(s_k) = \frac{\psi-k}{\psi} + \frac{1}{\psi}\sum_{i=1}^{k}\frac{R}{c_{p_i}+a}$. Calculating the partial derivatives of $\mu_B(s_k)$ we get that $\frac{\partial\mu_B(s_k)}{\partial c_{p_i}} = \frac{1}{\psi}\cdot\frac{-R}{(c_{p_i}+a)^2}$, i.e., $\mu_B(s_k)$ is monotonically decreasing as we increase c_{p_i} and keep c_{p_j} the same for $j \neq i$. Thus, the attacker wants to increase c_{p_i} to decrease the communication channel's delivery probability. However, we have the constraint $\sum_{i=1}^{k} c_{p_i} = C$. Integrating this constraint into our delivery probability function using a Lagrange coefficient denoted by β gives:

$$\mu_{B'}(s_k) = \frac{\psi-k}{\psi} + \frac{1}{\psi}\sum_{i=1}^{k}\frac{R}{c_{p_i}+a} + \beta(\sum_{i=1}^{k} c_{p_i} - C)$$

We now look for an extremum point by comparing the partial derivatives of $\mu_{B'}(s_k)$ to zero:

$$\begin{aligned}
\frac{\partial\mu_{B'}(s_k)}{\partial c_{p_i}} &= 0 \\
\frac{1}{\psi}\cdot\frac{-R}{(c_{p_i}+a)^2} + \beta &= 0 \\
c_{p_i} &= \sqrt{\frac{R}{\psi\beta}} - a
\end{aligned}$$

Putting the values of c_{p_i} into the constraint equation $C = \sum_{i=1}^{k} c_{p_i}$ gives:

$$\begin{aligned}
C &= \sum_{i=1}^{k}\left(\sqrt{\frac{R}{\psi\beta}} - a\right) \\
\beta &= \frac{R}{\psi\left(\frac{C}{k}+a\right)^2}
\end{aligned}$$

Going back to the equation for c_{p_i} we get:

$$c_{p_i} = \sqrt{\frac{R}{\psi\cdot\frac{R}{\psi\left(\frac{C}{k}+a\right)^2}}} - a = \sqrt{\left(\frac{C}{k}+a\right)^2} - a = \frac{C}{k}$$

This result also fits our constraint $c_{p_i} > 0$, and we have an extremum point for $\mu_B(s_k)$ at $c_{p_i} = \frac{C}{k}$. (We note that $\frac{C}{k}$ might not be an integer, but since we want a lower bound, this does not make a difference.) We denote this extremum point by s_k^*. Now we need to show that s_k^* is a minimum point. If we show that $\mu_B(s_k)$ is convex, then from Kuhn-Tucker Theorem we get that s_k^* is a global minimum point. We proceed by showing that $\mu_B(s_k)$ is convex.

We have already shown that $\frac{\partial\mu_B(s_k)}{\partial c_{p_i}} = \frac{R}{\psi}\cdot\frac{-1}{(c_{p_i}+a)^2}$. We get that $\mu_B(s_k)$ is twice continuously differentiable, and the second derivative is:

$$\frac{\partial^2\mu_B(s_k)}{\partial c_{p_i}\partial c_{p_j}} = \begin{cases} 0 & i \neq j \\ \frac{R}{\psi}\cdot\frac{2(c_{p_i}+a)}{(c_{p_i}+a)^4} & i = j \end{cases}$$

We get that the Hessian of $\mu_B(s_k)$ is a positive diagonal matrix. Thus, $\mu_B(s_k)$ is convex, and from Kuhn-Tucker Theorem, $\mu_B(s_k^*)$ is a global minimum of the delivery probability function $\mu_B(s_k)$. □

Lemma 12. *Let $R = a$, then an adversary with $C \geq \psi$ cannot decrease the expected delivery probability lower than $\frac{\psi a}{C+\psi a}$, and an adversary with $C \leq \psi$ cannot decrease the expected delivery probability lower than $1 - \frac{C}{\psi(1+a)}$.*

Proof: Let $R = a$. We get that $\mu'_B(k) = \frac{R-1}{\psi} - \frac{R}{\psi} \cdot \frac{C^2+(2kC+k^2R)(R-1)}{(C+kR)^2}$. We now show that $\mu'_B(k) < 0$:

$$\begin{aligned} \frac{R-1}{\psi} - \frac{R}{\psi} \cdot \frac{C^2 + (2kC + k^2R)(R-1)}{(C+kR)^2} &\stackrel{?}{<} 0 \\ 0 &\stackrel{?}{<} C^2 \end{aligned}$$

Clearly, the last inequality holds, and we get that $\mu_B(k)$ monotonically decreases as k increases. Thus, the adversary wants to choose k as large as possible. Ideally, $k = \psi$, $C \geq \psi(R - a + 1) = \psi$ and we get:

$$\mu_B(a, R, C) \geq \frac{a}{\psi} \cdot \frac{\psi^2}{C + \psi a} = \frac{\psi a}{C + \psi a}$$

However, this attack requires substantial strength from the adversary, i.e., the adversary needs to be more than ψ times stronger than B. If $C \leq \psi(R - a + 1) = \psi$ we get that $k = \frac{C}{R-a+1} = C$. The resulting degraded delivery probability is:

$$\mu_B(a, R, C) \geq \frac{\psi - C}{\psi} + \frac{a}{\psi} \cdot \frac{C^2}{C(1+a)} = \frac{\psi(1+a) - C(1+a) + aC}{\psi(1+a)} = 1 - \frac{C}{\psi(1+a)} \geq 1 - \frac{\psi}{\psi(1+a)} = 1 - \frac{1}{1+a}$$

□

Lemma 13. *Let $a < R$. Then an adversary with $C \geq \frac{\psi a}{\sqrt{\frac{R}{R-a}}-1}$ cannot decrease the expected delivery probability lower than $\frac{\psi R}{C+\psi a}$, and an adversary with $C \leq \frac{\psi a}{\sqrt{\frac{R}{R-a}}-1}$ cannot decrease the expected delivery probability lower than $\frac{\psi a - C\left(\sqrt{\frac{R}{R-a}}-1\right)}{\psi a} + \frac{R}{\psi} \cdot \frac{C\left(\sqrt{\frac{R}{R-a}}-1\right)^2}{a^2\sqrt{\frac{R}{R-a}}}$.*

Proof: Since $a < R$, we get $R \geq 2$. Let us find the value of k that minimizes the delivery probability:

$$\begin{aligned} \mu'_B(k) &= 0 \\ \frac{R-1}{\psi} - \frac{R}{\psi} \cdot \frac{C^2 + (2kC + k^2a)(a-1)}{(C+ka)^2} &= 0 \\ ak^2 + 2Ck - \frac{C^2}{R-a} &= 0 \end{aligned}$$

Since $k > 0$, we get that the solution is:

$$k = \frac{-2C + \sqrt{4C^2 + \frac{4C^2 a}{R-a}}}{2a} = \frac{-2C + \sqrt{\frac{4C^2 R}{R-a}}}{2a} = \frac{C\left(\sqrt{\frac{R}{R-a}} - 1\right)}{a}$$

Obviously, this value of k is not an integer. However, we use it to bound the minimum delivery probability under a blind DoS attack. First, we need to show that this value of k is indeed a minimum point. We do this by showing that the second derivative of $\mu_B(k)$ is always positive:

$$\mu''_B(k) = \frac{R}{\psi} \cdot \frac{2x(C+kx)\left[C^2 + (2kC + k^2 a)(a-1)\right] - (2C+2ka)(a-1)}{(C+k)^4}$$

It suffices to show that the numerator is always positive. I.e., we need to show:

$$a(2C+2ka)\left[C^2 + (2kC + k^2 a)(a-1)\right] > (2C+2ka)(a-1)$$

This is clearly true, since $a \geq 1$, $k \geq 1$, $C \geq 1$, and we get $a\left[C^2 + (2kC + k^2 a)(a-1)\right] > a - 1$. Thus, $\mu''_B(k)$ is always positive, and we have found a minimum point.

We also need the found k to be in range. Clearly, $k > 0$. We still need to show that $k \leq \frac{C}{R-a+1}$:

$$\begin{aligned} k &\overset{?}{\leq} \frac{C}{R-a+1} \\ \frac{C\left(\sqrt{\frac{R}{R-a}} - 1\right)}{a} &\overset{?}{\leq} \frac{C}{R-a+1} \\ a &\overset{?}{\leq} R - \frac{1}{R} \end{aligned}$$

The last inequality holds since $a < R$, a is an integer, and $R \geq 2$. Thus, $k \leq \frac{C}{R-a+1}$.

We can now bound the expected delivery probability $\mu(a, R, C)$ from below. For the case where $k = \frac{C\left(\sqrt{\frac{R}{R-a}}-1\right)}{a} \leq \psi$ we get:

$$\mu_B(a,R,C) \geq \frac{\psi - \frac{C\left(\sqrt{\frac{R}{R-a}}-1\right)}{a}}{\psi} + \frac{R}{\psi} \cdot \frac{\frac{C^2\left(\sqrt{\frac{R}{R-a}}-1\right)^2}{a^2}}{C + \frac{C\left(\sqrt{\frac{R}{R-a}}-1\right)}{a}a} = \frac{\psi a - C\left(\sqrt{\frac{R}{R-a}} - 1\right)}{\psi a} + \frac{R}{\psi} \cdot \frac{C\left(\sqrt{\frac{R}{R-a}} - 1\right)^2}{a^2\sqrt{\frac{R}{R-a}}}$$

For the case where $\frac{C\left(\sqrt{\frac{R}{R-a}}-1\right)}{a} > \psi$, since $\mu_B(k)$ has just one extremum point, and it is a minimum point with $k > \psi$, we get that the attacker's best strategy is to choose $k = \psi$, and we get:

$$\mu_B(a,R,C) \geq \frac{\psi - \psi}{\psi} + \frac{\psi^2 R}{\psi(C + \psi a)} = \frac{\psi R}{C + \psi a}$$

Note that we got the same result for $R = a$ and $k = \psi$. However, the conditions for choosing $k = \psi$ are different. For $R = a$ we choose $k = w$ if $C \geq w$. For $R > a$ we choose $k = \psi$ if $\frac{C\left(\sqrt{\frac{R}{R-a}}-1\right)}{a} > \psi$. $\square$

5.6 Ack-Based Protocol – Proof of Correctness

Invariant 1 *Let P_A and P_B be the P counters that A and B hold in the ack-based protocol, respectively. The probability that $P_B - P_A \in \{0,1\}$ is 1 up to a polynomially-negligible factor.*

Proof: After the initialization stage $P_A = P_B$, and the property $P_B - P_A \in \{0,1\}$ holds.
When the counters are equal, the part of the protocol that may update them proceeds as follows:

1. A sends a message to B on port $PRF_{S_{AB}}(P_A|\text{“}data\text{”})$.
2. If the message reaches B in a valid state, B adds 1 to P_B and sends an acknowledgment back to A on port $PRF_{S_{BA}}(P_B|\text{“}ack\text{”})$.
3. If the ack reaches A in a valid state, A adds 1 to P_A.

If steps 2 and 3 complete successfully, both counters advance by 1 and remain equal to each other. If step 2 fails (message dropped or modified in transit), both counters remain unchanged. If step 2 succeeds but step 3 fails (ack lost or changed in transit), P_B is incremented by 1, but P_A remains the same. Thus, if $P_A = P_B$, the next change of counters will still maintain the property $P_B - P_A \in \{0,1\}$.

Now suppose we have reached the state where $P_B = P_A + 1$. The portion of the protocol that may update the counters proceeds as follows:

1. A sends a message to B on port $PRF_{S_{AB}}(P_A|\text{“}data\text{”})$.
2. If the message reaches B in a valid state, B sends an ack back to A on port $PRF_{S_{BA}}(P_B - 1|\text{“}ack\text{”})$.
3. If the ack reaches A in a valid state, A adds 1 to P_A.

If steps 2 and 3 complete successfully, P_A advances by 1 and the counters become equal to each other. If steps 2 or 3 fail (messages dropped or are not valid), both counters remain unchanged. Thus, if $P_B = P_A + 1$, the next change of counters will still maintain the property $P_B - P_A \in \{0,1\}$.

The only way to break this invariant is if the attacker makes just one party advance its counter. This means that the adversary has to fabricate a message so one party will think it is valid. Thus, the attacker needs to guess both the port number and the authentication information attached to each message. The probability that the attacker succeeds in doing so is a polynomially-negligible factor. □

Theorem 1. *When using the ack-based protocol, the probability that a data message that A sends to port p arrives when p is open is 1 up to a polynomially-negligible factor.*

Proof: According to Invariant 1, when A sends a data message to B, either $P_A = P_B$ or $P_B = P_A + 1$, with probability 1 up to a polynomially-negligible factor.

For the first case, let M be a message A sends to B when $P_A = P_B$. Since B always opens two ports for data, we need to show that P_B does not increase by more than one until M actually reaches B. Since the link maintains the FIFO semantics, messages sent after M was sent cannot change the value of P_B before M reaches B. The only messages that can change P_B are messages that preceded M but reached B only after M was sent.

According to the protocol, P_B increases by one iff B receives a data message from A that was sent using the counter $P_A = P_B$. Furthermore, all messages preceding M were sent using a counter that is less than or equal to P_A. It follows that P_B can only increase by one from the time M leaves A until it reaches B.

Consider now the second case where M was sent when $P_B = P_A + 1$. Since B only opens two ports for data, we need to show that P_B does not change at all. Again, since the link has FIFO semantics, P_B can only change by messages preceding M that reach B after M was sent but before it reaches B. However, such messages have counters that are less than or equal to P_A, and thus strictly less than P_B. According to the protocol, messages sent with such counters do not affect the value of P_B. □

5.7 Ack-Based with Reinitializations – Proof of Correctness

Theorem 2. *When using the ack-based protocol with proactive reinitializations, the probability that a data message that A sends to port p arrives when p is open is 1 up to a polynomially-negligible factor.*

Proof: From Theorem 1 we get that if A and B both use the ack-based protocol initialized with $seed$, then messages sent by A arrive to open ports at B. To complete the proof, we need to show the following:

1. When A reinitializes the protocol with a new seed, B has already started running an ack-based protocol instance using the same seed.

2. B does not terminate a protocol instance while it may still receive messages corresponding to that instance.

For the first property, let us look at some real time t_n^A when A reinitializes the protocol, where $t_A(t_n^A) = n\delta$, $n \in \mathbb{N}$. From the bounded drift assumption we get the bound $t_n^A \geq n\delta - \Phi$. The seed corresponding to the initialization at t_n^A is $\frac{t_A(t_n^A)}{\delta} = n$. Now let us look at the real time t_n^B in which B starts a new ack-based protocol instance with the seed n. This happens when $t_B(t_n^B) + 2\Phi = n\delta$, i.e.,

when $t_B(t_n^B) = n\delta - 2\Phi$. Using the bounded drift assumption we get the bound $t_n^B \leq n\delta - 2\Phi + \Phi = n\delta - \Phi \leq t_n^A$.

For the second property, let us look at seed n again. A terminates the instance with seed n at real time t_{n+1}^A. The last packet sent using the ack-based protocol initialized with seed n inevitably reaches B before real time $t_{n+1}^A + \Delta$. B terminates the ack-based protocol instance in either one of the following two cases:

1. At time $t_B(t_{n+1}^B) + 4\Phi + \Delta$.
2. Δ time units after receiving the first message for a newer ack-based protocol instance.

For the first case, we get $t_{n+1}^B \geq (n+1)\delta - 2\Phi - \Phi + 4\Phi + \Delta = (n+1)\delta + \Phi + \Delta \geq t_{n+1}^A + \Delta$. For the second case, we observe that if a message for a newer instance of the ack-based protocol has arrived, then A is no longer sending messages with instances initialized with older seeds. However, the varying message propagation delay means that messages from older protocol instances can take up to Δ time units to arrive, while the new message might have taken negligible time to arrive. $\square$

Chapter 6

ϕ-Hopper Implementation and Measurements

The effectiveness of a DoS-mitigation solution can be quantified through mathematical analysis and empirical results. These two methods complement each other, as the analysis can provide results for *all* possible attacks, but these results are only applicable for a *model* of the system, which may or may not correctly reflect the real world. Both Chapter 5 and this chapter deal with using hopping to mitigate the effects of DoS attacks. But while Chapter 5 concentrated on mathematical analysis and modeling, this chapter focuses on actual implementation and concrete measurements.

The ϕ-Hopper implementation presented here is a refinement of the port-hopping mechanism that we presented and analyzed using a simplified network model in Chapter 5. The refinement supports communication from many clients to a server (can be extended to a server farm). We describe an implementation of ϕ-Hopper in two variations: (1) by modifying a Linux kernel's IPSec [3] implementation, and (2) by inserting code in a Windows NDIS layer. Our experimental results validate the analytical results presented in Chapter 5.

Cryptographic mechanisms ϕ-Hopper uses efficient, shared-key pseudorandom functions. Our usage of these cryptographic mechanisms is standard. Therefore, in this chapter, we omit their definitions and simplify their behavior; for details and definitions see, e.g., [18]. We explicitly use the following mechanisms in this chapter:

Pseudorandom function (PRF) PRF**:** a function $PRF : \{0,1\}^\kappa \times D \rightarrow R$, with range R and two parameters: key $k \in \{0,1\}^\kappa$ and data $x \in D$. The key k is a (random) binary string of sufficient length κ, e.g., $\kappa = 100$. The function is *pseudorandom* if it cannot be efficiently distinguished from a random function r from domain D to range R. Namely, let $g_{b,k}(x) = \{r(x)\ if\ b = 1, PRF_k(x)\ if\ b = 0\}$. Given 'black box' (oracle) access to $g_{b,k}$ for random b, k, r, feasible adversaries cannot guess b with significant advantage (over $\frac{1}{2}$).

For simplicity, for the rest of this chapter we neglect the probability that the adversary can forge the PRF without knowing the secret key.

6.1 ϕ-Hopper

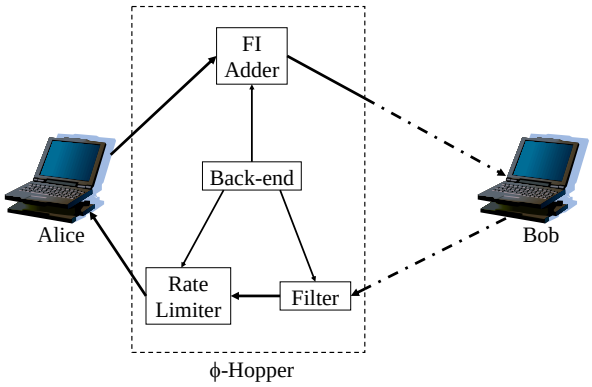


Figure 6.1: Communicating using ϕ-Hopper (Alice's view).

ϕ-Hopper leverages existing, cheap, network-level packet-filtering and rate-limiting solutions, along with more complex algorithms at a higher layer, which determine the filtering criteria and rate limits. Filtering is based on a *filtering identifier* (FI, or ϕ), which is some message field value that can be changed by the communicating parties, and is preserved en route. For example, it can be a combination of IP address and ports [30], as shown in Chapter 5, or IPSec's security parameter index (SPI) field [3]. The FI can also be an artificial field appended to the message. The FI's size can be set according to the wanted DoS-resistance guarantees.

At each communicating party, ϕ-Hopper has two parts: a *front-end* that performs fast packet-filtering, rate-limiting, and FI adding, and a *back-end* that controls the front-end's parameters, e.g., filtering criteria and rates. Figure 6.1 shows the decomposition of ϕ-Hopper and the interaction between its various components.

The two parties wishing to communicate share a secret. This secret is used to create pseudo-random sequences of FIs. Each message transmitted between the parties carries a FI taken from an appropriate pseudorandom sequence. The receiver's front-end anticipates the FI according to the pseudorandom sequence, and filters out all messages carrying invalid FIs. The FIs change in order to maintain DoS-resilience. Otherwise, the adversary could eavesdrop on messages and discover the FI in use. Hopping using an appropriate FI size ensures that with high probability, the adversary cannot discover the FI (see Chapter 5).

The front-end The front-end can be a gateway or firewall, a layer in the end host's protocol stack, or even a dynamically programmable NIC that allows fast filtering at wire-speed [57]. In fact, the front-end's components do not all have to be deployed on the same machine. The first

component is simple and handles fast filtering of incoming packets. Its purpose is to defend the recipient from being flooded with spoofed messages.

The second front-end component rate-limits incoming valid traffic according to its source. The rationale behind this component is that registered clients can also get corrupted, or try to receive better service at the expense of other valid clients. The rate-limiter ensures that the server does not receive more requests than it can handle, and that all clients receive their fair share of the server's time.

We use two types of rate-limiters: simple and round-robin (RR) based. When using the simple rate-limiter, each source is allocated a maximum allowed rate that can change during the session. This method is simple and fast. For example, a client may be allowed to send 10 requests every second. Note that when the server performs costly processing per each client request, the rate that needs to be limited is the rate of incoming requests, and not the rate of incoming bytes. Our simple rate-limiter approximates this by counting packets (indeed, in our experiments, each packet corresponds to a single request). However, even if the average rate of requests is adequate, but the client sends its traffic as bursts, packets will get dropped.

The RR late-limiter strives to use resources more efficiently, by sharing them among all clients. In RR rate-limiting, each source-destination pair has limited-size queues for incoming/outgoing messages. Messages arriving to a full queue are dropped. ϕ-Hopper sends messages from the queues to their destination in a RR fashion, provided that the total maximum allowed rate of messages is not exceeded. If a queue is empty, it is skipped for that RR cycle. This is very similar to Fair Queueing, which uses RR at the byte level [50]. RR rate-limiting handles bursty traffic well, but incurs an increase in latency, due to its periodic and cyclic nature. The importance of using RR to compensate for bursts of one client with idle time of others increases with the number of clients in the system.

Rate-limiting has been dealt with extensively in the literature [15, 50, 34], and the usage of rate-limiting with ϕ-Hopper is orthogonal to ϕ-Hopper's other functions. We therefore do not conduct a more detailed study of rate-limiting in this chapter beyond illustrating that ϕ-Hopper can work with various techniques.

The third front-end component is quite trivial, as it only adds the appropriate FI to outgoing packets, so that they will be accepted by the recipient.

The back-end Figure 6.2 shows the pseudocode for ϕ-Hopper's back-end. Each party communicating via ϕ-Hopper maintains a *virtual time* (line 5), which determines its current position in the pseudorandom sequence for outgoing messages (lines 7 and 16), and for incoming messages (lines 8 and 17). Every fixed time interval δ, ϕ-Hopper performs a *hop* (line 13), which locally changes the virtual time (line 15). A ϕ-Hopper session between two parties is initialized using a seed that is used as the initial virtual time, and a shared secret key used for generating the pseudorandom

(1) Initially:
(2) $\forall B \ \ out(B) \leftarrow \perp$
(3) $\forall B \ \ in(B) \leftarrow \perp$

(4) *initHopperSession(seed, key, B)*
(5) $virt(B) \leftarrow seed$
(6) $key(B) \leftarrow key$
(7) $out(B) \leftarrow PRF_{key(B)}(virt(B)||A||B)$
(8) $in(B) \leftarrow in(B) \cup PRF_{key(B)}(virt(B)||B||A)$
(9) Set timer('close', B, $virt(B)$) to $closeTimeout$
(10) Inform rate-limiter of '$initSession(B)$'

(11) **On** wakeup of timer('close', B, $virt$)
(12) $in(B) \leftarrow in(B) \backslash PRF_{key(B)}(virt||B||A))$

(13) *every δ time units*
(14) **for all** B s.t. $out(B) \neq \perp$ **do**
(15) $virt(B)$++
(16) $out(B) \leftarrow PRF_{key(B)}(virt(B)||A||B)$
(17) $in(B) \leftarrow in(B) \cup PRF_{key(B)}(virt(B)||B||A))$
(18) Set timer('close', B, $virt(B)$) to $closeTimeout$

(19) *endHopperSession(B)*
(20) $out(B) \leftarrow \perp$
(21) $in(B) \leftarrow \perp$
(22) Inform rate-limiter of '$endSession(B)$'

Figure 6.2: ϕ-Hopper's back-end protocol for A (communicating with B).

sequence (line 4).

During session initialization, each party allocates bounded resources for communication in this session. ϕ-Hopper allocates separate resources for each active client (line 10), which are freed when the session for that client ends (line 22). Whenever a client becomes active/inactive, resources allocated to other clients might change, e.g., to achieve fairness or better utilization of the server. We note that, in general, since the server seperately allocates bounded resources for each active client, compromised clients cannot significantly drain the server's resources by sending it an excessive number of requests, and thus valid clients get their share of the server's resources.

We say that each party *opens FIs* for communication when these FIs are added to the list of valid FIs (lines 8 and 17), and *closes FIs* when these FIs are invalidated (line 12), $closeTimout$ time after they were created (lines 9 and 18). ϕ-Hopper uses two parameters that determine $closeTimeout$: Δ, the message latency, and Φ, representing the *synchronization gap* between the parties. Roughly

speaking, the synchronization gap is the maximum differential between the times at which the parties decide to open the same FI in the pseudorandom sequence. It is the sum of the difference between the session starting time and the maximum clock drift during a ϕ-Hopper session. If the session time is so long that the clock drift might become a problem, i.e., Φ is too big, reinitialization is needed.

To compensate for the loose time synchronization between the paries, each party keeps multiple open FIs at the receiving end, corresponding to all virtual times the other party might be in. The recipient opens a new FI every δ time, and closes a FI $4\Phi + \Delta$ time after opening it, i.e., $closeTimeout = 4\Phi + \Delta$. For example, if $\Delta = 100$ms, $\delta = 200$ms, and $\Phi = 250$ms, we get that there are at most 6 open FIs at a given time. A simple optimization that reduces the number of open FIs is closing a FI (if it is still open) Δ time after receiving a message on the next FI in the sequence.

6.2 Implementation and Measurements

Implementation We present two implementations of ϕ-Hopper. The first installs the front-end on gateways as a modified IPSec layer in a Linux kernel. The IPSec layer operates in tunnel mode, and the FI is the 32-bit SPI field. IPSec first checks the SPI, and if it is valid, performs authentication using HMAC-SHA-1. This is also the setting for our rate-limiting experiments, where the IPSec gateway performs the rate-limiting. The second implementation installs the front-end on the communication end-points as an NDIS hook driver on a Windows system, and checks packets for an appended 160-bit FI. The hook only filters packets, and authentication is performed by the server, using a simple SHA-1 hash of the data and the secret key. This simulates server-side authentication, as done, e.g., in SSL. In both scenarios, we install the back-end on the same machine as the front-end.

Essentially, systems that use ϕ-Hopper do not need to perform cryptographic per-packet authentication to ensure that the probability of receiving invalid messages is negligible. This means that the processing of packets is fast. We use authentication in our experiments to show that even if a system requires authentication, it is better off using ϕ-Hopper as its DoS-prevention method, rather than relying solely on the authentication mechanism to filter DoS-attackers.

Our implementations use a shared secret of 160 bits. At each FI hop, we increase the virtual time by 1, and calculate the 160-bit SHA-1 hash of the current virtual time concatenated with the shared secret. We then truncate the hash value to fit the FI's size.

In our IPSec implementation, at each hop we add new entries to IPSec's list of valid states, and remove old states from the list. An IPSec state consists of a security association (SA) for two end-points. We utilize IPSec's tunnel mode to encapsulate the end-points' packets on their path between the gateways. The states we add have the same SA as the previous states for that session, except for a changing SPI. In our NDIS implementation, we simply save a list of all valid FI values per client,

and update this list every hop.

ϕ-Hopper is easy to implement and deploy. Our prototype implementations take only a few hundred lines of simple code.

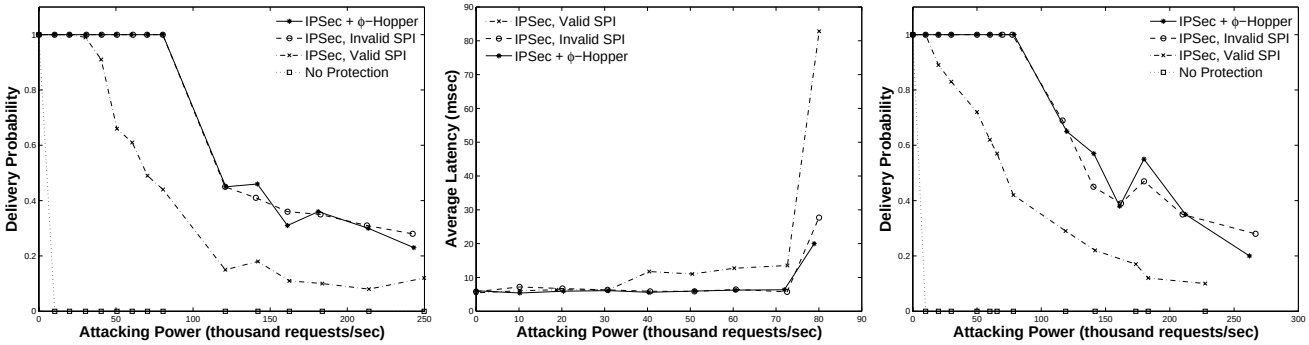
Measurements We measure the effect authentication and hopping have on the resistance to DoS. We experiment with a TCP/UDP HTTP server, an appropriate client, and an adversary (implemented using one to three machines), all connected to a 100Mbps LAN through a switch. In each experiment, the adversary sends bogus requests at an average constant rate to the web server. At the same time, the client sends valid requests to the server. The server processes each request, and dynamically forms a response, while consuming CPU power. Every UDP request or response fits into a single UDP packet. We measure the latency (round-trip time), and delivery probability, i.e., the probability that a client's valid request is processed by the web server, as a function of the attacker's strength. Each data point represents 100 experiments.

UDP/Linux In our first setting, we measure the advantages ϕ-Hopper offers, as compared to IPSec [3], when deployed on gateways. For this setting, we have a client, connected to gateway A, where gateway A is connected to gateway B, which in turn is connected to a web server. The gateways run Linux with IPSec in tunnel mode, with or without ϕ-Hopper installed, according to the experiment. The gateways have a Pentium 3 650MHz CPU, and 256MB of RAM.

We compare 4 different scenarios: (1) The server has no DoS protection at all; (2) the gateways run IPSec in Authenticated Header (AH) mode, and the adversary knows the SPI used; (3) the gateways run IPSec in AH mode, and the adversary does not know the SPI used; and (4) the gateways run IPSec in AH mode with ϕ-Hopper. When attacking, the attacker sends bogus requests at a constant rate. In scenario (2), the bogus requests carry the correct SPI field, but fail authentication. In scenarios (3) and (4), the bogus requests carry an incorrect (arbitrary) SPI field (w.h.p., for scenario (4)), and so the bogus requests do not reach the authentication phase.

Scenario (3) protects the server well from DoS attacks as long as the SPI used cannot be easily guessed, and the session time is short. However, if the session time exceeds the exposure delay $\mathcal{E}$, the adversary has ample time to discover the SPI, e.g., by ARP-poisoning a LAN, or by sniffing packets in intermediate routers. Once the adversary obtains the SPI, scenario (3) transforms into scenario (2). Since we assume relatively long sessions, we include scenario (3) mainly to quantify the overhead of port hopping.

Figure 6.3(a) depicts the delivery probability as the attacker's strength increases. We see that ϕ-Hopper achieves the same delivery probability exhibited when the adversary does not know the SPI used, as filtering in these cases is based on a simple comparison of a header field. However, ϕ-Hopper does not rely on keeping the SPI, which is sent in the clear, secret. ϕ-Hopper signicantly outpeforms IPSec when the SPI is compromised. The delivery probability is much lower when the SPI is known to the attacker, since this case requires complete authentication of every packet.



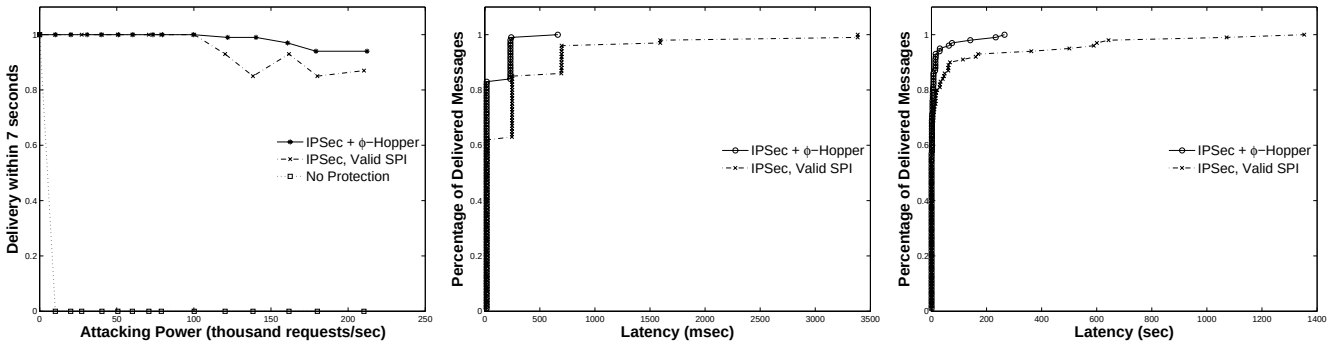
(a) Delivery probability. (b) Latency (RTT). (c) Delivery probability, bursty attacker.

Figure 6.3: DoS attacks on IPSec on Linux, with and without ϕ-Hopper (UDP). ϕ-Hopper achieves the same results as IPSec with an unknown SPI, *without* requiring the cleartext SPI to remain secret.

This difference is most evident for relatively weak attacks ($80{,}000$ requests/sec), where ϕ-Hopper maintains 100% delivery, but the delivery for IPSec with a known SPI drops sharply to 44%. We can further see that having *any* form of protection is better than having no protection at all. When the server has no protection, it crashes even when the attack is very weak, reducing delivery probability to 0.

Figure 6.3(b) shows the effect of increasing-strength attacks on latency. In this experiment the server does not really process the request, but rather returns a reply immediately. We measure this parameter since we want to isolate the effect the algorithms run by the gateways have on latency. We can see that unless the SPI is known, the latency stays the same even when the attack strength increases, with some slight incline for severe attacks. Additionally, the latency is virtually equal for ϕ-Hopper and for IPSec when the SPI is unknown. This is also the same latency measured when IPSec and ϕ-Hopper do not run at all (not shown on graph). Thus, as opposed to overlay networks, ϕ-Hopper ensures DoS-resilience with no or small penalty in latency. Conversely, when only IPSec is used and the SPI is known, the latency exhibited is double the one for ϕ-Hopper even for mild attacks, and it increases significantly for more severe attacks. Since the delivery probability is low for attacks stronger than the ones plotted, it is meaningless to calculate the latency for such attacks.

Figure 6.3(c) displays the delivery probability under a bursty DoS attack, where bogus requests are not sent at constant intervals, but rather as bursts. The attack strength is measured as the average number of bogus requests per second. Comparing these results to Figure 6.3(a), we observe that a bursty attacker induces less damage than an attacker whose sending times are uniformly distributed over time. This can be explained by the fact that at times in which the attacker does not send any bogus message, the client's requests can be easily processed.



(a) Delivery within 7 secs. Latency CDFs: (b) Attack: $100{,}000$ requests/sec.(c) Attack: $240{,}000$ requests/sec.

Figure 6.4: DoS attacks on IPSec on Linux, with and without ϕ-Hopper (TCP).

TCP/Linux Having seen the benefits ϕ-Hopper offers for UDP traffic, we turn to test its effects on TCP traffic. We start by noting that using TCP with no IPSec protection is problematic for two reasons: First, if the adversary discovers or guesses the sequence numbers used in TCP, it can bring down the connection by sending a single RST packet [56]. We validated this in an experiment in which our attacker software sniffed a single packet sent by the client to the server over a TCP connection. The attacker then discovered the sequence numbers used in the TCP connection, and sent a single RST packet to the server with an appropriate sequence number. This packet brought down the connection immediately, as that is the purpose of an RST packet, when the sequence number falls inside the range of numbers the server expects.

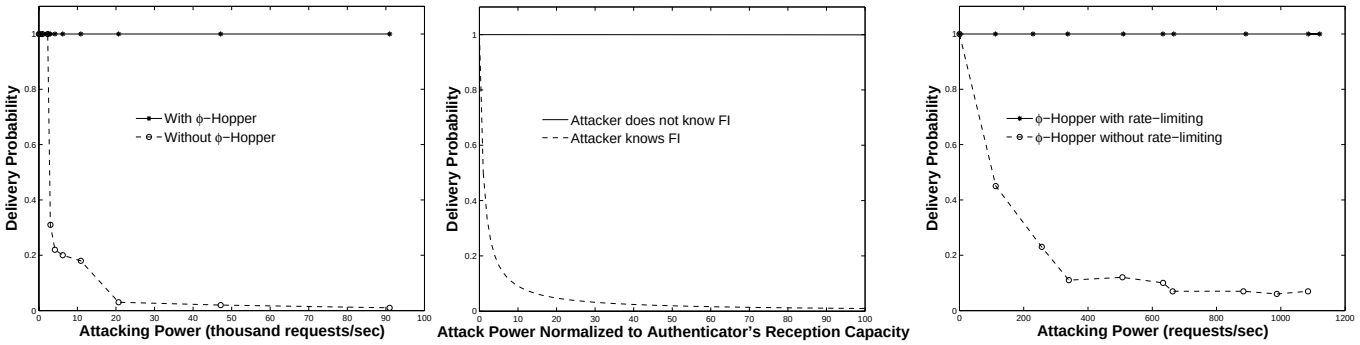
The second problem when using TCP without client authentication, is that bogus clients can connect and overload the server. Thus, for both reasons, TCP without authentication is insufficient. We therefore experiment with TCP over IPSec with AH, as in our UDP setting.

Figure 6.4(a) shows the delivery probability of TCP traffic over IPSec, with and without ϕ-Hopper. TCP's retransmission mechanism ensures that all messages eventually arrived to their destination. The figure shows the percentage of requests for which the client receives a response within 7 seconds of the moment the request was sent. As expected, when no protection is in use, the server crashes due to the heavy load. We can see that using ϕ-Hopper provides better delivery probability compared to IPSec with a compromised SPI, for attacks stronger than $100{,}000$ requests per second. For weaker attacks, all packets are delivered within 7 seconds in both scenarios.

Figure 6.4(b) shows the cumulative distribution function (CDF) of TCP latencies (RTT) for ϕ-Hopper and IPSec with a compromised SPI, for an attack power of $100{,}000$ requests per second. We can see that ϕ-Hopper provides better RTTs than IPSec with a compromised SPI. While over 80% of the messages passing through ϕ-Hopper had no latency penalty (cf. data point 0 in Figure 6.3(b)), IPSec managed to deliver only 60% of the messages with no delay. This corresponds to about 20%

message loss in the first transmission when using ϕ-Hopper, compared to about 40% message loss in the first transmission for IPSec with a known SPI (cf. Figure 6.3(a).) Furthermore, ϕ-Hopper managed to deliver 99% of the messages within 250 msecs, while IPSec delivered only about 82% of the messages by that time, and had overall delays of up to 3.5 seconds in delivery. We can clearly see TCP's exponential backoff in action, as delays get about 2 times longer for each retransmission.

Figure 6.4(c) depicts the CDF of TCP latencies for a stronger attack, of $240{,}000$ requests per second. Notice that the latency in the figure is given in *secs*, and not in msecs, as before. The figure clearly shows that ϕ-Hopper provides reasonable latency for 85% to 90% of the messages, while IPSec's latency starts deteriorating at about 75% to 80%. Moreover, the delivery of some messages in IPSec takes over 20 minutes – about 4.5 times worse than the longest delay in ϕ-Hopper.



(a) Windows NDIS layer. (b) Theoretical values. (c) 1 compromised/1 valid client (IPSec).

Figure 6.5: Delivery probability under DoS attacks.

UDP/Windows In our second setting, the client communicates directly with the web server, and we measure the effect ϕ-Hopper has when it runs on the server's machine, and not on a dedicated machine. The server runs on a Windows machine along with ϕ-Hopper (in the appropriate experiments), which performs user-level filtering of packets through a kernel-level NDIS hook driver. The server machine has a Pentium 4 3.2GHz CPU, and 1GB of RAM.

Figure 6.5(a) shows the delivery probability with and without ϕ-Hopper, where authentication is performed at the server. At a relatively weak attack strength ($6{,}200$ requests/sec) there is a dramatic drop in delivery to 20% when ϕ-Hopper is not used, whereas ϕ-Hopper allows for 100% delivery even for much stronger attacks. Here too, attacking an unprotected server crashes it (not shown in figure).

Theoretical Values We compare our results to analytical results for the delivery probability under DoS attacks, as taken from Chapter 5 (see Figure 6.5(b)). The theoretical analysis assumes the attacker's sending times are uniformly distributed, and thus the results shown in the figure can be compared to figures 6.3(a) and 6.5(a). Indeed, we can see that the actual measurements closely

match the theoretical analysis.

Rate-Limiting Figure 6.5(c) shows the effect of rate-limiting on the delivery probability for UDP traffic on IPSec/Linux. In this experiment, we have two clients: one valid client, and one compromised client. The valid client sends requests at a rate of 10 requests per second. The compromised client tries to load the server. We measure the delivery probability for the valid client as a function of the rate of requests sent by the compromised client. We can see that when rate-limiting is not enforced, the delivery probability drops rapidly due to the load on the server. Limiting the rate of each client to at most 12 requests per second suffices to ensure a delivery probability of 1.

We now turn to compare this simple rate-limiting to the round-robin-based rate-limiting algorithm. In our experiments, we have 3 clients that send 100 messages per second on average, for a total of 1000 messages each. All clients send their messages either at constant intervals, as a Poisson process, or as bursts. The effectiveness of the simple rate-limiting and the RR rate-limiting techniques are measured in these 3 scenarios, for a total of 6 experiments. The total rate allowed by the server is set to 315 messages per second. When using simple rate limiting, we allow each client a rate of 105 messages per second. For RR rate-limiting, we give each session a queue of 300 messages, and wake the RR dispatcher every 100 ms. The dispatcher sends messages from the queues in a cyclic fashion, and goes back to sleep after sending roughly 30 messages, or when all the queues are empty.

Table 6.1 shows the difference in delivery probability and latency (RTT) for a client chosen arbitrarily from the 3 clients communicating with the server. We can see that, although RR rate-limiting imposes higher latency due to its periodic and cyclic nature, it handles bursty traffic much better than simple rate-limiting. While the delivery probability drops down to 11% for simple rate-limiting in conjunction with bursty traffic, RR rate-limiting manages to deliver all messages contained in the bursts. RR rate-limiting's superiority is achieved beacause RR allows all queues to share a single pool of resources, and so if a queue is empty, the other flows gain better maximum rates.

Our rate-limiting experiments show the flexibility and modularity of ϕ-Hopper. ϕ-Hopper works well with different rate-limiting approaches suitable for various systems. Of course, one can employ more elaborate rate-limiting approaches as well [15, 50].

Sending Type	**Simple Rate-Limiting**			**Round-Robin Rate-Limiting**		
	Del. Prob.	**Avg. RTT (ms)**	**Std. Dev.**	**Del. Prob.**	**Avg. RTT (ms)**	**Std. Dev.**
Constant	1	0.925	0.07	1	148.82	0.33
Poisson	1	0.89	0.05	1	150.45	2.38
Bursty	0.11	3.06	0.32	1	632.83	20.147

Table 6.1: Simple vs. RR rate-limiting.

6.3 Related Work

The idea of repeatedly changing authentication credentials to avoid suffering damage due to exposure, has been used in different contexts, e.g., in the S/KEY authentication method [21]. ϕ-Hopper is based on ideas that have been suggested in Chapter 5 and in [30]. However, these previous suggestions lacked in several areas, and so ϕ-Hopper differs from them in the following ways:

1. Instead of using the current time as the seed to the pseudorandom sequence, the initial seed used to start the sequence is given to the protocol and used as virtual time. Thus, there should only be means for the parties to share the seed (which is not secret), and no time synchronization between the communicating parties is needed, but rather a bounded clock drift.
2. ϕ-Hopper supports communication between many clients and a single server, and not just two-party communication.
3. ϕ-Hopper uses realistic rate-limiting techniques, as opposed to the purely theoretical analysis in Chapter 5 that assumed a simplified model of rate-limiting at the network level. Additionally, rate-limiting is performed per client, and not per FI. The protocol described in [30] uses no rate-limiting at all.
4. ϕ-Hopper is implemented in two variations, and we provide measurements of the actual protocol implementation, and not of its simulated behavior [30] or of an analytical anaysis of the protocol (as given in Chapter 5).

The analysis in Chapter 5 shows that the basic idea of hopping is very effective against DoS attacks, but does so under simplified network and rate-limiting models. Other work *simulates* the effect port-hopping has on the delivery probability under attack, and shows that using it is expected to decrease the load on the server [30]. In Section 6.2 we have shown that the analysis in Chapter 5 gives a good estimate of realistic results, using a real implementation of all of ϕ-Hopper's components. Our results not only show that ϕ-Hopper provides strong resistance against DoS attacks, they also show that relying merely on authentication to provide DoS protection is futile.

Chapter 7

Beaver

We consider the problem of protecting legacy servers from (Distributed) Denial of Service (DoS) attacks by realistic adversaries. There are many methods for DoS attacks, e.g. exploiting different application vulnerabilities. We focus on the family of *flooding* DoS attacks, which try to disrupt services by sending a very large number of packets concurrently (a "flood"). To obtain sufficient bandwidth and foil filtering, these attacks often originate from many clients concurrently; this is referred to as a *Distributed Denial of Service (DDoS) flooding attack*. Currently, attackers are often able to control a very large number of corrupted personal computers ("zombies"), resulting in many DDoS flooding attacks, and significant over-allocation of networking resources as a crude, wasteful defense mechanism. Our goal is to investigate more efficient defense mechanisms, which will avoid excessive costs or overheads (e.g., no significant added latency).

Existing DoS solutions deployed in firewalls or gateways typically use two methods: filtering according to packet header fields like addresses and ports, and rate-limiting traffic. These simple methods are very efficient, but are insufficient. Header fields can be spoofed to match filtering criteria. Cryptographically-authenticated traffic cannot be spoofed, but causes significant overhead to all traffic. Rate-limiting of legitimate traffic along with spoofed traffic is not effective, as valid packets are indiscriminately discarded, (esp. when applications are very sensitive to packet loss, e.g., due to TCP's congestion control mechanism).

Our measurements show that even when the network is not loaded, a large number of bogus requests can kill a server that does not require authentication, and can virtually drop to zero the reliability of client-server communication when the server does require authenticated requests. In this chapter we address this challenge – we present Beaver, a highly-efficient, low-overhead filtering mechanism to resist spoofed packets. This mechanism avoids expensive cryptographic authentication of each packet, by requiring packets of each legitimate client to include a "fresh", pseudorandom *Filtering Identifier (FI)* ϕ.

In most DDoS flooding attacks, the attacker controls and utilizes a very large number of corrupted computers ("zombies"), but a much smaller number of legitimate user accounts. That is why it is important to make sure only authorized clients are allowed to communicate with the server, by using registration and admission procedures.

Beaver is composed of two main components: (1) ϕ-Hopper – enforces filtering and rate-limiting, using the filtering identifier ϕ. (2) The admission servers – responsible for registering clients and authorizing new client-server sessions. Beaver's components can be implemented in several different ways, depending on the required deployment scenario and capabilities.

7.1 Design Goals

We consider the problem of protecting the following basic client-server communication from DoS attacks:

- A client registers with the system before being able to use it. During the registration process, the client may receive a unique secret to allow the server to authenticate its requests. We assume the use of public/private key pairs and certificates at this stage.
- A server, or a server farm, provides service to authorized clients. Client-server sessions are relatively long, and consist of several transactions, potentially using authenticated communication.

The number of registered clients may be very large, e.g., $1{,}000{,}000$, but it is expected that only a small number of them, e.g., $1{,}000$, will communicate with the server simultaneously. These basic properties are found in many web-based services, e.g., banking, stock trading, and online auctions. DoS attacks on these services may degrade the service so much that clients lose money due to its unavailability.

Our goals in protecting the basic system against DoS-attacks are as follows:

- *Session DoS-resistance*. Protect ongoing client-server sessions. Moreover, separate the "war zones" – attacking the admission and registration processes should not affect ongoing sessions.
- *Admission DoS-resistance*. Protect the admission process in which registered clients create new sessions with the server.
- *Best-effort registration availability*. Implement a registration process that allows new clients to obtain the required shared secrets, but allow this service to degrade due to DoS.
- *Fast communication*. Do not harm communication latency for established client-server sessions.

One might argue that authenticating client-server communication alone is enough to filter out invalid packets sent by DoS attackers. But although authentication is enough to discriminate bogus

messages from valid ones, the validation itself is costly. This is especially a problem if the server is the one performing the validation, as happens when using SSL. Since the server should be mainly busy with answering requests, we would like to minimize the number of invalid packets that reach the server and cause extra processing. Our measurements in Chapter 6 show that by avoiding per-packet authentication we can resist much stronger DoS attacks.

7.2 Environment and Adversary Assumptions

Timing and communication Beaver is implemented in a realistic network setting, where all parties have monotonically increasing clocks, local events may be scheduled according to local time (clock value), and timestamps may be read from clocks. Clocks are not synchronized among parties, but the synchronization gap, Φ, is bounded. Messages arrive within Δ seconds from the moment that they are sent, or are otherwise considered lost. Beaver is implemented at the level of datagram protocols such as raw IP, where message latency is generally bounded, and some messages are lost. Higher-level protocols such as TCP compensate for message loss using retransmissions.

Adversary In a DDoS attack, the attacker often controls many compromised workstations ("zombies"), from which it sends its attack traffic. Beyond controlling these zombie machines, which are not part of the system, we assume that the adversary can also control some of the clients and admission servers. The set of machines the attacker controls determines its capacity for sending messages, and its a-priori knowledge of the private information used in the system. The number of messages the adversary can send per second is bounded by a parameter, C, representing its capacity. The adversary has a global view of the compromised machines, but it cannot modify messages sent by correct parties. As the purpose of Beaver is essentially to protect the server, there is no point in assuming the server may be compromised. Furthermore, we assume that machines that are not communication endpoints, e.g., gateways and routers, are not compromised.

Some of the zombies may be able to eavesdrop to some of the legitimate traffic. For example, a zombie may be able to eavesdrop, when it is on the same LAN as the server or as a legitimate client (e.g., by deploying an ARP-poisoning attack). However, even if the attacker controls a proxy which can eavesdrop on all traffic, it would still incur a substantial delay in forwarding the captured information to many zombies, so as to generate enough traffic. We assume a lower bound of $\mathcal{E}$ seconds from the time a message is sent until its contents can be incorporated into the adversary's decision process. $\mathcal{E}$ is called the *exposure delay*. Once the adversary decides to act, though, it may send arbitrarily crafted messages with zero latency.

We assume that feasible adversaries are bounded in their computational resources (e.g., probabilistic polynomial time machines).

Cryptographic mechanisms Beaver uses cryptography to protect against feasible adversaries.

Whereas for the registration process we use public key signatures and encryption schemes, for ϕ-Hopper and the admission process we use efficient, shared-key pseudorandom functions. Shared key cryptography is much more efficient, and often implemented by one or two applications of a cryptographic hash function (such as SHA-1).

Our usage of these cryptographic mechanisms is standard. Therefore, in this chapter, we omit their definitions and simplify their behavior; for details and definitions see, e.g., [18]. We explicitly use the following mechanisms in this chapter:

Public key encryption scheme $\langle KG, E, D\rangle$**:** decryption recovers plaintext, i.e., for every message m and random keys $\langle e, d\rangle \in KG(1^\kappa)$ holds $m = D_d(E_e(m))$. Furthermore, adversary does not learn anything from ciphertext. Namely, for every feasible adversary A and every two messages m_L and m_R holds $Prob\left(A(e, E_e(m_L)) = L\right) < Prob\left(A(e, E_e(m_R)) = L\right) + \epsilon(\kappa)$, where ϵ is a negligible function.

Public key signature scheme $\langle KG, Sign, Valid\rangle$**:** properly signed messages validate correctly, i.e., for every message m and random keys $\langle s, v\rangle x \in KG(1^\kappa)$ holds $Valid_v(m, Sign_s(m)) = True$. Furthermore, feasible adversaries cannot generate valid signatures for unsigned messages (with non-negligible probability).

Message Authentication Code (MAC): efficient, shared-key function $MAC : \{0,1\}^\kappa \times D \rightarrow R$. We assume that it is infeasible to forge the MAC for a message m, even if attacker can receive the correct values of the MACs for every other message $m\prime \neq m$. Notice that every pseudorandom function is also a MAC.

For simplicity, for the rest of this chapter we neglect the probability that the adversary can forge the cryptographic functions without knowing the secret key.

7.3 Beaver's Architecture

We present *Beaver* – a robust architecture and method to protect servers from DoS attacks. Beaver employs two DoS-protection mechanisms: one for registration and admission of new client sessions, and another for protecting ongoing sessions. The former uses dedicated *admission servers* (ADMs). The latter is *ϕ-Hopper* – a two-party communication protocol that mitigates DoS attacks by filtering packets based on dynamic, "pseudorandom hopping" fields [30] (see Chapters 5 and 6).

The ADMs can be provided as a common Internet service to multiple legacy servers, and therefore, they are not all trusted. The use of ADMs takes the registration and admission load off the server, so that the server is not concerned with DoS attacks on clients trying to be admitted into the system.

ϕ-Hopper protects client-server communication from DoS, but does not authenticate the communication by itself. The choice of the authentication method to use, if at all, and its implementation, is left entirely to the server. ϕ-Hopper only provides dynamic filtering and rate-limiting facilities. Naturally, ϕ-Hopper can be easily integrated with an authentication component, as done in our IPSec implementation, described in Chapter 6. Together, the ADMs and ϕ-Hopper are very effective against DoS attacks.

7.4 Admission Servers

The ADM has two roles. First, it allows clients to *register* to the service. Second, the ADM performs *the admission process* – authenticating registered clients before authorizing them to communicate with the server. We now detail these two roles.

7.4.1 The Registration Process

A new client that wishes to use the service needs to first *register* to it through an ADM. To be able to register, a client needs to hold a valid *certificate*, which binds the client's public key to the client's identity. The certificate should be signed by an external *Certificate Authority* (CA). The CA is responsible for validating that the client is entitled to the service, possibly by receiving a payment and/or deposit from the client. This certification service can be based on authentication as complex as a biometrics match, or as simple as a credit card number, and is beyond our scope; Beaver just needs to know that it is hard for the same client to obtain many valid identities, or to impersonate another client.

As part of the registration process, the client obtains a unique client ID and shared secrets with the ADM and the server, S_{CADM}, and S_{CS}, respectively. The ADMs do *not* know S_{CS}, as it is encrypted with the client's public key. Different clients have different secrets, and the same client may have different secrets with multiple ADMs.

To register, a client contacts an ADM, and provides it with a certificate. The ADM rate-limits registration requests, and hence may decline the request if it exceeds its quota of registrations at a given time. If the ADM does handle the request, it first validates the certificate and checks whether it is new. If it is invalid, the request is declined. If it is valid but not new, the ADM replays to the client the response it previously sent for that request. Otherwise, the ADM creates S_{CADM} and stores it in its local client database. The certificate includes the client's public key, which the ADM uses to encrypt S_{CADM} before sending it to the client. The ADM then informs the server of the new client's registration.

The server also rate-limits incoming registration requests, and hence may decline it. If the server handles the request, then it also validates the certificate, and if it is valid and new, generates S_{CS} for the client. The server then encrypts S_{CS} with the client's public key, and sends it back the the

ADM, which forwards it to the client.

Both the ADM and the server rate-limit registration requests in order to continue functioning even while under a DoS attack. This makes the registration process a best-effot procedure, but does not pose a problem, since the registration process needs to be performed only once per several years, as long as the client's private key is kept secret.

7.4.2 The Admission Process

The ADM authenticates registered clients before authorizing them to communicate with the server. This is called *the admission process*. There may be multiple admission servers, and all of them are identical, except for a unique secret, S_{SADM} (of a specific ADM), each of them shares with the server. The use of many admission servers protects the admission process from DoS attacks, as the client can initiate the admission process with an arbitrary ADM. A DoS-attacker that wishes to severly harm the admission process needs to launch a massive attack that targets most, if not all of the ADMs. This idea is very similar to the one used for SOS SOAPs [27, 52], and it can be employed here since replicating an ADM is cheap and easy, as opposed to, say, replicating the server.

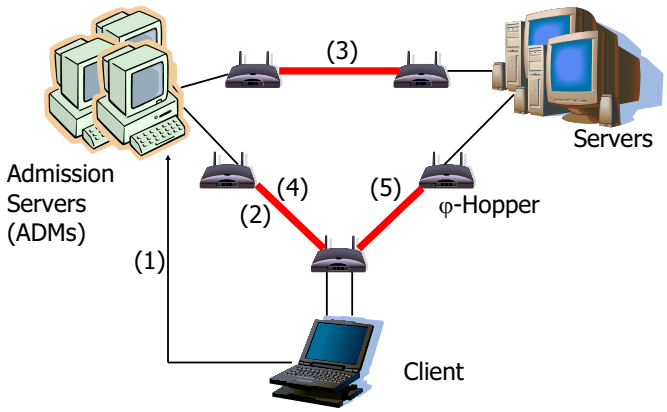


Figure 7.1: Beaver's admission process, where ϕ-Hopper operates in tunnel mode (marked in bold lines).

Figure 7.1 illustrates Beaver's architecture, and shows the admission process in action: (1) A pre-registered client requests an ADM to start a new session with the server. The client can choose the ADM arbitrarily. Specifically, a client that fails to start a session through some ADM may choose a different ADM for the admission process. (2) The ADM communicates with the client via ϕ-Hopper and authenticates the client. Communication via ϕ-Hopper is marked in bold lines. The

figure illustrates ϕ-Hopper in tunnel mode, i.e., hopping between gateways. (3) The ADM contacts the server through a constant ϕ-Hopper session that they share, and asks it to start a new session with the client. The server then opens FIs for the new session with the client. (4) The ADM notifies the client that it can start communicating with the server. (5) The client communicates with the server via ϕ-Hopper. More generally, there can be multiple servers (e.g., a server farm), and an ADM can direct the client to any one of them.

Figure 7.2 provides pseudocode for the admission process. Each message sent in the admission process contains fields of 3 categories: 1) meta-data (source/destination/message type) – omitted in the pseudocode; 2) data fields; and 3) MACs. In Figure 7.2, MAC refers to the MAC field of the appropriate message. If also followed by parentheses, MAC means calculating the MAC field by running the MAC function on the input given in parentheses. For brevity, $newmsg \leftarrow \langle data \leftarrow msg.data \rangle$ means copying all the data fields of message msg to fields with the same name in the new message $newmsg$.

The specific stages and messages used in the admission process are (see Figure 7.2):

1. *Connection request*. The client sends the ADM a connection request containing the client's ID, the current local timestamp, and a random κ-bit number, *requestID*, used along with the client's ID to uniquely identify this admission process. κ is a security parameter, e.g., 128. If no challenge is received within some timeout period $< \mathcal{E}$, the client terminates the admission process. The client may restart the admission process to start a session in spite of transient failures.

2. *Challenge*. If the connection request is valid and its timestamp is more recent than the last saved timestamp for that client, the ADM saves the new timestamp and request ID for that client. Then, the ADM sends the client a challenge comprised of a random nonce. If no response is received within *responseTimeout*$< \mathcal{E}$ seconds, the ADM effectively terminates the admission process, which must be restarted for that client to be admitted into the system.

   The challenge and timeout are used to prevent an adversary from launching a replay attack after dropping the client's messages. Without this mechanism, it would have been possible for the adversary to accumulate dropped client connection requests over a long period of time (even hours), and then replay messages from many clients at once, which would all be deemed valid by the ADM, and cause the server to start many new client sessions. Note that we do not assume that the client and ADM's clocks are synchronized with each other, hence, the ADM cannot check the freshness of connection requests.

3. *Response*. The client proves it holds S_{CADM} by responding with a MAC on the challenge sent by the ADM.

4. *Admission request*. If the response is valid, the ADM trusts the authenticity of the client and

CLIENT

open:
clientTS← local time
requestID← random κ-bit number
connectionRequest← ⟨ *data* ← { *clientID, requestID, clientTS* }, $MAC_{S_{CADM}}$*(data)* ⟩
send *connectionRequest* to ADM
if no valid challenge received within timeout **then**
    invalidate *requestID*
    return connection failure

Upon receiving *challenge* from ADM:
if *challenge.clientID = clientID* **and** *challenge.requestID* is valid **and**
$challenge.MAC_{S_{CADM}} = MAC_{S_{CADM}}$*(challenge.data)* **then**
    response ← ⟨ *data* ← *challenge.*{*clientID, requestID, clientTS, nonce*}, $MAC_{S_{CADM}}$*(data)*, $MAC_{S_{CS}}$*(data)* ⟩
    send *response* to ADM
    if no valid admission completion received within timeout **then**
        invalidate *requestID*
        return connection failure

Upon receiving *admissionCompletion* from ADM:
if *admissionApproval.clientID = clientID* **and** *admissionCompletion.requestID* is valid **and**
$admissionCompletion.MAC_{S_{CS}} = MAC_{S_{CS}}$*(admissionCompletion.data)* **then**
    seed← *admissionApproval.clientID* || *admissionApproval.requestID* || *admissionApproval.clientTS*
    initHopperSession(*seed*, S_{CS}, *admissionCompletion.serverID*)

SERVER

init($ADMs$**):**
for each ADM **in** $ADMs$ **do**
    initHopperSession(0, $ADM.S_{SADM}$, *ADM.ADMID*)

Upon receiving *admissionRequest* from ADM for client $A \leftarrow$ *admissionRequest.clientID*:
if A is authorized to connect through ADM **and** no session with A is pending or in progress **and**
(*admReqTS[A]* is uninitialized **or** *admissionRequest.clientTS* $>$ *admReqTS[A]*) **and**
$admissionRequest.MAC_{S_{SADM}} = MAC_{S_{SADM}}$*(admissionRequest.data)* **and**
$admissionRequest.MAC_{S_{CS}} = MAC_{S_{CS}}$*(admissionRequest.data)* **then**
    admReqTS[A] ← *admissionRequest.clientTS*
    seed ← *admissionRequest.clientID* || *admissionRequest.requestID* || *admissionRequest.clientTS*
    initHopperSession(*seed*, S_{CS}, *serverID*)
    admissionApproval ← ⟨ *data* ← { *admissionRequest.data, serverID* }, $MAC_{S_{SADM}}$*(data)*, $MAC_{S_{CS}}$*(data)* ⟩
    hopperSend(*admissionApproval, ADM*)
    if no session with A begins within *sessionInitTimeout* seconds **then**
        endHopperSession(A)

Figure 7.2: Pseudocode for the admission process (continued on next page).

ADMISSION SERVER

init(serverID**):**
initHopperSession(0, S_{SADM}, *serverID*)

Upon receiving *connectionRequest* from client $A \leftarrow$ *connectionRequest.clientID*:
if (*connReqTS[A]* is uninitialized **or** *connectionRequest.clientTS* $>$ *connReqTS[A]*) **and**
connectionRequest.MAC$_{S_{CADM}}$ = *MAC*$_{S_{CADM}}$*(connectionRequest.data)* **then**
connReqTS[A]$\leftarrow$ *connectionRequest.clientTS*
connReqID[A]$\leftarrow$ *connectionRequest.requestID*
nonce$\leftarrow$ random κ-bit number
connReqNonce[A]$\leftarrow$ *nonce*
challenge$\leftarrow \langle$ *data* $\leftarrow \{$ *connectionRequest.*$\{$*clientID, requestID, clientTS*$\}$, *nonce* $\}$, *MAC* $_{S_{CADM}}$*(data)* $\rangle$
send *challenge* to A
if no valid response received within *responseTimeout* seconds **then**
connReqNonce[A] $\leftarrow$ null

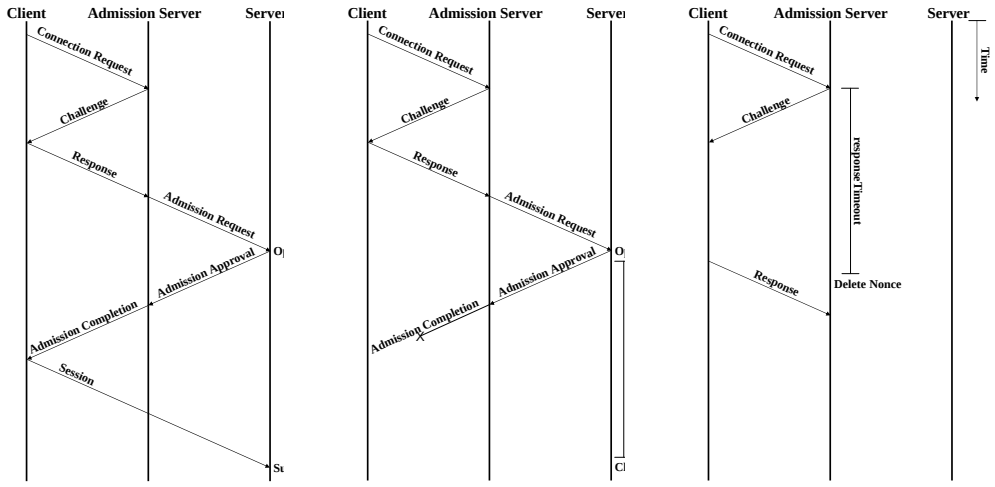
Upon receiving *response* from client $A \leftarrow$*response.clientID*:
if *response.clientTS = connReqTS[A]* **and** *response.requestID = connReqID[A]* **and**
connReqNonce[A] != null **and** *response.nonce = connReqNonce[A]* **and**
response.MAC$_{S_{CADM}}$ = *MAC*$_{S_{CADM}}$*(response.data)* **then**
admissionRequest $\leftarrow \langle$ *data* $\leftarrow$ *response.*$\{$*clientID, requestID, clientTS, nonce*$\}$, *response.MAC* $_{S_{CS}}$, *MAC*$_{S_{SADM}}$*(data)* $\rangle$
hopperSend(*admissionRequest, server*)

Upon receiving *admissionApproval* from server for client $A \leftarrow$ *admissionApproval.clientID*:
if *admissionApproval.requestID = connReqID[A]* **and**
admissionApproval.MAC$_{S_{SADM}}$ = *MAC*$_{S_{SADM}}$*(admissionApproval.data)* **then**
admissionCompletion $\leftarrow \langle$ *data* $\leftarrow$ *admissionApproval.data, MAC* $_{S_{CADM}}$*(data)* $\rangle$
send *admissionCompletion* to A

Figure 7.2 (continued). Pseudocode for the admission process.

sends an admission request with the client's ID to the server.

5. *Admission approval*. If the server does not currently have resources allocated for a session with that client, and the client's request is fresh, the server is willing to start a session with the client. The server then sends back to the ADM a message approving the client's admittance, and allocates Hopper resources for communicating with that client. If the client does not communicate with the server within *sessionInitTimeout* seconds from this stage, these resources are freed. The timeout is used to free resources allocated by a compromised ADM that delays the transmission of admission requests for valid clients, and then sends these requests once the clients no longer try to communicate with the server. In that sense, *sessionInitTimeout* is much shorter than the timeout for session expiration, which is used after the client communicates with the server.
6. *Admission completion*. The ADM sends a message to the client indicating that communication with the server can take place.
7. *Session*. Upon receiving an admission completion message, the client starts a communication session with the server.



(a) Correct execution. (b) Message loss. (c) Delayed client.

Figure 7.3: Admission process.

Figure 7.3(a) shows the messages passed during the admission process if all procedures succeed. Figure 7.3(b) shows a case where the admission completion message is lost, and so the client never

knows that it can connect to the server. After *sessionInitTimeout* seconds expire, the server releases the resources allocated for the session.

Figure 7.3(c) shows a case where the client delays its response to the ADM's challenge, perhaps due to some unexpected multitasking processing. The ADM maintains the nonce used in the challenge for *responseTimeout* time, but if that time passes and no response is received by the ADM, the ADM invalidates the nonce and effectively terminates the admission process. When the client responds later, its message is silently discarded by the ADM.

7.5 Security Analysis

We now analyze Beaver's robustness against different attacks. We consider the goals presented in Section 7.1, and show that the adversary, although able to utilize many methods, cannot prevent Beaver from achieving these goals. We start by giving some definitions (Section 7.5.1). We then study the load induced by authorized communication sessions in Beaver, (Section 7.5.2), and proceed to discuss DoS attacks in detail, (Section 7.5.3). The formal proofs can be found in Section 7.7.

7.5.1 Definitions

Definition 2 (Client validity) *A client is* valid *if it possesses valid registration information. Otherwise, the client is* invalid.

Definition 3 (Message validity) *A message is* valid *if it can be successfully authenticated by the receiving party. Otherwise, the message is* invalid.

Definition 4 (Session establishment) *A session is* established *if the server has resources allocated for that session, and has received at least one valid message from the corresponding client. Otherwise, the session is* not established. *When the server receives the client's first valid message for that server, the client* establishes the session.

Definition 5 (Session validity) *A session is* valid *if it is established or already has resources allocated for it at the server and can be established. Otherwise, the session is* invalid.

7.5.2 Sessions Load on Server

The following lemma and corollary show that the server does not initiate invalid ϕ-Hopper sessions.

Lemma 15 *An invalid client that requests admission from a correct ADM can never successfully pass step* 3 *(response) of the admission process.*

Corollary 6 *If no ADM is malicious, then no server ever allocates resources for communicating with invalid clients.*

Recall that ϕ-Hopper requires only a small maximum number of open FIs per session, typically $n_\phi = 6$. We use this to bound the number of FIs used by invalid sessions.

Lemma 16 *Let n_{ADM} be the number of ADMs in Beaver, of which k ADMs are compromised. Let λ be the anticipated maximum rate of incoming connection requests, let* sessionInitTimeout$= \tau$*, and let n_ϕ be the maximum number of FIs each party opens in a single session. At any given time there are no more than $\frac{\lambda k \tau n_\phi}{n_{ADM}}$ open FIs for invalid sessions.*

sessionInitTimeout is at least 3Δ, as only 3 transmissions after opening the FIs can the server receive a correct client's first message. In that case, we get:

Corollary 7 *Let p be the fraction of compromised ADMs in Beaver. Then for* sessionInitTimeout$=$ 3Δ*, at any given time there are no more than $3\lambda p \Delta n_\phi$ open FIs for invalid sessions.*

7.5.3 Resilience to DoS Attacks

We now quantify the adversary's maximum probability to cause admission DoS as a function of its ability to disrupt communication and cause message loss. Such message dropping can be caused by network-level DoS attacks, whereby the adversary floods the network with traffic.We denote by L_{ADM} the probability of dropping messages from client A to the ADM, and by L_C the probability of dropping a message destined to A.

Lemma 17 (Admission DoS-resistance) *Let $\mathcal{E} > 2\Delta$, and let A be some valid and correct client. If A is about to execute an admission process with a correct ADM exactly once, then an adversary that does not possess A's registration information cannot prevent A from establishing a session with the server with probability better than $1 - (1 - L_{ADM})^2(1 - L_C)$.*

Lemma 17 gives an upper bound on admission failures. This upper bound is depicted in Figure 7.4, for various values of L_{ADM} and L_C. We observe that the failure probability is proportional to the loss rates. For example, when both loss rates are 10%, the failure probability is bounded by roughly 15%.

We next show that Beaver achieves session DoS-resistance.

Lemma 18 (Session DoS-resistance I) *Let $\ell > 1$ be the number of bits representing a FI, and assume the adversary knows the identity of at least one client who has an established session with the server. If the adversary sends the server C invalid messages per second, then on average the server's load will increase by at most $\frac{n_\phi C}{2^\ell}$ messages per second.*

Lemma 19 (Session DoS-resistance II) *If no ADM is malicious, a compromised valid client that does not impersonate other valid clients cannot load the server with more messages per second than the server rate-limits each session.*

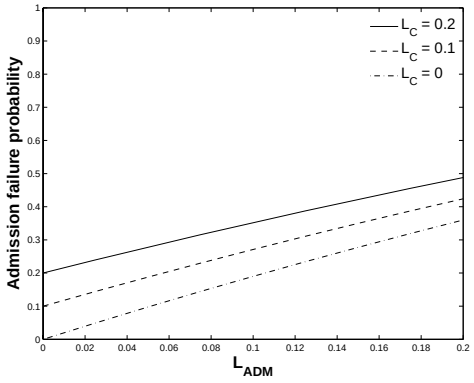


Figure 7.4: Admission failure probability as a function of the message loss probability.

7.6 Related Work

The use of multiple ADMs resembles the use of overlay (proxy) networks in SOS [27], Mayday [1], and other work [52, 54]. However, these systems also screen DoS attacks by hiding the server's identity and making it known only to a few nodes in the overlay network. Thus, in these solutions, all client messages, including those for ongoing sessions, are routed through the overlay, causing the latency of the client-server communication to increase by a factor of 5 or even 10 [27]. Additionally, this is a form of security-by-obscurity. Once the filtering criteria are revealed, spoofed packets that match the server's filtering criteria can penetrate the system's defenses and reach the server. Another drawback of Mayday and SOS is that overlay networks are more complex to set up and update.

In contrast, Beaver only uses the ADMs to authenticate new connections, and does not need the use of an overlay network. It does not hide the server's identity, and enables clients to communicate with the server directly, once their admission request is approved. On the other hand, SOS and Mayday protect the server and its gateway from network-level and application-level DoS attacks, whereas we concentrate solely on application-level DoS mitigation, assuming that some method of protecting the network from DoS attacks is already in place. Our motivation stems from the fact that, as we show in Chapter 6, it is easy to launch an application-level DoS attack that renders the server useless, but does not congest the network.

7.7 Security Analysis: Proofs

Lemma 15. *An invalid client that requests admission from a correct ADM can never successfully pass step* 3 *(response) of the admission process.*

Proof: Let A be an invalid client. Specifically, A does not possess S_{CADM}. Suppose A has managed to pass step 1 (connection request) of the admission process, masquerading as a valid

client A^*. In step 2 (challenge) the ADM provides A with a fresh nonce, and requires A in step 3 (response) to use S_{CADM} to compute the MAC of a message containing that nonce, the client's ID, the request ID, and the client's fresh timestamp. A cannot compute the MAC directly, because it does not hold S_{CADM}. A cannot even watch a legitimate client's traffic to gather many pairs of nonces and their corresponding MACs, as the content being authenticated is unique – the ADM makes sure the client's timestamp was not previously received.

Since A cannot generate a proper response of its own, it must perform the following actions: (1) wait for A^* to reach step 3 and send a valid response; (2) intercept the response (read it while making sure the server does not receive it); and (3) masquerade as A^* and replay the response to the server at some later time. Even if A can perform these actions, it takes at least $\mathcal{E}$ seconds from the time the legitimate client sends a response until the ADM receives A's replayed response. However, by that time, *responseTimeout*$< \mathcal{E}$ seconds have passed from step 2 (challenge), and the ADM has already terminated the admission process by invalidating the nonce. Therefore, A cannot successfully pass step 3 (response) of the admission process. □

Corollary 6. *If no ADM is malicious, then no server ever allocates resources for communicating with invalid clients.*

Proof: A server allocates resources for communicating with a client only after receiving an admission request for that client from an ADM. The admission request is sent in step 4 of the admission process. Since the client does not possess S_{SADM}, it cannot impersonate an ADM and communicate with the server, since its fabricated message will not pass the server's validity checks. Since no ADM is malicious, from Lemma 15 we get that an invalid client does not pass step 3 (response) of the admission process. Therefore, an admission request for that client is never sent, and the server never allocates resources for communicating with that client. □

Lemma 16. *Let n_{ADM} be the number of ADMs in Beaver, of which k ADMs are compromised. Let λ be the anticipated maximum rate of incoming connection requests, let* sessionInitTimeout$= \tau$*, and let n_ϕ be the maximum number of FIs each party opens in a single session. At any given time there are no more than $\frac{\lambda k \tau n_\phi}{n_{ADM}}$ open FIs for invalid sessions.*

Proof: FIs for sessions that are not yet established are opened only in step 5 of the admission process (admission approval). If the ADM reaching step 5 is correct, then from Lemma 15 we get that the client is valid. Valid clients create valid sessions, so we can disregard them. We thus consider only compromised ADMs. A compromised ADM does not need any client interaction to reach step 5, so we can disregard all clients.

The server allows each compromised ADM to send admission requests at a maximum rate of $\frac{\lambda}{n_{ADM}}$ requests per second. Each such request opens a single FI at the server, and more FIs are open as needed, up to n_ϕ FIs. Since the session affiliated with these FIs can never be established, the FIs

are closed τ seconds after they are opened. That is, each compromised ADM is the cause of at most $\frac{\lambda \tau n_\phi}{n_{ADM}}$ open FIs at the server. Since there are k compromised ADMs, the maximum number of open FIs waiting for invalid sessions is $\frac{\lambda k \tau n_\phi}{n_{ADM}}$. □

Lemma 17. (Admission DoS-resistance) *Let $\mathcal{E} > 2\Delta$, and let A be some valid and correct client. If A is about to execute an admission process with a correct ADM exactly once, then an adversary that does not possess A's registration information, with probability L_{ADM} of dropping messages from A to the ADM and probability L_C of dropping a message destined to A, cannot prevent A from establishing a session with the server with probability better than* $1 - (1 - L_{ADM})^2(1 - L_C)$.

Proof: In the admission process, once A's response is validated by the ADM (when step 3 completes), there is nothing to stop A from establishing a session with the server – the communication of the ADM or the client with the server cannot be disrupted with no DoS attacks thanks to ϕ-Hopper, and A is going to try and establish a session with the server regardless of whether or not the admission completion message from the server has been received or not.

To prevent the client from establishing a session with the server, the adversary must sabotage the admission process before step 3 is over. The probability of A successfully completing step 3 of the admission process, when considering only message loss induced by the adversary, is $(1 - L_{ADM})^2(1 - L_C)$, as the connection request and response must be received by the ADM, and the challenge must be received by A. We are left to show that the adversary cannot use other methods to interfere with steps 1 through 3 of the admission process.

Let us first examine step 1, the connection request. Each new valid connection request terminates any pending admission processes and starts a new admission process. A connection request is considered valid if it can be authenticated, and if the timestamp on the request is more recent than the last timestamp received on a valid connection request from A. The adversary may try to send a connection request with a new timestamp to tear down A's current admission request, or to cause the ADM to discard future connection requests due to an "old" timestamp. However, the adversary does not possess A's registration information. Specifically, it does not posses S_{CADM}, and thus cannot fabricate a connection request message that passes the ADM's authentication.

Other than message loss, the only way the adversary can harm step 2 is by sending A a wrong nonce that will be considered instead of the correct nonce sent from the ADM. However, in order for A to accept the nonce, the challenge message must carry the same $requestID$ randomly chosen by A and sent in the connection request message. Since $\mathcal{E} > 2\Delta$, the adversary cannot eavesdrop to the connection request, see the value of $requestID$, and send a fake challenge with the appropriate $requestID$ to A before A receives the correct challenge from the ADM. Thus, the adversary needs to guess the value of $requestID$, and the probability that this guess succeeds is negligible.

Finally, the adversary cannot disrupt step 3, as its message will again not pass the ADM's

authentication procedure. the ADM will continue waiting for A's correct answer, regardless of the number of invalid answers sent to the ADM by the adversary, supposedly on behalf of A, until *responseTimeout* seconds expire. By that time, A will surely send the correct response.

We get that, other than inducing message loss, the adversary cannot disrupt steps 1 through 3 with non-negligible probability, and so A is able to establish a session with the server with probability $1-(1-L_{ADM})^2(1-L_C)$. □

Lemma 18. (Session DoS-resistance I) *Let $\ell > 1$ be the number of bits representing a FI, and assume the adversary knows the identity of at least one client who has an established session with the server. If the adversary sends the server C invalid messages per second, then on average the server's load will increase by at most $\frac{n_\phi C}{2^\ell}$ messages per second.*

Proof: All communication with the server is via ϕ-Hopper. ϕ-Hopper filters messages at the gateway according to a matching between client IDs and FIs open for those clients. To pass the filter, a message must contain a client ID for a client that has an established session with the server. The message must also contain a FI that is open for that client. This matching between client IDs and FIs means that the number of active clients the adversary knows is irrelevant, as long as the adversary knows at least one such client.

For each session the server maintains at most n_ϕ open FIs. In total, there are 2^ℓ potential FIs, and the FIs to open are chosen by hashing different values. Since communication is via ϕ-Hopper, the adversary needs to guess FI numbers when sending invalid messages. By our hash-functions assumption, the attacker cannot distinguish the open FIs from FIs chosen uniformly at random. The probability that a single invalid message hits an open FI is $\frac{n_\phi}{2^\ell}$ Since the adversary sends C independent messages, the expected number of messages that will hit an open FI and make its way to the server is $\frac{n_\phi C}{2^\ell}$, and this is the maximum average increase in server load. □

Lemma 19. (Session DoS-resistance II) *If no ADM is malicious, a compromised valid client that does not impersonate other valid clients cannot load the server with more messages per second than the server rate-limits each session.*

Proof: The server's load increases by messages that are not filtered out by ϕ-Hopper, i.e., by messages that potentially belong to an active session. Clearly, a compromised valid client can establish a session and send messages to the server. The server is then loaded in accordance with the rate by which each session is limited. We are left to show that the client cannot load the server with any other message.

As mentioned in Section 7.4.2, the server allows a client to have only one valid session at a time. Therefore, the client cannot load the server more by creating another session, and is limited to having a single session. Every message the client sends to the server is rate-limited according to this session, and so if the client wishes to load the server by performing a DoS attack, it must

use other client IDs. However, since no ADM is malicious, we get from Corollary 6 that only valid clients can load the server. Since the compromised client does not impersonate other valid clients, it cannot load the server with more messages than it does with its single established session. □

Chapter 8

Discussion, Results and Conclusions

We presented 3 novel systems and protocols that deal with DoS attacks:

1. Drum – a DoS-resistant Gossip-based multicast protocol that maintains typical propagation times even when under a DoS attack. Adaptive Drum is an extension of this protocol, that also locally adapts node behavior according to their local perceived state of the attack, and thus achieves better propagation times under attack.

2. ϕ-Hopper – a two-party communication protocol that uses pseudorandom hopping of header field values in packets in order to provide DoS resistance. We have also shown an extension of this protocol that supports client-server communication.

3. Beaver – a method and architecture to protect legacy servers from DoS attacks.

Drum and Adaptive Drum

We have conducted the first systematic study of the impact of DoS attacks on multicast protocols, using asymptotic analysis, simulations, and measurements. Our study has exposed weaknesses of traditional gossip-based multicast protocols: although such protocols are very robust in the face of process crashes, we have shown that they can be extremely vulnerable to DoS attacks. In particular, an attacker with limited attack strength can cause severe performance degradation by focusing on a small subset of the processes.

We have suggested a few simple measures that one can take in order to improve a system's resilience to DoS attacks: (i) combining pull and push operations; (ii) bounding resources separately for each operation; and (iii) random port selection for each communication channel. We have presented Drum, a simple gossip-based multicast protocol that uses these measures in order to eliminate vulnerabilities to DoS attacks. Our closed-form mathematical analysis, simulations, and

empirical tests have proven that these measures go a long way in fortifying a system against DoS attacks. We have shown that, as the attack strength increases asymptotically, the most effective attack against Drum is one that divides the attack power among all the correct processes in the system. As expected, the inevitable performance degradation due to such a broad attack is identical for all the studied protocols. However, protocols that use only pull or only push operations perform much worse under more focused attacks, which have little influence on Drum.

We expect our proposed methods for mitigating the effect of DoS attacks to be applicable to various other systems operating in different contexts. Specifically, the use of well-known ports should be minimized, and each process should be able to choose some of its communication partners by itself. Our analysis process and its corresponding metric can be used to generally quantify the effect of DoS attacks. We hope that other researchers will be able to apply similar techniques in order to quantitatively analyze their system's resilience to DoS attacks.

We presented a novel approach to dealing with DoS attacks – adapting the protocol's behavior according to the perceived attack. Adaptation is done locally at each node, but a global improvement is achieved. The adaptation is based on a set of constraints that compose an optimization problem, which is solved using linear programming. Our simulations showed that in our case study adaptation increases performance by up to 34%. We believe that our work is the first step in designing adaptive protocols that deal with DoS attack better than static protocols.

ϕ-Hopper

We have presented a model for port-based rationing channels, and a protocol robust to DoS attacks, for communication over such channels. Our protocol is simple and efficient, and hence can sustain high loads of traffic, as happens, e.g., in high-speed networks. At the same time, our analysis shows that the protocol is highly effective in mitigating the effects of DoS attacks. Our formal framework and suggested protocol apply not only to port-based filtering, but to a much broader category of filtering based on any packet identifier. Thus, our work constitutes the first step in evaluating existing filtering and rate-limiting mechanisms.

As the important field of application-level DoS mitigation is relatively new, there is much research space to explore. While our worst case analysis is valuable, it can be followed by simulations, experiments, and common case analysis. Moreover, the system aspects of deploying such a protocol in today's Internet are yet to be dealt with. We now describe several exemplary future research directions.

Our model is realistic, as it only requires the underlying channel to provide port-based filtering; therefore, it can be efficiently implemented using existing mechanisms, typically at a gateway firewall or router. This raises an interesting question regarding the trade-off between the cost and the possible added value of implementing additional functionality by the channel (e.g., at the firewall).

We hope that future work will take further strides towards defining realistic yet tractable models of the channel and the adversary that will aid in answering this question.

This work has focused on two parties only. It would be interesting to extend it to multiparty scenarios, such as client-sever and multicast. These scenarios may require a somewhat different approach, and will obviously necessitate analyses of their own. Furthermore, we required the parties to share a secret key; we believe we can extend the solution to establish this key using additional parties, e.g., a key distribution center, or using 'proof of work' [17].

Our work has focused on resisting DoS attacks; however, it could impact the performance and reliability properties of the connection; in fact, it is interesting to explore combinations between our model and problem, and the classical problems of reliable communication over unreliable channels and networks. Furthermore, since our work requires a shared secret key, it may be desirable to merge it with protocols using shared secret keys for confidentiality and authentication, such as SSL/TLS and IPSec.

Beaver

We presented Beaver, a method and architecture to protect applications from DoS attacks. Beaver uses the following ideas to provide strong protection against DoS attacks:

- A best-effort registration process that distributes shared secrets (keys). Only pre-registered clients can start sessions with the server, and it is hard to fake many identities.
- An admission process that authorizes clients to communicate with the server. The server does not allocate resources for a client that was not authorized. The admission servers are a seperate entity and so provide seperation of "war zones" – attacking the admission servers does not harm ongoing client-server sessions. Additionally, having redundant admission servers makes it hard for the attacker to easily harm the admission process.
- Filtering based on a pseudorandom number that is hard to guess, and changing the pseudorandom number periodically ("hopping"), so that even if a filter is revealed, it becomes irrelevant before the attacker has the opportunity to load the server with bogus requests.
- Rate-limiting each authorized client to make sure compromised or selfish clients cannot consume much of the server's resources, at the expense of other clients.

We formally proved Beaver's good properties in withstanding DoS attacks. The measurements we presented in this work show that indeed Beaver is a promising solution.

Summary

Our results show that it is not enough to protect just the network layer from DoS attacks, but the application layer should also be protected. Additionally, we show that using authentication alone to mitigate the effects of DoS attacks is insufficient, and may effectively shift the DoS problem from the prospective target to the authenticator.

In contrast, our robust systems leverage existing, cheap components such as packet filters and rate-limiters to perform efficient DoS-mitigation. We have analyzed our systems and proved their good properties in facing DoS attacks. In addition to the analytical framework we have devised, we also implemented and tested our systems in real conditions, and provided measurements that support our analysis and show that our systems continue to function properly, or gracefully degrade in the face of massive DoS attacks.

References

[1] D. G. Andersen. Mayday: Distributed filtering for Internet services. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[2] K. Argyraki and D. R. Cheriton. Active Internet traffic filtering: Real-time response to denial-of-service attacks. In *Proceedings of the USENIX Annual Technical Conference*, April 2005.

[3] R. Atkinson. Security architecture for the Internet Protocol. RFC 2401, IETF, 1998.

[4] G. Badishi, A. Herzberg, and I. Keidar. Keeping denial-of-service attackers in the dark. In *DISC*, volume 3724 of *LNCS*, pages 18–32, September 2005.

[5] G. Badishi, A. Herzberg, and I. Keidar. Keeping denial-of-service attackers in the dark. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, July–September 2007.

[6] G. Badishi, I. Keidar, and A. Sasson. Exposing and eliminating vulnerabilities to denial of service attacks in secure gossip-based multicast. In *The International Conference on Dependable Systems and Networks (DSN)*, pages 223–232, June–July 2004.

[7] G. Badishi, I. Keidar, and A. Sasson. Exposing and eliminating vulnerabilities to denial of service attacks in secure gossip-based multicast. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 3(1):45–61, March 2006.

[8] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.

[9] R. K. C. Chang. Defending against flooding-based distributed denial-of-service attacks: A tutorial. *IEEE Communications Magazine*, 40:42–51, October 2002.

[10] Cisco Systems. Defining strategies to protect against TCP SYN denial of service attacks.

[11] M. Collins and M. K. Reiter. An empirical analysis of target-resident DoS filters. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 103–114, May 2004.

[12] CSI/FBI. Computer crime and security survey, 2003.

[13] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Stuygis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *6th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–12, 1987.

[14] P. T. Eugster, R. Guerraoui, S. B. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)*, 21(4):341–374, 2003.

[15] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *ACM/IEEE Transactions on Networking*, 1(4):397–413, August 1993.

[16] X. Geng and A. B. Whinston. Defeating distributed denial of service attacks. *IEEE IT Professional*, pages 46–51, July/August 2000.

[17] V. D. Gligor. Guaranteeing access in spite of service-flooding attacks. In *the Security Protocols Workshop*, 2003.

[18] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the Association for Computing Machinery*, 33(4):792–807, 1986.

[19] I. Gupta, A.-M. Kermarrec, and A. J. Ganesh. Efficient epidemic-style protocols for reliable and scalable multicast. In *21st IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 180–189, October 2002.

[20] I. Gupta, R. van Renesse, and K. P. Birman. Scalable fault-tolerant aggregation in large process groups. In *The International Conference on Dependable Systems and Networks (DSN)*, pages 433–442, 2001.

[21] N. M. Haller. The S/KEY one-time password system. In *the ISOC Symposium on Network and Distributed System Security*, February 1994.

[22] C. Jin, H. Wang, and K. G. Shin. Hop-count filtering: an effective defense against spoofed DDoS traffic. In V. Atluri and P. Liu, editors, *Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS-03)*, pages 30–41, New York, Oct. 27–30 2003. ACM Press.

[23] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. In *Proceedings of the International World Wide Web Conference*, pages 252–262. IEEE, May 2002.

[24] Juniper Networks. The need for pervasive application-level attack protection.

[25] R. M. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized rumor spreading. In *IEEE Symposium on Foundations of Computer Science*, pages 565–574, 2000.

[26] A.-M. Kermarrec, L. Massouli, and A. J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):248–258, March 2003.

[27] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: An architecture for mitigating DDoS attacks. *Journal on Selected Areas in Communications*, 21(1):176–188, 2004.

[28] B. Krishnamurthy and J. Wang. On network-aware clustering of Web clients. In *Proceedings of the SIGCOMM*, Aug. 2000.

[29] P. Kyasanur, R. R. Choudhury, and I. Gupta. Smart gossip: an adaptive gossip-based broadcasting service for sensor networks. In *The IEEE International Conference on Mobile Adhoc and Sensor Systems*, pages 91–100, October 2006.

[30] H. C. J. Lee and V. L. L. Thing. Port hopping for resilient networks. In *the 60th IEEE Vehicular Technology Conference*, September 2004.

[31] M. J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. In *European Dependable Computing Conference (EDCC)*, pages 364–379, 1999.

[32] M. J. Lin, K. Marzullo, and S. Masini. Gossip versus deterministically constrained flooding on small networks. In *14th International Symposium on DIStributed Computing (DISC)*, pages 253–267, 2000.

[33] P. Linga, I. Gupta, and K. Birman. A churn-resistant peer-to-peer web caching system. *ACM Workshop on Survivable and Self-Regenerative Systems*, October 2003.

[34] P. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *Computer Communications Review*, 32(3):62–73, July 2002.

[35] D. Malkhi, Y. Mansour, and M. K. Reiter. Diffusion without false rumors: On propagating updates in a Byzantine environment. *Theoretical Computer Science*, 299(1–3):289–306, April 2003.

[36] D. Malkhi, E. Pavlov, and Y. Sella. Optimal unconditional information diffusion. In *15th International Symposium on DIStributed Computing (DISC)*, 2001.

[37] D. Malkhi, M. K. Reiter, O. Rodeh, and Y. Sella. Efficient update diffusion in Byzantine environments. In *20th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, October 2001.

[38] Y. M. Minsky and F. B. Schneider. Tolerating malicious gossip. *Distributed Computing*, 16(1):49–68, February 2003.

[39] D. Moore, G. Voelker, and S. Savage. Inferring Internet denial-of-service activity. In *Proceedings of the 10th USENIX Security Symposium*, pages 9–22, August 2001.

[40] W. G. Morein, A. Stavrou, D. L. Cook, A. D. Keromytis, V. Misra, and D. Rubenstein. Using graphic Turing tests to counter automated DDoS attacks against web servers. In *CCS*, pages 8–19, 2003.

[41] NetContinuum. Web application firewall: How NetContinuum stops the 21 classes of web application threats.

[42] P-Cube. DoS protection.

[43] P-Cube. Minimizing the effects of DoS attacks.

[44] B. Pittel. On spreading a rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, February 1987.

[45] Riverhead Networks. Defeating DDoS attacks.

[46] Riverhead Networks. Products overview.

[47] L. Rodrigues, S. Handurukande, J. P. U. do Minho, R. Guerraoui, and A.-M. Kermarrec. Adaptive gossip-based broadcast. In *The International Conference on Dependable Systems and Networks (DSN)*, pages 47–56, 2003.

[48] C. L. Schuba, I. V. Krsul, M. G. Kuhn, E. H. Spafford, A. Sundaram, and D. Zamboni. Analysis of a denial of service attack on TCP. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 208–223, May 1997.

[49] S. M. Schwartz. Frequency hopping spread spectrum (fhss).

[50] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round-robin. *ACM Transactions on Networking*, 4(3):375–385, 1996.

[51] S. Staniford, V. Paxson, and N. Weaver. How to own the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, August 2002.

[52] A. Stavrou and A. D. Keromytis. Countering DoS attacks with stateless multipath overlays. In *CCS*, November 2005.

[53] Stephen de Vries. Application Denial of Service Attacks.

[54] J. Wang, X. Liu, and A. A. Chien. Empirical study of tolerating denial-of-service attacks with a proxy network. In *Usenix Security*, 2005.

[55] J. Wang, L. Lu, and A. A. Chien. Tolerating denial-of-service attacks using overlay networks – impact of overlay network topology. *ACM Workshop on Survivable and Self-Regenerative Systems*, October 2003.

[56] P. Watson. Slipping in the window: TCP reset attacks. In *CanSecWest*, 2004.

[57] Y. Weinsberg, T. Anker, D. Dolev, and S. Kirkpatrick. On a NIC's operating system, schedulers and high-performance networking applications. In *HPCC*, September 2006.

[58] E. W. Weisstein. *CRC Concise Encyclopedia of Mathematics*.

[59] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[60] A. Yaar, A. Perrig, and D. Song. Pi: A path identification mechanism to defend against DDoS attacks. In *IEEE Symposium on Security and Privacy*, May 2003.

[61] A. Yaar, A. Perrig, and D. Song. Siff: A stateless internet flow filter to mitigate ddos flooding attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[62] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, 2002.

הגנה על מערכות מפני התקפות מניעת שרות ברמת האפליקציה

גל בדישי

הגנה על מערכות מפני התקפות מניעת שרות ברמת האפליקציה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת תואר
דוקטור לפילוסופיה

גל בדישי

הוגש לסנט הטכניון — מכון טכנולוגי לישראל

חשון תשס"ח חיפה נובמבר 2007

המחקר נעשה בהדרכת פרופ' עדית קידר
בפקולטה להנדסת חשמל

הכרת תודה

ראשית, ברצוני להודות למנחה שלי, פרופ' עדית קידר, שמצאתי בה את כל אשר יכולתי לקוות למצוא במנחה.

תודה גדולה מגיעה גם למשפחתי האוהבת והתומכת, ולחברי ועמיתי, ארן ברגמן, יששכר (זיגי) ולטר, נדב לביא, וצביקה גוז, על התמיכה והעזרה שלהם, ועל החברות הנפלאה שלהם.

אחרונה חביבה, אני מודה למירי שאול, שהעניקה לי השראה לעשות דברים שמעולם לא חשבתי שאעשה.

אני מודה למשרד המדע ולטכניון על התמיכה הכספית הנדיבה בהשתלמותי

תוכן עניינים

רשימת איורים

רשימת טבלאות

תקציר

התקפות מניעת שרות (Denial of Service – DoS) הפכו בשנים האחרונות לבעיה משמעותית ברחבי האינטרנט [12]. בהתקפות אלה, התוקף מנסה למנוע מהמחשב המותקף לספק שרות טוב ואיכותי, ואולי אף מנסה למנוע את השרות בכלל. ישנן מספר דרכים להשיג מטרה זו, ואנו נתרכז בתוקף השולח בקשות מזויפות ברשת למחשב המותקף, ומעמיס עליו עבודה. באופן טיפוסי, התקפת DoS כוללת מספר רב של מחשבים עליהם התוקף השתלט מבעוד מועד. מחשבים אלה מכונים "זומבים", ויעודם היחיד הוא להוציא לפועל התקפת DoS בצורה מתואמת כנגד מטרה כלשהי, ובכך להגדיל את עוצמת ההתקפה שתוקף בודד יכול לבצע [51].

הדבר הראשון אותו יש לבצע כאשר מגנים על מערכת מפני התקפה כנ"ל, הוא להגן על הרשת עצמה מפני עומס של הודעות מזויפות. ברור הוא, שאם התוקף יכול להעמיס את הרשת כך שהודעות ילכו לאיבוד, האפליקציה המותקפת תמיד תסבול, ללא קשר לשיטות בהן היא נוקטת על מנת לצמצם את השפעת ההתקפה. מכאן, שהגנה על הרשת היא הכרחית, ואכן, תחום ההגנה על הרשת זכה לתשומת לב רבה [46, 43].

בד"כ, הגנה מפני התקפות DoS מתבצעת בעזרת שימוש ברכיבים סטנדרטיים כגון סינון הודעות והגבלת קצב. סינון הודעות משווה ערכי שדות מסוימים בכותרת ההודעה (למשל, כתובת השולח) לערכים שנקבעו מראש. אם הערכים זהים, ההודעה ממשיכה ליעדה. אחרת, ההודעה נזרקת. הבעיה היא, שהתוקף יכול ליצור חבילות כרצונו, ובפרט הוא יכול ליצור חבילות שערכי שדות הכותרת שלהם מתאימים לערכים אותם מחפש המסנן. כך, למרות הסינון, התוקף יכול להציף את המותקף. הגבלת קצב ההודעות הנכנסות אכן מגנה על הרשת מפני העמסה, אך אינה מגנה על האפליקציה, שכן אין הבחנה בין ההודעות הנזרקות לשם הגבלת קצב ההגעה. כך, הודעות טובות נזרקות יחדיו עם הודעות מזויפות, והפגיעה באפליקציה עדיין מתרחשת. יתרה מזו - הגנה ברמת הרשת אינה מבטיחה הגנה גם ברמת האפליקציה, שכן אם האפליקציה מבצעת עיבוד משמעותי על כל בקשה שמתקבלת, הרי שניתן לגרום לה למיצוי משאבים גם ע"י שליחת בקשות בקצב שאינו מעמיס את הרשת. דבר זה נכון במיוחד עבור אפליקציות המשתמשות בפרימיטיביים קריפטוגרפיים כדי לאמת או לפענח/להצפין חבילות.

הנחה מקובלת היא שאימות קריפטוגרפי של כל חבילה שעוברת ברשת, לדוגמה, בעזרת IPSec [3], מגן על האפליקציה מפני התקפות DoS, כיוון שחבילות מזויפות לא יכולות לעבור את המחשב המאמת. אמנם, חבילות רעות לא מגיעות למותקף, אך עצם השימוש באימות קריפטוגרפי (פעולה יקרה יחסית) חושף את המאמת להתקפות DoS, בהן המאמת ממצה את משאביו באימות חבילות

מזויפות, והחבילות הטובות נזרקות כיוון שאין משאבים פנויים לשם טיפול בהן. לכן, כפי שאנו מראים בעבודה זו, שימוש ב-IPSec בלבד אינו מספיק על מנת לפתור את בעית מניעת השרות.

מטרתנו היא לבנות מערכות ופרוטוקולים העמידים בפני התקפות DoS, תוך שימוש ברכיבים סטנדרטיים, זולים ופשוטים, כך שניתן יהיה לממש את המערכות בקלות. הנחת העבודה שלנו היא שרשת התקשורת כבר מוגנת בפני התקפות, ואנו נערכים למנוע התקפות ברמת האפליקציה, בהנחה שרשת התקשורת אינה מועמסת. בעבודה זו, אנו מספקים פתרונות עבור שרותי התקשורת הבאים: 1) הפצה מרובת משתמשים (gossip-based multicast [8, 14]); 2) תקשורת בין שני מחשבים; ו- 3) תקשורת שרת-לקוח [27, 1]. הקו המנחה אותנו במערכות אלה הוא שמירה על מספר כללים שמאפשרים הגנה יעילה מפני התקפות מניעת שרות. בכל המערכות המשאבים מוגבלים ומופרדים לפעולות שונות - פעולות שעשויות לעבור את מכסת המשאבים המותרת אינן מתבצעות. בנוסף, התקשורת בין המחשבים מתבצעת בצורה המאפשרת סינון פשוט ובטוח של הודעות התוקף, או בצורה המקנה יתירות רבה, ולכן להודעות התוקף אין השפעה רבה.

אנו מניחים שלתוקף יש הגבלה על הקצב המקסימלי בו הוא יכול לשלוח הודעות. הקצב המקסימלי שהתוקף יכול לשלוט בו הודעות הוא למעשה סכום הקצבים בהם יכולים לשלוח הודעות המחשבים עליהם הוא שולט. התוקף יכול ליצור חבילות כרצונו, ואף לראות חבילות שעוברות ברשת, אולם התוקף אינו יכול להגיב באופן מיידי להודעה אותה הוא ראה. לדוגמה, התוקף אינו יכול מיידית לשנות את התקפתו כתוצאה מצפייה בהודעה, כיוון ששינוי ההתקפה גורר מתן פקודות חדשות למחשבים הנשלטים, והעברת הפקודות לוקחת זמן. כמו כן, התוקף אינו מסוגל לשבור פרימיטיבים קריפטוגרפיים בהסתברות שאינה זניחה.

פרק 3 מציג את Drum. מערכת Drum היא מערכת המשלבת מספר טכניקות על מנת להתגונן בפני התקפות מניעת שרות:

1) שימוש בפורטים אקראיים, במידת האפשר, על מנת לאפשר תקשורת חסינת התקפות. כיוון שאנו מניחים כי הרשת אינה עמוסה, הרי שהתוקף יכול לגרום נזק לנקודות הקצה רק אם הודעותיו נשלחות לפורטים עליהם המחשבים מאזינים. שימוש בפורטים אקראיים מונע מהתוקף לרכז את מאמציו בנקודות התורפה של המחשבים, ומאלץ אותו להתקיף את נקודות הקצה בצורה עיוורת.

2) הפרדת משאבים והגבלתם לכל פעולה. כך, גם אם התוקף מצליח להעמיס פעולה מסוימת, הרי שעומס זה אינו משפיע על שאר הפעולות. בנוסף, עקב הגבלת המשאבים לכל פעולה, העמסה של פעולה מסוימת לא יכולה לגרום לקריסת המחשב.

3) שילוב של דחיפת הודעות ומשיכתן. כל סיבוב, כל צומת בוחר אקראית מספר צמתים אחרים, ושולח אליהם הודעות. מצמתים אחרים שנבחרו באופן אקראי, הוא מבקש הודעות. השילוב הזה של בחירה אקראית של צמתים ושל דחיפת הודעות ומשיכתן, מונע מהתוקף אפשרות למנוע מהצומת לקבל או לשלוח הודעות, גם אם הוא תוקף פורטים ידועים מראש.

פרק 4 מציג את Adaptive Drum. Adaptive Drum הוא שיפור של Drum המבוסס על התאמה של התנהגויות הצמתים למצב ההתקפה על המערכת, כפי שהוא נצפה על ידי כל צומת בנפרד. בכל סיבוב, כל צומת מעריך את מצב ההתקפה על פי המידע המקומי שיש ברשותו. בהתאם להערכה זו, פותר הצומת בעית אופטימיזציה, ומשנה את התנהגותו בהתאם לפתרון הבעיה. הקצאת המשאבים

החדשה של כל צומת נעשית בנפרד, ללא תאום עם צמתים אחרים. למרות זאת, התוצאה הסופית היא זמן התפשטות טוב יותר משל Drum תחת התקפת מניעת שרות.

פרק 5 מציג את ϕ-Hopper. פרוטוקול ϕ-Hopper משתמש אף הוא בשדות מזהים אקראיים על כל הודעה לשם סינון הודעות. בנוסף, יש מנגנון הגבלת קצב עבור הודעות שעוברות את הסינון, כדי שגם אם התוקף בטעות הצליח להשחיל הודעות רעות דרך המסנן, הן לא יעמיסו את מחשב הקצה. זה, כמובן, תקף גם עבור לקוח שמנסה להעמיס שרת, בין אם כתוצאה מהשתלטות זדונית על הלקוח, או כתוצאה מכך שהלקוח מנסה להשיג שרות טוב יותר על חשבון לקוחות אחרים. ערכי השדות שמשמשים לסינון משתנים כל פרק זמן קבוע, כך שגם אם התוקף יכול לראות הודעות ולנסות להשתמש במידע זה על מנת ליצור הודעות שיעברו את המסנן, הרי שעד שהתוקף מרכיב הודעות מתאימות, הערך המסונן הופך להיות לא רלוונטי, והמסנן מצפה לקבל ערך אחר. כך, המערכת מגנה על עצמה בצורה פרואקטיבית מפני התקפות בהן התוקף עשוי לראות את ההודעות העוברות ברשת (אך להגיב עליהן במועד מאוחר יותר).

פרק 7 מציג את Beaver. מערכת Beaver מגנה על שרתים קיימים מפני התקפות מניעת שרות בעזרת תקשורת מבוססת phi-Hopper מתי שניתן, ושימוש בשרתי כניסה ייעודיים. שרתי הכניסה משמשים לשם חלוקת מפתחות ללקוחות, והרשאת לקוחות לחיבור עם השרת. ללא הרשאה של שרת כניסה, אין ללקוח אפשרות לתקשר עם השרת, ולכן תוקף לא יכול להעמיס את השרת. העמידות של שרתי הכניסה מבוססת על מספרם הרב, כיוון שגם מחשב פשוט יכול לשמש כשרת כניסה. שרתי הכניסה ממוקמים באזורים שונים ברחבי האינטרנט, והלקוח יכול לבחור להתחבר לשרת דרך שרת כניסה שהוא בוחר באופן שרירותי. כיוון שכך, תוקף שרוצה למנוע מלקוח חיבור לשרת, צריך לתקוף את כל שרתי הכניסה, דבר המצריך ממנו יכולת ליצר קצב תעבורה אדיר.

אנו מנתחים ניתוח פורמלי כל מערכת שפיתחנו, ומוכיחים את נכונותה ואת עמידותה בפני התקפות. בנוסף, אנו מראים תוצאות מסימולציות וניסויים שהרצנו, ואשר מדגימים את יכולת המערכת לתפקד תחת התקפה, ללא הנחות המודל שיצרנו לשם ניתוח המערכת. התוצאות בעבודה זו מראות בברור שהמערכות שלנו ישימות, וניתנות למימוש בקלות. החידוש שלנו נובע לא רק מבניית המערכות החדשות, אלא אף מעצם העובדה הפשוטה שאנו מנתחים את בעית מניעת השרות ברמת האפליקציה בצורה פורמלית, ממדלים את הסביבה ואת התוקף, ומוכיחים הוכחות על המערכות שאנו מפתחים.