Denial of Service? Leave it to Beaver

Gal Badishi*† Amir Herzberg ‡ Idit Keidar † Oleg Romanov † Avital Yachin †

†{*badishi@ee, idish@ee, oleg@softlab, saty@t2*}*.technion.ac.il, EE Department, Technion*
‡*herzbea@macs.biu.ac.il, CS Department, Bar-Ilan University*

Abstract

We present Beaver, a method and architecture to "build dams" to protect servers from Denial of Service (DoS) attacks. 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 of messages, allowing the use of efficient routing (avoiding overlays), and establishing keys and state as needed. We present two prototype implementations of Beaver, one as part of IPSec in a Linux kernel, and a second as an NDIS hook driver on a Windows machine. Our measurements illustrate that Beaver withstands severe DoS attacks without hampering the client-server communication. Moreover, Beaver is simple and easy to deploy.

1 Introduction

We consider the problem of protecting legacy servers from (Distributed) Denial of Service (DoS) attacks by realistic adversaries. We aim to provide a simple method that effectively mitigates DoS attacks, while maintaining virtually the same latency and load characteristics the legacy system exhibits.

The first step in providing DoS resistance is protecting the network layer. Readily-available 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, and rate-limiting, while good for the network, is of little use to the application, as valid packets are indiscriminately discarded. 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. Thus, network-level protection needs to be complemented by application-level mechanisms.

Therefore, our goal, as described in Section 2, is to provide simple and practical DoS protection for the application layer, using readily-available network-level solutions. We note that in a typical application-level DDoS attack, it is quite likely that many of the computers participating in the attack are fake clients. 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.

The adversaries we consider may adapt their attacks (with some delay) as a result of gaining knowledge of their attacks' successfulness, e.g., by observing service performance degradation, or by eavesdropping on messages or parts thereof. An adversary may control many machines, some of them may even be part of the system. The adversary may also craft arbitrary messages or send many messages to some destination to try to cause DoS by resource exhaustion at the receiver. The model we use to tackle the DDoS problem is further elaborated in Section 3.

In Section 4 we present *Beaver* – a robust architecture and method for application-level DoS protection. Beaver is composed of two main components: (1) ϕ-Hopper – enforces filtering and rate-limiting, as presented in Section 5. (2) The admission servers – responsible for registering clients and authorizing new client-server sessions, as described in Section 6.

Beaver's components can be implemented in several different ways, depending on the required deployment scenario and capabilities. Feasibility analysis and deployment options are presented in Section 7. We describe an implementation of our system in two variations: (1) by modifying a Linux kernel's IPSec [2] implementation, and (2) by inserting code in a Windows NDIS layer.

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

*Supported by the Israeli Ministry of Science.

these results are only applicable for a *model* of the system, which may or may not correctly reflect the real world. In Section 8 we present a formal analysis of Beaver, while in Section 9 we focus on empirical results. Our findings validate earlier theoretical results [3]. Further comparison to earlier results is presented in Section 10. Section 11 concludes.

2 Design Goals

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

- A client registers to 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 Section 9 show that by avoiding per-packet authentication we can resist much stronger DoS attacks.

3 Model

Timing and communication We assume a realistic semi-synchronous timing model. 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 have a bounded drift, D, i.e., no clock progresses more than $D\%$ faster than another. Messages arrive within Δ seconds from the moment that they are sent, or are otherwise considered lost. Since Beaver uses datagram protocols such as UDP, as opposed to circuit-oriented protocols such as TCP, Δ should be thought of as some percentile of the message latencies. For example, to keep the loss rate to 1%, we select Δ to be the 99% percentile of message latencies.

Adversary The adversary can utilize compromised hosts that are not a part of the system. We also assume that some clients and admission servers might be compromised. 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 adversary has a global view of the compromised machines. 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.

The adversary can gain additional knowledge by eavesdropping on communication lines. However, 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. The number of messages the adversary can send per second is bounded by a parameter, C, representing its capacity. The adversary cannot modify messages sent by correct parties.

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 paper, we omit their definitions and simplify their behavior; for details and definitions see, e.g., [6, 4, 5]. We explicitly use the following mechanisms in this paper:

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 $Prob\,(A(e, E_e(m_L)) = L) < Prob\,(A(e, E_e(m_R)) = L) + \epsilon(\kappa)$, 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).

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 $\frac{1}{2}$).

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 $m' \neq m$. Notice that every pseudorandom function is also a MAC.

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

4 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 [3, 9].

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 (see Section 9).

Together, the ADMs and ϕ-Hopper are very effective against DoS attacks. In Section 9 we give experimental results for two implementations of ϕ-Hopper, and show that an ongoing session is almost unaffected by severe DoS attacks. These measurements validate a formal analysis of port-hopping and its guarantees given in previous work [3]. In Section 8 we give an analysis of the entire Beaver architecture and its guarantees, proving, among other things, that even when a heavy DoS attack is launched, clients can still start new sessions with the server, and that compromised clients cannot severly harm the server's performance.

4.1 Sessions in Beaver

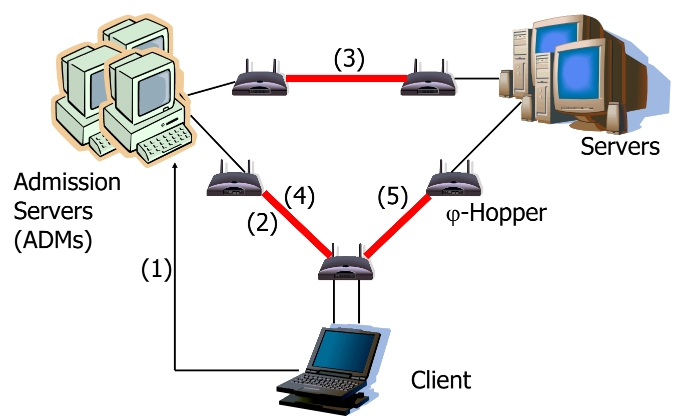


Figure 1. Beaver's architecture, where ϕ-Hopper operates in tunnel mode (marked in bold lines).

Figure 1 illustrates Beaver's architecture, and shows how a session is established: (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.

The resources allocated by the server for communicating with a specific client are freed when the client notifies the server that it wishes to cease communication, or when no valid message has been received from the client for a certain, predetermined amount of time. This procedure is very similar to cookie-based authentication used in web servers, where cookies expire after some inactivity period and the session must be reestablished to communicate with the server once more. An additional similarity is that the server allows a client to have only one active session at a time.

The resources used by the server to communicate with the ADM are allocated at boot time and never freed, as it is always possible for clients to start new sessions. Since the server does not expect many clients to start new sessions at the same time, these ADM-communication resources are low compared to the resources allocated for ongoing sessions with clients. The server rate-limits each ADM to the anticipated maximum rate (with high probability) of incoming connection requests from valid clients through that ADM.

5 ϕ-Hopper

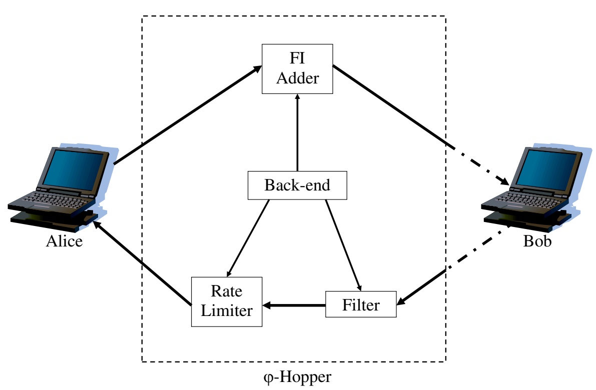


Figure 2. Communicating using ϕ-Hopper (Alice's view).

Beaver's two-party communication component, 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 [3, 9], or IPSec's security parameter index (SPI) field [2]. 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 2 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 pseudorandom 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 [3].

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 [10]. 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 be corrupt, or try to receive better service at the expense of other valid clients. Each source is allocated a maximum allowed rate that can change during the session. 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.

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 3 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, Beaver

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

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 sequence (line 4). In Beaver, the former is provided at admission time (by an ADM), and the latter – during the registration process.

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 resources are bounded and separated between clients, 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 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.

6.1 The Registration Process

Figure 4 shows the pseudocode of 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.

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.

CLIENT

register(CPK_{sig}, CPK_{dec}, cert ← ⟨ *cdata* ← { clientID, CPK_{ver}, CPK_{enc}}, $SIG_{CAPK_{sig}}$(*cdata*) ⟩)**:**
clientTS← local time
requestID← random κ-bit number
registrationInitiation ← ⟨ *data* ← { *clientID, requestID, clientTS, cert* }, $SIG_{CPK_{sig}}(data)$⟩
send *registrationInitiation* to ADM
if no valid registration completion received within timeout **then**
invalidate *requestID*
return registration failure

Upon receiving *registrationCompletion* from ADM:
if *registrationCompletion.clientID = clientID* **and** *registrationCompletion.requestID* is valid **and**
registrationCompletion.$SIG_{ADMPK_{sig}}$ is verified by $ADMPK_{ver}$ **and**
registrationCompletion.registrationApproval.$SIG_{SPK_{sig}}$ is verified by SPK_{ver} **then**
extract S_{CADM} and S_{CS} from *registrationCompletion* using CPK_{dec}

ADMISSION SERVER

Upon receiving *registrationInitiation* from client A ← *registrationInitiation.clientID***:**
if A is allowed to register **and**
(*regInitTS[A]* is uninitialized **or** *registrationInitiation.clientTS* > *regInitTS[A]*) **and**
registrationInitiation.SIG is verified by *registrationInitiation.cert.*CPK_{ver} **and**
registrationInitiation.cert.SIG is verified by $CAPK_{ver}$ **then**
regInitTS[A] ← *registrationInitiation.clientTS*
regInitID[A] ← *registrationInitiation.requestID*
RegistrationRequest ← ⟨ *data* ← *registrationInitiation, MAC*$_{S_{SADM}}$*(data)* ⟩
hopperSend(*RegistrationRequest, server*)

Upon receiving *registrationApproval* from server for client A ← *registrationApproval.clientID***:**
if *registrationApproval.requestID = regInitID[A]* **and**
registrationApproval.MAC = MAC$_{S_{SADM}}$*(registrationApproval.data)* **then**
generate shared secret S_{CADM}
data ← { *registrationApproval.data, ENC*$_{registrationApproval.cert.CPK_{enc}}(S_{CADM})$ }
registrationCompletion ← ⟨ *data*, $SIG_{ADMPK_{sig}}$*(data)* ⟩
send *registrationCompletion to* A

SERVER

Upon receiving *registrationRequest* from ADM for client A ←*registrationRequest.clientID***:**
if A is authorized to register through ADM **and**
(*regReqTS[A]* is uninitialized **or** *registrationInitiation.clientTS* > *regReqTS[A]*) **and**
registrationRequest.MAC = MAC$_{S_{SADM}}$*(registrationRequest.data)* **and**
registrationRequest.SIG is verified by *registrationRequest.cert.*CPK_{ver} **and**
registrationRequest.cert.SIG is verified by $CAPK_{ver}$ **then**
regReq[A] ← *registrationInitiation.clientTS*
generate shared secret S_{CS}
response ← { *registrationRequest.data, ENC*$_{registrationRequest.cert.CPK_{enc}}(S_{CS})$ }
registrationApproval ← ⟨ *data* ← { *response*, $SIG_{SPK_{sig}}$*(response)* }, *MAC*$_{S_{SADM}}$*(data)* ⟩
hopperSend(*registrationApproval*, ADM)

Figure 4. Pseudocode for the registration process.

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 too 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 is performed only once per several years, as long as the client's private key is kept secret.

6.2 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 [8], and it can be employed here since replicating an ADM is cheap and easy, as opposed to, say, replicating the server.

The admission process commences and proceeds as follows (see pseudocode in Figure 5):

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 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 ϕ-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.

Figure 6(a) shows the messages passed during the admission process if all procedures succeed. Figure 6(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 6(c) shows a case where the client delays its response to the ADM's challenge, perhaps due to some un-

CLIENT

open:

clientTS$\leftarrow$ local time
requestID$\leftarrow$ random κ-bit number
connectionRequest$\leftarrow$ $\langle$ *data* $\leftarrow$ $\{$ *clientID, requestID, clientTS* $\}$, $MAC_{S_{CADM}}$*(data)* $\rangle$
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 $\leftarrow$ $\langle$ *data* $\leftarrow$ *challenge.data*, $MAC_{S_{CADM}}$*(data)*, $MAC_{S_{CS}}$*(data)* $\rangle$
  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$\leftarrow$ *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] $\leftarrow$ *admissionRequest.clientTS*
  seed $\leftarrow$ *admissionRequest.clientID* $||$ *admissionRequest.requestID* $||$ *admissionRequest.clientTS*
  initHopperSession(*seed*, S_{CS}, *serverID*)
  admissionApproval $\leftarrow$ $\langle$ *data* $\leftarrow$ $\{$ *admissionRequest.data, serverID* $\}$, $MAC_{S_{SADM}}$*(data)*, $MAC_{S_{CS}}$*(data)* $\rangle$
  hopperSend(*admissionApproval, ADM*)
  if no session with A begins within *sessionInitTimeout* seconds **then**
    endHopperSession(A)

Figure 5. 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 = *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.data, 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 = *MAC*$_{S_{CADM}}$*(response.data)* **then**
admissionRequest $\leftarrow$ $\langle$ *data* $\leftarrow$ *response.data, 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 5 (continued). Pseudocode for the admission process.

expected 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 Feasibility

In this section we show that Beaver's mechanisms are light-weight, do not incur unreasonable load, and are easily deployed. Since a thorough analysis of Beaver under attack in given in Section 8, we assume all communication is legitimate. We discuss deployment options for the ADMs and server, test whether an ADM's CPU power is enough to process as many messages as its network interface permits, and measure the additional load on the server due to the use of Beaver.

For this discussion, we assume that computing a hash function takes about $1\mu s$ (as measured on a simple desktop computer), i.e., it is possible to compute $1{,}000{,}000$ hashes per second. Note that this is a conservative estimate, as a dedicated server or ADM has more CPU power than a simple desktop computer. We further simplify and assume that for our purposes, one or two hash function calculations suffice to implement a PRF or a MAC. This holds for common implementations, such as using the HMAC construction for these purposes, e.g., as in [2].

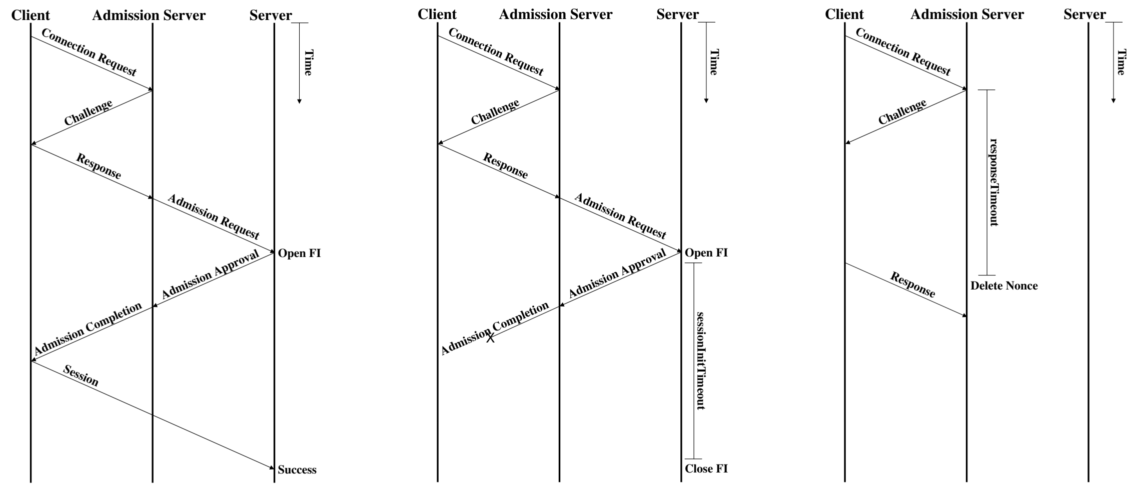
We further assume that FI hops occur every $200ms$, i.e., the FIs change 5 times a second (this is a reasonable assumption according to the analysis in [3]).

The Admission Server (ADM)

Deployment Deploying the Admission Servers is fairly easy, as they are not part of the legacy system and can be added as need requires. They can be deployed on the same LAN, or at geographically-disperse locations for better survivability. A client typically has a certificate that is valid for a particular ADM, and registers with that ADM.

Each ADM uses a different network interface and link to communicate with the servers than it does when communicating with clients. Deploying the ADMs and the servers on the same local or enterprise network facilitates the filtering of invalid traffic, as messages originating from outside the network and heading for the server cannot contain one of the ADMs as their source address, as properly configured border routers just discard such messages.

Load We analyze the CPU power needed by the ADM to support its full incoming-link's capacity. We consider a realistic scenario in which the Internet connection through



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

Figure 6. Admission process.

which the ADM receives client messages has a capacity of 100Mbps, and that each message sent or received in the admission process has a mandatory fixed length of 125 bytes ($1{,}000$ bits). This length accounts for packet headers (network, IP, and UDP), along with payload (client ID, hashes, etc.) and padding. The maximum rate of incoming client messages is therefore $100{,}000 \frac{msgs}{sec}$. When the ADM is not under attack, $100{,}000$ messages per second mean that $50{,}000$ clients are trying to gain admittance to the system at the same time (see below). This is an unreasonably high number.

These $100{,}000$ messages are sent to the server by clients performing the admission process. Each client sends the server 2 messages during the admission process – connection request and challenge response. Each such message requires one hash-table lookup and one hash to validate the message's contents. Additionally, a connection request requires one more hash to generate the nonce in the challenge. We take a conservative approach and assume that *both* the connection request and the response entail the calculation of 3 hashes. We get that all client messages together require at most $300{,}000$ hashes per second.

Other than that, the ADM communicates with the server via ϕ-Hopper, where it hops 5 times a second. Since the connection is bi-directional, the ADM needs to hop in both directions, for a total of 10 hashes per second for communicating via ϕ-Hopper, regardless of the number of admission processes running in parallel. All in all, the ADM performs at most $300{,}010$ hashes per second, about 30% of the maximum it can calculate. As the sole purpose of the ADM is to handle new connections, this leaves its CPU mostly idle.

For each client, the ADM maintains static and dynamic sets of data. The static set consists of parts of the client's registration information, such as S_{CADM}. The dynamic set is composed of the last valid timestamp that client used in a connection request, the state of the current admission process for that client (if there is one), and the nonce used in the admission process (if applicable). When the same client starts another admission process, the static set remains intact, and the dynamic set is *replaced* by a new dynamic set. Thus, the memory requirements for the ADM are always linear in the number of registered clients – one static set and at most one dynamic set for each registered client.

The Server

Deployment The legacy system consists of one or more servers providing the service, and a method to choose the server that interacts with a client during a session. As far as Beaver is concerned, these parts remain virtually unchanged. To deploy Beaver, we need to allow communication through ϕ-Hopper, and interact with the servers' potential authentication modules, or modify them.

Deploying ϕ-Hopper's front-end is straightforward. The front-end only deals with filtering messages according to their header fields and rate-limiting, and this can be easily and efficiently done in hardware by, e.g., a router, or a gateway running a firewall. Thus, the front-end is deployed at a machine that already filters messages for the server (if no such machine exists, such a machine is installed).

If the legacy server uses authenticated communication with its clients (as is common), it is most natural to deploy ϕ-Hopper's back-end (hopping logic) by integrating it with the server's authentication module, as both are likely to use the same key S_{CS}. To incur minimal load on the server, it is best to implement these functions at a machine residing between the front-end and the server, for example, an IPSec gateway working in tunnel mode. This machine is responsible for receiving filtered messages, authenticating them, forwarding them to the server, and hopping as necessary. The responses from the server are forwarded in the other direction. This deployment option requires the back-end machine to know the secrets shared between the server and its clients.

A second deployment scenario for ϕ-Hopper's back-end and the server's message authentication module is at the server itself (or its networking stack). The server then does not need to expose any client secrets to other machines. This approach resembles endpoint security mechanisms such as SSL.

The choice of where to deploy the back-end does not affect the client. The client just needs to make sure it communicates with the server via ϕ-Hopper. Since performance may not be as important on the client's side as it is on the server's side, ϕ-Hopper can even be implemented at the application layer of the client machine.

Load We now calculate the load Beaver imposes on the server. We consider the deployment scenario that incurs the most load on the server: ϕ-Hopper's front-end is deployed at a gateway or firewall, but the back-end is implemented at the server. Thus, we need to analyze the overhead caused by ϕ-Hopper's back-end, i.e., by hopping FIs.

Each FI hop takes 2 hashes per client or ADM, and there are 5 hops per second. Assume that the server communicates with $1{,}000$ clients and 10 ADMs, then we get that hopping takes $2 \cdot 5 \cdot (1{,}000 + 10) = 10,150$ hashes per second – far short of the $1{,}000{,}000$ hashes per second a weak desktop computer can perform.

It is sufficient for the server to use FIs of size 32 bits (4 bytes). The server needs to save at most 6 FIs for each communication party. The memory size needed by ϕ-Hopper is therefore $1010 \cdot 4 \cdot 6 \approx 24$KB. For ϕ-Hopper's front-end, this small memory size allows the use of an associative cache for efficient lookups, and facilitates filtering in hardware, e.g., in routers. Current routers easily provide fast filtering according to packet headers – as needed for ϕ-Hopper. For ϕ-Hopper's back-end, this means that the memory requirements for the server to support ϕ-Hopper are extremely low. All other memory requirements for the server, e.g., saving potential clients' secrets, are already part of the legacy system, and are not induced by Beaver.

8 Security Analysis

We now analyze Beaver's robustness against different attacks. We consider the goals presented in Section 2, 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 8.1). We then study the load induced by authorized communication sessions in Beaver, (Section 8.2), and proceed to discuss DoS attacks in detail, (Section 8.3). The formal proofs can be found in Appendix A.

8.1 Definitions

Definition 1 (Client validity)**.** *A client is* valid *if it possesses valid registration information. Otherwise, the client is* invalid.

Definition 2 (Message validity)**.** *A message is* valid *if it can be successfully authenticated by the receiving party. Otherwise, the message is* invalid.

Definition 3 (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 4 (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.

8.2 Sessions Load on Server

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

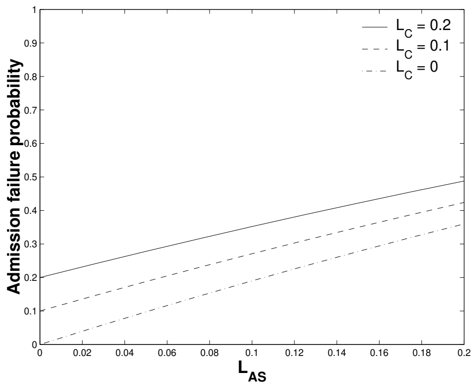
Lemma 1. *An invalid client that requests admission from a correct ADM can never successfully pass step* 3 *(response) of the admission process.*

Corollary 1. *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 2. *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, and let* sessionInitTimeout$= \tau$*. 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 2. *Let p be the fraction of compromised ADMs in Beaver. Then for* sessionInitTimeout$= 3\Delta$*, at any given time there are no more than $3\lambda p\Delta n_\phi$ open FIs for invalid sessions.*

8.3 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 3 (Admission DoS-resistance I)**.** *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 3 gives an upper bound on admission failures. This upper bound is depicted in Figure 8.3, 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%.

Lemma 4 (Admission DoS-resistance II)**.** *A DoS attack on the ADMs' incoming link from the servers does not prevent a valid client from starting a session with the server.*

We next show that Beaver achieves session DoS-resistance.

Lemma 5 (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 $\frac{C}{2^{\ell-2}}$ messages per second.*

Lemma 6 (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.*

9 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. 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.

Our implementations use a shared secret of 160 bit. At each 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.

Measurements We measure the effect authentication and hopping have on the resistance to DoS. We experiment with a 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, consuming CPU power while doing that. Every request or response fit 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.

In our first setting, we measure the advantages ϕ-Hopper offers, as compared to IPSec [2], 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 (1), 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 gets a hold of 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 7(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. The delivery probability is much lower when the SPI is known to the attacker, since this case requires complete authentication of every packet. 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 7(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, there is no use calculating the latency for such attacks.

Figure 7(c) displays the delivery probability under a bursty DoS attack, where bogus requests are not sent at constant intervals, but rather at bursts. The attack strength is measured as the average number of bogus requests per second. Comparing these results to Figure 7(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.

Figure 8(a) shows the effect of rate-limiting on the delivery probability. 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.

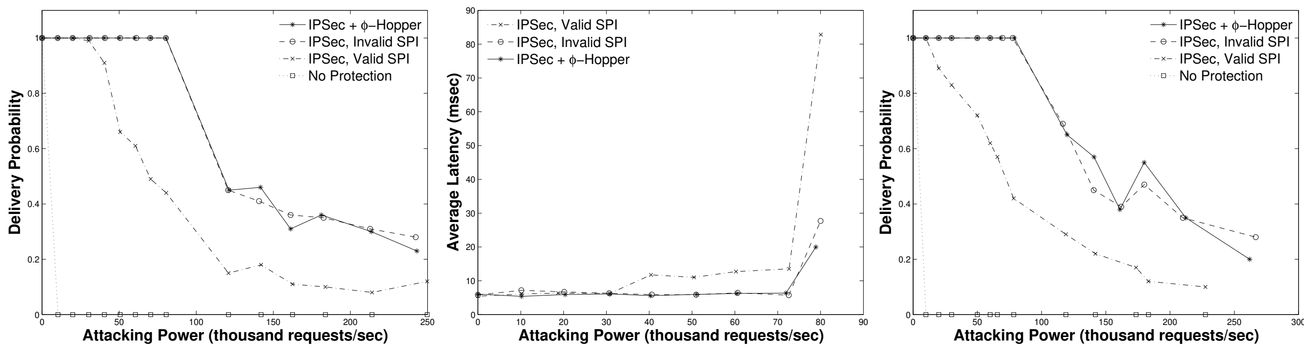
In our second setting, the client communicates directly with the web server, and we measure the effect ϕ-Hopper has when it runs on the server's machine, and not on a dedicated machine. The server runs on a Windows machine along with ϕ-Hopper (in the appropriate experiments), which performs user-level 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 8(b) 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).

We compare our results to analytical results for the delivery probability under DoS attacks, as taken from our previous work [3] (see Figure 8(c)). 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 7(a) and 8(b,c). Indeed, we can see that the actual measurements closely match the theoretical analysis.

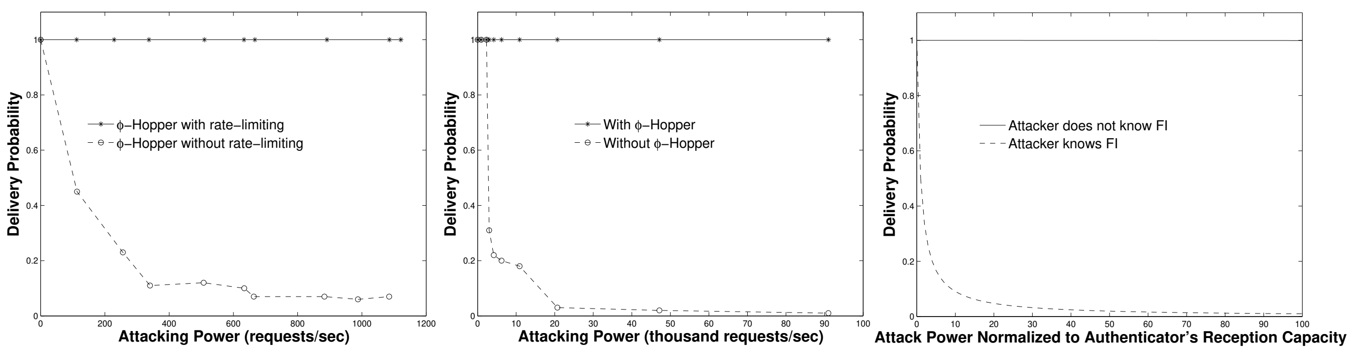
10 Related Work

The use of multiple ADMs resembles the use of overlay networks in SOS [8] and Mayday [1]. However, SOS and Mayday also screen DoS attacks by hiding the server's iden-



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

Figure 7. DoS attacks on IPSec on Linux, with and without ϕ-Hopper.



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

Figure 8. Delivery probability under DoS attacks.

tity 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 [8]. 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, it is easy to launch an application-level DoS attack that renders the server useless, but does not congest the network.

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 [7]. Beaver's significant component that uses this notion, ϕ-Hopper, is based on ideas that have been suggested before [3, 9]. 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 decided on during the admission process, and is used as virtual time. Thus, no time synchronization between the communicating parties is needed, but rather a bounded clock drift.
2. We do not assume pre-shared keys, and provide a key distribution mechanism. The shared secret is distributed during the registration process.
3. ϕ-Hopper supports communication between many clients and a single server, and not just two-party communication.
4. ϕ-Hopper uses realistic rate-limiting techniques, as opposed to the purely theoretical analysis in [3] that as-

sumed 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 [9] uses no rate-limiting at all.

5. ϕ-Hopper is implemented in two variations, and we provide measurements of the actual protocol implementation, and not of its simulated behavior [9] or of an analytical anaysis of the protocol [3].

To the best of our knowledge, we are the first to present a complete system that provides strong DoS-resistance guarantees through the use of hopping.

The analysis in [3] 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 [9]. In Section 9 we have shown that the analysis in [3] 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.

11 Conclusions

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, on the expense of other clients.

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

References

[1] D. G. Andersen. Mayday: Distributed filtering for Internet services. In *USITS*, 2003.

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

[3] 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.

[4] O. Goldreich. *Foundations of Cryptography: Basic Tools*. 2001.

[5] O. Goldreich. *Foundations of Cryptography: Volume 2: Basic Applications*. 2004.

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

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

[8] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: An architecture for mitigating DDoS attacks. *JSAC*, 21(1):176–188, 2004.

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

[10] 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.

A Security Analysis: Proofs

Lemma 1. *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. $\square$

Corollary 1. *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 1 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. $\square$

Lemma 2. *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, and let* sessionInitTimeout= τ. *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 1 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 $\frac{\lambda k \tau n_\phi}{n_{ADM}}$. $\square$

Lemma 3. (Admission DoS-resistance I) *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 4. (Admission DoS-resistance II) *A DoS attack on the ADMs' incoming link from the servers does not prevent a valid client from starting a session with the server.*

Proof. A DoS attack on the ADMs' incoming link from the servers can only drop the admission approval message sent in step 5. By that time, the client has already been authenticated, and the server has opened the FIs for the session. Since the admission approval message was lost, step 6 will not be executed, and the client will not get an admission completion message. In that case, the client times out and tries to establish the session. Since both the server and client try to communicate, the session will begin. □

Lemma 5. (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 $\frac{C}{2^{\ell-2}}$ 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 6. (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 4.1, 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 1 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 that it does with its single established session. □