Beaver on IPSec - Protection from DoS Attacks

Oleg Romanov

Beaver on IPSec - Protection from DoS Attacks

Final Paper

In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Oleg Romanov

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

ELUL, 5768 HAIFA SEPTEMBER, 2008

The Final Paper Was Done Under The Supervision of
Associate Professor Idit Keidar
in the Department of Electrical Engineering

Acknowledgment

I deeply thank my advisor, Associate Professor Idit Keidar, for giving me the opportunity to perform this research under her supervision, for the support and close guidance that I was privileged to receive.

I thank Associate Professor Amir Herzberg and Dr. Gal Badishi for lending their experience and being an excellent research partners.

I thank Dr. Keslassy Isaac for his helpful remarks.

Many thanks to Dr. Ilana David, Mr. Viktor Kulikov, and to all software laboratory staff for all the assistance they willingly gave me.

I would like to thank my dear family for the perpetual support they have given me.

Special thanks to my fiancee Svetlana for her support through all my studies.

The Generous Financial Help Of Technion Is Gratefully Acknowledged.

Contents

List of Figures

List of Tables

Abstract

Many client-server systems on the Internet are susceptible to Denial of Service (DoS) attacks. This thesis presents *Beaver* - a client-server architecture that is robust against DoS attacks. Its main purpose is to protect client-server communication from DoS attacks, especially from flooding it with messages.

Beaver employs two DoS-protection mechanisms: one for admission of new client sessions, and another for protecting ongoing sessions. The former uses dedicated admission servers (ADMs). The use of ADMs takes the admission load off the server, so that the server is not concerned with DoS attacks on clients trying to be admitted into the system. The latter is Φ-Hopper - a two-party communication protocol that mitigates DoS attacks by filtering packets. Φ-Hopper protects client-server communication sessions from DoS attacks, but does not authenticate the communication by itself. Φ-Hopper only provides dynamic filtering and rate-limiting facilities. Together, the ADMs and Φ-Hopper are very effective against DoS attacks.

At the first stage, we design and implement a Φ-Hopper. Our implementation extends a Linux kernel's IPSec implementation. IPSec (IP Security) is a suite of protocols for securing Internet Protocol (IP) communications by authenticating and/or encrypting each IP packet in a data stream. The implementation is followed by experiments, which investigate the influence of attacking power on systems with and without our protection.

Next, we design and implement a rate-limiting mechanism also by extending Linux kernel's IPSec implementation. Next, we design and implement the admission server.

Finally, we build the whole system by putting all parts together, and by running experiments, we show that the system is robust even when DoS attacks and compromised clients are present.

List of Symbols

DoS	Denial of Service
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
IPSec	Internet Protocol Security
ESP	Encapsulating Security Payload
AH	Authentication Header
SHA-1	Security Hash Algorithm
SSL	Secure Sockets Layer
PRF	Pseudo-Random Function

1. Introduction

Denial of service (DoS) attacks, in which an attacker attempts to deplete the target's resources, are common over the Internet [9]. In client-server communication, a DoS attack may be launched by sending many bogus requests to the server. These bogus requests might consume most of the server's resources, preventing it from answering legitimate client requests. To obtain a large capacity for sending invalid requests, an attacker sometimes utilizes many compromised machines, which send the bogus requests to the server in concert. This is referred to as a *distributed DoS* (DDoS) attack. Since in this research we are not concerned with the source of an attack, we simply use the term DoS to refer to either DoS or DDoS.

The simplest way to perform a DoS attack, independent of the target's specifics, is to congest the network leading to or from the target. However, such an attack requires large transmission capacity, is easy to detect, and commercial solutions that solve this problem already exist [22]. But even if the network is not congested by the attack, it is still possible to overload the server so that it cannot answer valid client requests [18]. Attacks that do not congest the network are more difficult to detect. The more resources, e.g., processing time, the server allocates per request, the easier it is for the attacker to overload the server without congesting the network. In this research, we deal with DoS attacks that do not congest the network, but may still degrade the service provided to the clients. We compare the effectiveness of authentication-based DoS-resistance solutions by measuring the performance of real system implementations under various DoS attacks. This empirical study results in important insights regarding DoS attacks and defenses.

The first approach that we examine is using per-packet authentication, as done in IPSec [3] for example. IPSec uses shared secret keys to provide per-packet authentication. In [10] it is argued that providing per-packet authentication for valid client-server traffic is sufficient to prevent a DoS attack, since bogus requests are identified and discarded. Indeed, IPSec helps in mitigating the effects of DoS attacks, as servers usually perform much more work per request than IPSec needs in order to validate the request. In our experiments we show that a simple HTTP server that is not protected against DoS attacks collapses when faced with 10,000 bogus requests per second. When IPSec is deployed to authenticate communication, the system can withstand almost up to 30,000 bogus requests per second and still answer virtually all valid client requests.

Although IPSec helps defend against medium-strength attacks, its shortcoming is that calculating the authentication information for each packet requires substantial CPU power for large volumes of traffic, and may in effect shift the DoS problem from the server to the authentication module. For example, in our experiments, at 80,000 bogus requests per second and IPSec deployed, the server manages to answer only about 45% of the valid requests, while a 100 Mbit network becomes congested at about 150,000 requests per second.

In addition to authentication information, the IPSec header includes a 32-bit Security Parameter Index (SPI) field, which is unique for a flow. A packet that does not have a valid SPI is discarded, while a packet that contains a valid SPI goes through IPSec's authentication phase. In this research, we show that it is possible to significantly boost the resilience of IPSec to DoS attacks by using a *random* SPI value that is *unknown* to the attacker, and thus reducing the number of cryptographic operations performed. Thus, proposals to use a predictable SPI value [1] are doomed to provide only a weak defense against DoS attacks, since the adversary can craft all of its bogus packets to reach IPSec's computationally-intensive authentication phase. Nevertheless, even using a random SPI value only provides temporary protection if the SPI value is fixed for the entire IPSec session, as is typically the case. In reality, attackers may eventually discover a session's SPI value, either by intercepting a message pertaining to the session as it traverses the Internet, or by observing the effects of their own actions (a DoS attack succeeds when a correct SPI is targeted).

In order to avoid using a fixed SPI, and still reduce the number of cryptographic computations under heavy attacks, one can employ authentication information that changes over time, and not per packet [7, 17]. Each packet contains a *filtering identifier* (FI), taken from a secret pseudorandom sequence, known only to the two communicating parties. In this research, we use IPSec's SPI field as the FI. The pseudorandom sequence is locally generated by the client and the server using a shared secret key (as in IPSec), and each client shares a different key with the server. Secret keys are common in existing client-server communication systems, e.g., SSL-based transactions, or IPSec-based VPNs. Every fixed time interval, the FI is chosen to be the next number from the sequence. A party that receives a packet validates the FI against the expected value. If there is no match, the packet is discarded and no further processing is performed. This approach is called *FI hopping*. FI hopping requires less processing time when dealing with high volumes of traffic (as in a DoS attack), since the FI needs to be recalculated only once per time interval, e.g., 5 seconds, and not per packet. Naturally, many packets may be transmitted during a single time interval. On the other hand, the interval can be short enough so that an attacker will not have time to detect the FI value in use and react to it. Recall that a real-world attacker usually employs thousands of zombie machines, and coordinating all of them to start employing a discovered FI value may take a long time, perhaps even 10 seconds or more.

We compare implementations of the two methods, per-packet authentication and FI hopping. We call our implementation of FI hopping Φ-Hopper. The per-packet authentication used in our experiments is a standard Linux IPSec implementation. The Φ-Hopper implementation presented here is a refinement of the ideas presented in [7, 17]. These papers suggested hopping in the context of ports, and communicating with a single client, while providing no real implementation. We implement and deploy a protocol that supports communication from many clients to a server (can be extended to a server farm), and can use various header fields of different lengths to hold the FI, e.g., IPSec's SPI field. Alternatively, the FI can be appended to the packets in transit. We describe an implementation of Φ-Hopper by modifying a Linux kernel's IPSec [3] implementation. Both IPSec and Φ-Hopper use SHA-1 [21] as a pseudorandom function (PRF) [11] for the calculation of the authentication information. For simplicity, for the rest of this paper we neglect the probability that the adversary can forge the PRF without knowing the secret key.

Φ-Hopper includes a rate-limiter that protects the server from corrupt legitimate clients, instead of just letting authenticated communication pass through, as IPSec does. It is common that valid clients get corrupted by a virus or a worm [24], and these clients may behave unexpectedly, possibly overloading the server.

We provide measurement results for HTTP traffic over UDP or TCP and for file transfers over TCP. When the communication is not authenticated, we show that the server crashes even when the attack strength is light. Additionally, with no authentication in place, it is easy to tear down a TCP connection using a low-rate DoS attack [16] or a single RST packet [27]. We validate these results in our experiments. For authenticated communication, we show that IPSec alone can only mitigate DoS attacks to a limited extent, while Φ-Hopper provides virtually perfect protection even for attacks almost three times stronger. For file transfers over TCP, even when IPSec provides adequate protection in terms of delivery probability, it incurs a severe penalty on latency, with latencies ranging from 5 to 1,000 times more than the latency exhibited by Φ-Hopper, for attacks ranging from 30,000 to 50,000 bogus requests per second, respectively. For these attacks, Φ-Hopper exhibits the same latency as the latency when no attack is performed at all. Our experimental results validate the analytical results presented in [7].

Some important insights follow from our measurements:

- A server that has no DoS protection at all collapses even under a light DoS attack.
- Per-packet authentication is effective against medium-strength attacks, but fails for attacks well under the wire speed.
- It is important to keep IPSec's SPI field unknown to the attacker at all times. To support this, the initial SPI should be random.
- FI hopping can ensure that IPSec's SPI is unknown to the attacker with high probability, and can thus leverage IPSec's capabilities to provide better DoS protection, as we show in the first real implementation and deployment of FI hopping.
- Rate-limiting traffic is important even when authentication is performed, since traffic corrupt valid clients passes authentication, and can thus consume an arbitrary amount of the server's resources. Fixed limits per flow are not adequate for bursty traffic, and it is better to be flexible and adjust rates between flows according to the actual generated traffic.

2. Design Goals

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

- 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 process should not affect ongoing sessions.
- *Admission DoS-resistance*. Protect the admission process in which registered clients create new sessions with the server.
- *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 6 show that by avoiding per-packet authentication we can resist much stronger DoS attacks.

3. 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.

Other mechanisms for mitigation of DoS attacks include the use of proxy networks [26] such as SOS [15] and Mayday [2]. This approach is different from ours, since proxy networks cause a substantial delay in latency as messages are routed through the overlay, and rely on the client not knowing the server's IP address. In contrast, the systems we examine do not require the complicated setup of an overlay network and allow the direct client-server communication, without incurring a penalty on latency. Other work focuses on quantifying DoS activity over the Internet [20], while our focus is on DoS protection.

An additional work [25] 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.

Independently of our work, Lee and Thing [17] examined the use of port-hopping to mitigate the effect of DoS attacks. 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.

Wang, et al. [26] provide simulation results for various DDoS attacks on general proxy networks, and the applications protected by them. However, they only deal with general proxy networks.

It has already been shown that DoS attacks can be harmful even when they are low-rate and do not congest the network [16, 18]. We want to gain insights on the effect of low-rate DoS attacks on systems protected using efficient authentication mechanisms such as IPSec and FI hopping. Our complete, fully-tested FI hopping implementation, Φ-Hopper, is one of our new contributions, resulting from the need to perform extensive measurements on a real implementation.

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 [12]. Φ-Hopper is based on ideas that have been suggested in [7] and in [17]. However, these previous suggestions lacked in several areas, and so Φ-Hopper differs from them in the following ways:

1. Φ-Hopper supports communication between many clients and a single server, and not just two-party communication.
2. Φ-Hopper uses realistic rate-limiting techniques, as opposed to the purely theoretical analysis in [7] 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 [17] uses no rate-limiting at all.

3. Φ-Hopper is implemented, and we provide measurements of the actual protocol implementation, and not of its simulated behavior [17] or of an analytical analysis of the protocol (as given in [7]). The analysis in [7] shows that the basic idea of hopping is very effective against DoS attacks, but does so under simplified network and rate-limiting models. In Section 6 we have shown that the analysis in [7] gives a good estimate of realistic results, using a real implementation of all of Φ-Hopper's components. 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 [17].

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 with high probability, by constantly changing the clear text filtering identifier, e.g., the SPI. Our results in Section 6 show that relying only on authentication to provide DoS protection is futile.

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 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. The use of ADMs takes the 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 sessions from DoS attacks, but does not authenticate the communication by itself. Φ-Hopper only provides dynamic filtering and rate-limiting facilities. Together, the ADMs and Φ-Hopper are very effective against DoS attacks.

4.1 Sessions in Beaver

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 a new Φ-Hopper 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.

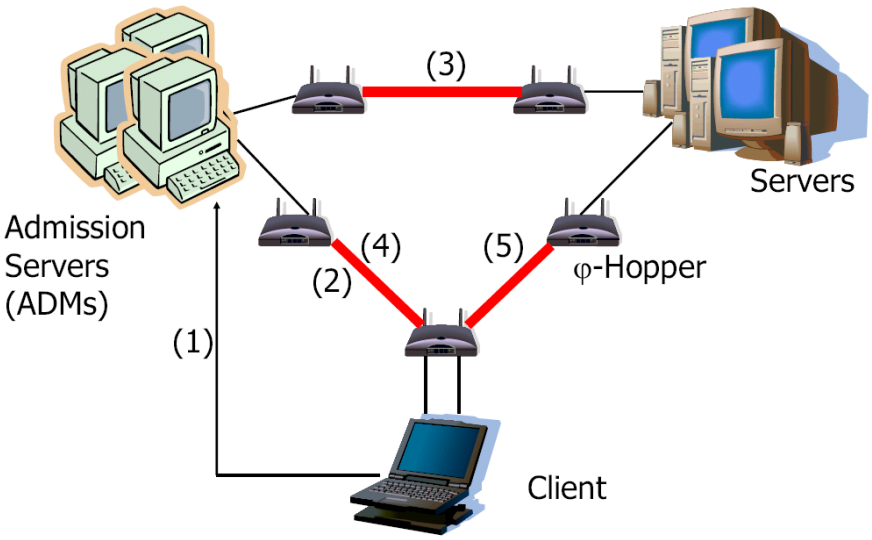


Figure 1: Beaver's Architecture.

The resources allocated by the server for communicating with a specific client are freed when the client notifies the server that it wishes to terminate communication, or when no valid message has been received from the client for a given 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 re-established to communicate with the server once more. An additional similarity is that the server allows the 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.

4.2 Φ-Hopper

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 during the transmission of the message. For example, it can be a combination of IP address and ports [17], as shown in [7], 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 2 shows the decomposition of Φ-Hopper and the interaction between its various components.

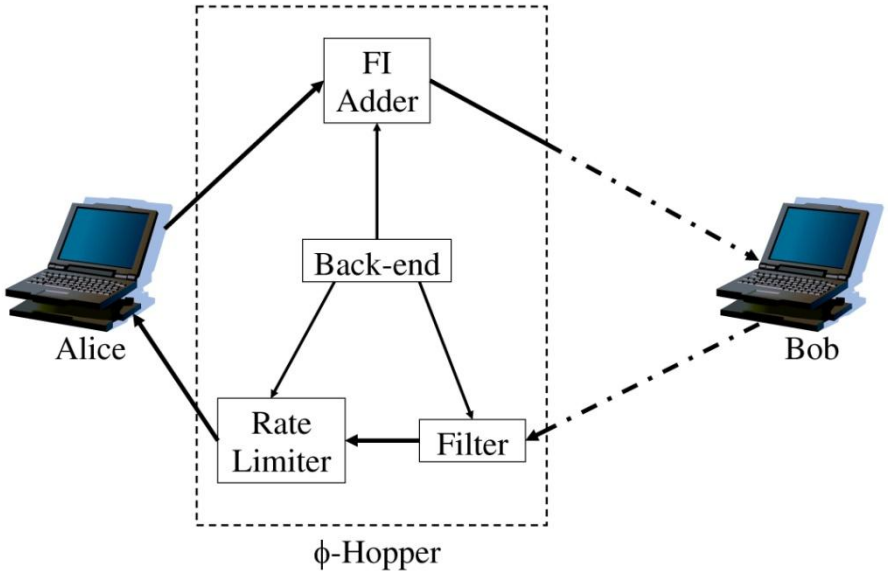


Figure 2: Communicating using Φ-Hopper (Alice's view).

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 (see [7]).

4.2.1 The front-end

The front-end can be a gateway or firewall. 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: fixed-quota (FQ) and round-robin (RR) based. When using the FQ 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 FQ 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 rate-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. The size of the queues is defined according to the number of clients and the destination’s ability to serve arriving requests. 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. 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.

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.

4.2.2 The back-end

Each party communicating via Φ-Hopper uses its clock to determine its current position in the pseudorandom sequence for incoming and outgoing messages. A Φ-Hopper session between two parties is initialized using a shared secret key used for generating the pseudorandom sequence.

During session initialization, each party allocates bounded resources for communication in this session. Φ-Hopper allocates separate resources for each active client, which are freed when the session for that client ends.

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 separately 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.

To compensate for loose time synchronization between the parties, each party keeps multiple open FIs at the receiving end. Every fixed time interval t, Φ-Hopper performs a *hop*, where it closes the oldest open FI and opens one new FI.We say that each party *opens FIs* for communication, when these FIs are added to the list of valid FIs, and *closes FIs,* when these FIs are invalidated.

5. Implementation

5.1 Φ-Hopper

The implementation of Φ-Hopper was done inside the Linux Kernel 2.6 within the "AF_KEY" module. This module is responsible for creating and managing PF_KEY sockets. PF_KEY [19] is a socket protocol family used by trusted privileged key management applications outside of the kernel to communicate with the kernel. In our work, PF_KEY sockets are used by *ipsec-tools* [14] for initializing / finalizing IPSec services by sending appropriate requests to the kernel.

When IPSec is applied between two gateways, a separate IPSec channel is created for incoming and outgoing directions (see Figure 3). IPSec uses shared secret keys (private_key) to provide per packet authentication.

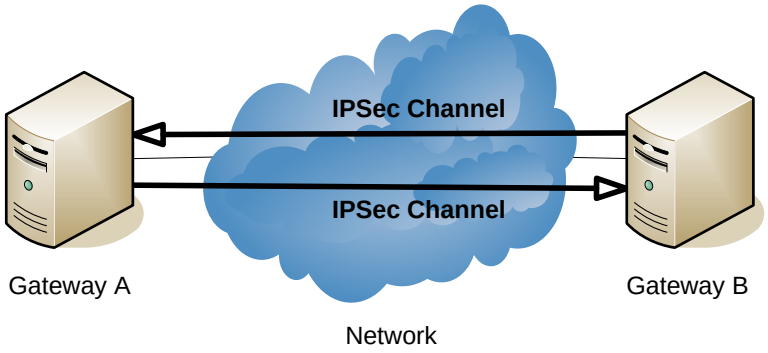


Figure 3: IPSec channel communication between two gateways.

IPSec stores a database (state) for each channel. In addition to authentication information, the IPSec state includes a 32-bit Security Parameter Index (SPI) field, which is unique for each channel. Every arrived packet that does not have a valid SPI is discarded, while a packet that contains a valid SPI goes through IPSec's authentication phase.

The purpose of Φ-Hopper is to change the SPI of each channel in both gateways, so that in every given time there will be the same SPI for the same channel in both gateways. Because of the fact that it is impossible to synchronize both gateways perfectly, and the delivery of packets takes time, the receiving gateway should agree to accept packets with a number of previous SPIs.

In our implementation of Φ-Hopper, in order to compensate the lack of synchronization between the gateways, we store a linked list of incoming and outgoing states in the Φ-Hopper extension to IPSec. For each outgoing channel, only one current state is saved. For each incoming channel, in every given time, in addition to the current state, k previous and k next states are saved; totally $(2k+1)$ states, where k is a parameter. In our experiments, we used $k=2$.

Each element of the linked list of incoming and outgoing states has all the data that IPSec holds for every created channel and in addition, it has a lifetime identifier that maps between the elements of linked list and $(2k+1)$ stored states $[-k,...,k]$. In more details, each element of liked list has:

- Destination address of IPSec channel (daddr)
- Source address of IPSec channel (saddr)
- SPI (Security Parameter Index) of the IPSec channel
- Communication protocol
- Network family (IPv4 or IPv6)
- Private key (shared secret) that is specified when IPSec channel is created and initialized for the first time (private_key)
- Lifetime identifier of the state - e.g., $-k$ - for the k'th previous state, -1 - first previous state, 0 - current state, 1 - first next state, k - k'th next state.

In order to hop states, a timer was defined. The time between hopping states is τ. All simulations were performed with τ equal to 5 seconds.

Every time the timer expires, old states are deleted (the old state of the outgoing channel and the k'th previous state of the incoming channel). Instead of the deleted states, new ones (with a new SPI) are created.

For example, if $\tau = 5$ seconds and $k = 2$, we store 5 states for an incoming channel - of times $(t-10)$, $(t-5)$, (t), $(t+5)$, $(t+10)$, when t is "now" mod 5 (the current time rounded downtown to the nearest value that can be divided by 5 without remainder). This can compensate for the lack of synchronization between the gateways of up to 20 seconds.

The new SPI of the new state of the *outgoing* channel is calculated by XOR'ing the private key with the source and the destination addresses and with the hashed current time rounded downtown to the nearest value that can be divided by 5 without remainder as follows:

$$\boxed{\text{New SPI} = (\text{private\_key}) \oplus (\text{saddr}) \oplus (\text{daddr}) \oplus (\text{sha-1}(\text{t}))}$$

where "sha-1" is a cryptographic hash function that produce the 160-bit hash value.

The new SPI of k'th next state of *incoming* channel is calculated by XOR'ing the private key with the source and the destination addresses and with the hashed time of k'th next state mod 5 (rounded downtown to the nearest value that can be divided by 5 without remainder) as follows:

$$\boxed{\text{New SPI} = (\text{private\_key}) \oplus (\text{saddr}) \oplus (\text{daddr}) \oplus (\text{sha-1}(\text{t+k}*\tau))}$$

where "sha-1" is a cryptographic hash function that produce the 160-bit hash value.

5.2 Rate-Limiting Mechanisms

The implementation of rate-limiting mechanisms was done inside the Linux Kernel 2.6 within the "xfrm4_input.c" file. The initialization and finalization were done within the "AF_KEY" module. All functions that were originally implemented inside the "xfrm4_input.c" file are responsible for decapsulation, parsing and validation of received IPSec packets. The validation includes: parsing the SPI, lookup for the IPSec incoming state (according to the parsed SPI), checking its expiration time, encryption and authentication.

We implemented two types of rate-limiters: Fixed-Quota (FQ) and Round-Robin (RR) based. Both implementations use timers for controlling packet counters.

Figure 4 shows the pseudo-code for the Fixed-Quota (FQ) rate-limiter implementation. The counter of transmitted packets is cleared every period of time (lines 5-8). In order to distinguish between different clients, an IP address of the client's computer is used as a filtering criterion. For every packet with valid checksum that arrives, the filtering criterion is checked (line 10). If the filtering criterion is not met, the packet is dropped (line 11). If the filtering criterion is met, the transmitted packets counter is checked (line 12). If the counter exceeds the limit, the packet is dropped (line 13). Otherwise, the transmitted packets counter is incremented and the packet is transmitted (lines 14-15).

```
(1)   Initially:
(2)     for each client in clients_list do
(3)       counter[client] ← 0
(4)     Set timer to Timeout

(5)   On wakeup of timer()
(6)     for each client in clients_list do
(7)       counter[client] ← 0
(8)     Set timer to Timeout

(9)   On receiving packet with valid checksum from client
(10)    if client does not exists in clients_list then
(11)      Drop packet
(12)      return
(13)    if counter[client] >= max_allowed_per_round then
(14)      Drop packet
(15)      return
(16)    counter[client]++
(17)    Transmit packet
```

Figure 4: Pseudo-code for Fixed-Quota (FQ) rate-limiter.

Figure 5 shows the pseudo-code for the Round-Robin (RR) rate-limiter's implementation. A separate waiting queue is created for every IPSec channel (identified as a combination of SPI, source address and destination address). Every waiting queue has a limited size; in our experiments each queue is limited to 500 packets.

For every packet with valid checksum that arrives, it is first checked if the client exists in the list of known clients (line 25). If the client does not exist, the client is added to the list and the waiting queue for newly added client is created (lines 26-27). Next, the size of an appropriate waiting queue is checked (line 28). If the size exceeds the limit, the packet is dropped (line 29). Otherwise, the packet is added to the waiting queue (line 31), and if the timer was not initialized, it is set to timeout (lines 32-33).

A limited number of packets can be transmitted every period of time (line 13). The transmission is performed "one by one" - one packet is removed from every non-empty queue and transmitted (lines 16-17). This is done until all queues become empty or until the limit for the current round is reached.

If all queues are empty during maximum allowed number of timer calls (line 9), all queues are deleted and all allocated memory is freed (lines 10-11).

```
(1)   Initially:
(2)     transmitted_in_round ← 0
(3)     Set timer to Timeout

(4)   On wakeup of timer()
(5)     if no non-empty waiting_queue exists then
(6)       empty_rounds_counter + +
(7)     else
(8)       empty_rounds_counter ← 0
(9)     if empty_rounds_counter > max_allowed_empty_rounds_counter then
(10)        Delete all waiting_queues
(11)        Remove all clients from clients_list
(12)        return
(13)    while transmitted_in_round < max_allowed_per_round and
                at least one non-empty waiting_queue exists
(14)      for each non-empty waiting_queue do
(15)        if transmitted_in_round < max_allowed_per_round then
(16)          transmitted_in_round + +
(17)          Transmit first packet from waiting_queue
(18)        else
(19)          transmitted_in_round ← 0
(20)          Set timer to Timeout
(21)          return
(22)    transmitted_in_round ← 0
(23)    Set timer to Timeout

(24)  On receiving packet with valid checksum from client
(25)    if client does not exist in clients_list then
(26)      Add client to clients_list
(27)      Create waiting_queue for client
(28)    if size of waiting_queue >= max_queue_size then
(29)      Drop packet
(30)      return
(31)    Add packet to waiting_queue
(32)    if timer is not initialized then
(33)      Set timer to Timeout
```

Figure 5: Pseudo-code for Round-Robin (RR) rate-limiter.

5.3 Servers

5.3.1 Simulation Server

Our simulation server was written in the C language and it runs under Linux as a command-line utility. The server can listen to TCP / UDP incoming connections. The protocol (TCP / UDP) and the port number for listening on are received as arguments.

Every message (request or response) that was sent from a client to the server (or from the server to a client) has a header that includes the following information:

- ID of message
- Length of message
- ID of simulation the message belongs to
- ID of current request within current simulation

The ID of the simulation and the ID of the current request are used for identifying the messages - if some messages are lost, we should know not only how many messages are lost, but also which messages exactly are lost. This information is stored in a local database and used later for providing results and statistics calculations.

Figure 6 shows the pseudo-code for the Simulation Server implementation. When the simulation starts, the server initializes the local database, creates callback for "kill" signal, and waits for new clients to connect (lines 1-3).

```
Main thread:
(1)   Initialize local database
(2)   Initialize callback for kill signal
(3)   Wait for new clients to connect

(4)   On accepting new client connection
(5)     If protocol is TCP then
(6)       Create a receiver thread for newly accepted client
(7)     else
(8)       Analyze the request
(9)       Update local database
(10)      Send the response

On receiving "kill" signal
(11)  Terminate the communication
(12)  Analyze local database
(13)  Print database information
(14)  Exit

Receiver thread:
(15)  On receiving valid packet from client
(16)    Analyze the request
(17)    Update local database
(18)    Send the response
```

Figure 6: Pseudo-code for Simulation Server.

If the server is listening for TCP connections, for every client that is trying to establish communication with the server, a separate receiver thread is created (line 5-6). The newly created receiver thread is responsible for analyzing all requests, updating the local database and sending a response to the client (lines 15-18).

If the server is listening for UDP connections, then analyzing all received requests, updating the local database and sending a response to the appropriate client is all done within the same main thread (lines 8-10).

If the server receives "kill" signal, it terminates the communication, analyzes the database, prints information from database, and exits (lines 11-14).

5.3.2 Admission Server

According to [6], an admission server (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. In the current work, only the second role (the admission process) was implemented.

5.3.3 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, $S_{S,ADM}$ (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 severely harm the admission process needs to launch a massive attack that targets most, if not all of the ADMs. It is make sense to add more ADMs to resist admission DDoS attacks instead of just adding more servers to resist DDoS attacks directly. The main reason for it is that ADMs should not be synchronized between them as servers should. In addition, ADMs can be a simple desktop computers, while servers should be strong enough to handle all client's requests.

The admission process commences and proceeds as follows (see pseudo-code in Figure 7 and Figure 8):

1. **Connection request (lines 1-8).** 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, the client terminates the admission process. The client may restart the admission process to start a session in spite of transient failures.

2. **Challenge (lines 39-48).** 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 (a random number used only once in the protocol). If no response is received within $responseTimeout < E$ 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 (lines 9-15).** The client proves it holds $S_{C,ADM}$ by responding with a MAC on the challenge sent by the ADM.

4. **Admission request (lines 49-56).** 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 (lines 24-32).** 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.

6. **Admission completion (lines 57-58).** The ADM sends a message to the client indicating that communication with the server can take place.

7. **Session (lines 16-20).** Upon receiving an admission completion message, the client starts a communication session with the server.

CLIENT

(1) **Open:**
(2) clientTS ← local time
(3) requestID ← random κ − bit number
(4) connectionRequest ← ⟨data ← {clientID, requestID, clientTS}, $\mathrm{MAC}_{S_{C,ADM}}$ (data)⟩
(5) send *connectionRequest* to ADM
(6) **if** no valid challenge received within timeout **then**
(7) invalidate requestID
(8) **return** connection failure

(9) **Upon receiving challenge from ADM:**
(10) **if** challenge. clientID = clientID **and** challenge. requestID is valid **and** challenge. $\mathrm{MAC}_{S_{C,ADM}}$ = $\mathrm{MAC}_{S_{C,ADM}}$ (challenge. data) **then**
(11) response ← ⟨data ← challenge. data, $\mathrm{MAC}_{S_{C,ADM}}$ (data), $\mathrm{MAC}_{S_{C,S}}$ (data)⟩
(12) send response to ADM
(13) **if** no valid admission completion received within timeout **then**
(14) invalidate requestID
(15) **return** connection failure

(16) **Upon receiving admissionCompletion from ADM:**
(17) **if** admissionApproval. clientID = clientID **and** admissionCompletion. requestID is valid **and** admissionCompletion. $\mathrm{MAC}_{S_{C,ADM}}$ = $\mathrm{MAC}_{S_{C,ADM}}$ (admissionCompletion. data) **and**
(18) admissionCompletion. $\mathrm{MAC}_{S_{C,S}}$ = $\mathrm{MAC}_{S_{C,S}}$ (admissionCompletion. data) **then**
(19) seed ← admissionApproval. clientID || admissionApproval. requestID || admissionApproval. clientTS
(20) initHopperSession (seed, $S_{C,S}$, admissionCompletion. serverID)

SERVER

(21) **Init**(ADMs):
(22) **for each** ADM **in** ADMs **do**
(23) initHopperSession (0, ADM. $S_{S,ADM}$, ADM. ADMID)

(24) **Upon receiving admissionRequest from ADM for client A ← admissionRequest. clientID:**
(25) **if** A is authorized to connect through ADM **and** no session with A is pending or in progress **and**
(26) (admReqTS[A] is uninitialized or admissionRequest. clientTS > $admReqTS$[A]) **and**
(27) admissionRequest. $\mathrm{MAC}_{S_{S,ADM}}$ = $\mathrm{MAC}_{S_{S,ADM}}$ (admissionRequest. data) **and**
(28) admissionRequest. $\mathrm{MAC}_{S_{C,S}}$ = $\mathrm{MAC}_{S_{C,S}}$ (admissionRequest. data) **then**
(29) admReqTS[A] ← admissionRequest. clientTS
(30) seed ← admissionRequest. clientID || admissionRequest. requestID || admissionRequest. clientTS
(31) initHopperSession (seed, $S_{C,S}$, serverID)
(32) admissionApproval ← ⟨data ← {admissionRequest. data, serverID}, $\mathrm{MAC}_{S_{S,ADM}}$ (data), $\mathrm{MAC}_{S_{C,S}}$ (data)⟩

Figure 7: Pseudo-code for the admission process (continued on next page).

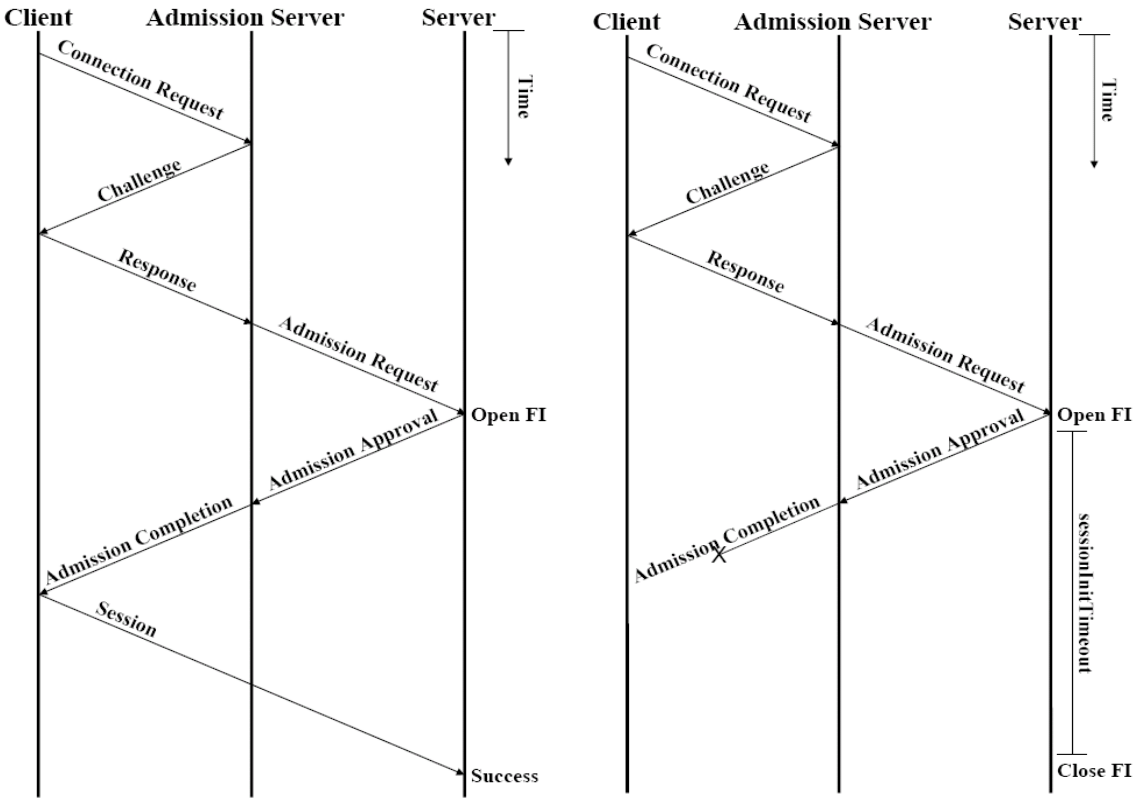
ADMISSION SERVER

```
(36) Init(serverID):
(37)   initHopperSession(0, S_{S,ADM}, serverID)

(38) Upon receiving connectionRequest from client A ← connectionRequest.clientID:
(39)   if (connReqTS[A] is uninitialized or connectionRequest.clientTS >
       connReqTS[A] ) and
(40)   connectionRequest.MAC = MAC_{S_{C,ADM}} (connectionRequest.data) then
(41)     connReqTS[A] ← connectionRequest.clientTS
(42)     connReqID[A] ← connectionRequest.requestID
(43)     nonce ← random κ − bit number
(44)     connReqNonce[A] ← nonce
(45)     challenge ← ⟨data ← {connectionRequest.data, nonce}, MAC_{S_{C,ADM}} (data)⟩
(46)     send challenge to A
(47)     if no valid response received within responseTimeout seconds then
(48)       connReqNonce[A] ← null

(49) Upon receiving response from client A ← response.clientID:
(50)   if response.clientTS = connReqTS[A] and response.requestID = connReqID[A]
       and
(51)   connReqNonce[A] ! = null and response.nonce = connReqNonce[A] and
(52)   response.MAC = MAC_{S_{C,ADM}} (response.data) then
(53)     admissionRequest ←
         ⟨data ← response.data, response.MAC_{S_{C,S}}, MAC_{S_{S,ADM}} (data)⟩
(54)     hopperSend(admissionRequest, server)
(55)     if no valid response received within responseTimeout seconds then
(56)       connReqNonce[A] ← null

(57) Upon receiving admissionApproval from server for client
     A ← admissionApproval.clientID:
(58)   if admissionApproval.requestID = connReqID[A] and
```

Figure 8 (continued): Pseudo-code for the admission process.

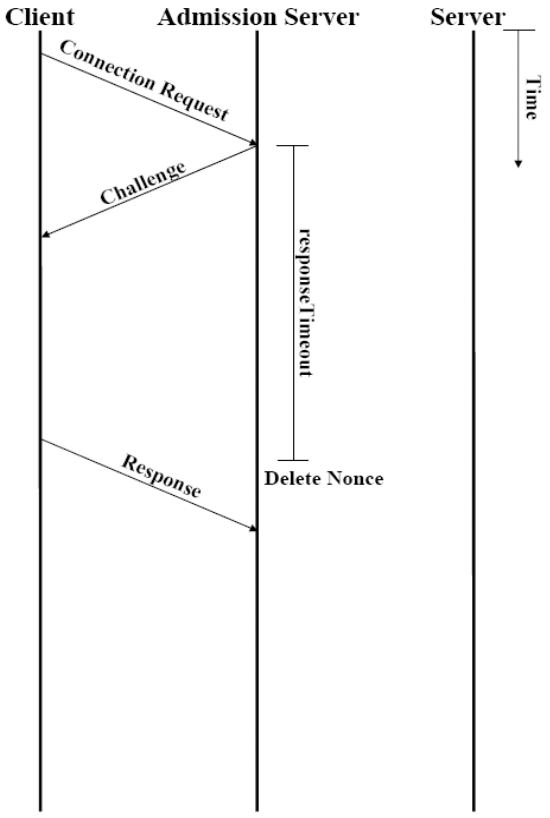
Figure 9(a) shows the messages passed during the admission process if all procedures succeed. Figure 9(b) shows a case where the admission completion message is lost, and so the client never knows that it can connect to the server. After $sessionInitTimeout$ seconds expire, the server releases the resources allocated for the session.

Figure 9(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.



(a) Correct execution.

(b) Message loss.



(c) Delayed client.

Figure 9: Admission Process.

5.4 Simulation Client

Our simulation client was written in the C language and it runs under Linux as a command-line utility. The client can connect to the server by using TCP / UDP sockets. The protocol (TCP / UDP) and the port number for connecting to the server are received as arguments.

Additional parameters that the client receives as arguments are the number of simulations to perform, the number of requests that have to be sent in each simulation, an average desired rate for sending requests, a timeout - maximum time to wait for receiving response, and an optional parameter - name of file to transfer (used for file transfer experiments).

Figure 10 shows the pseudo-code for the Simulation Client implementation. When the simulation starts, the client initializes the local database, creates callbacks for “alarm”, “kill”, and “broken pipe” signals, and connects to the server (lines 1-3).

For each simulation, the client first creates a separate receiver thread (lines 4-5), which is responsible for receiving responses to all sent requests and updating the database with received information (lines 19-21). If the timeout occurs before receiving responses to all sent requests, receiver thread returns (lines 17-18).

Next, the client prepares and sends all requests for the current simulation one by one and updates the local database (lines 6-9).

When the last simulation finishes, the client analyzes local database, performs statistics calculations and exits (lines 11-12).

If the client receives a “kill” or “broken pipe” signal, it terminates the communication, analyzes the local database, performs statistics calculations, and exits (lines 13-16).

Main thread:

(1) Initialize local database
(2) Initialize callbacks for alarm / kill / broken pipe signals
(3) Connect to server

(4) **For each** simulation **do**
(5) Create a receiver thread for current simulation
(6) **For each** request in current simulation **do**
(7) Prepare the request
(8) Send the request
(9) Update local database
(10) Wait for receiver thread of current simulation to return
(11) Analyze local database
(12) Perform statistics calculations

On receiving "kill" or "broken pipe" signal

(13) Terminate the communication
(14) Analyze local database
(15) Perform statistics calculations
(16) Exit

Receiver thread:

(17) **On** timeout
(18) Return
(19) **On** receiving packet with valid checksum **from** server
(20) Analyze the response
(21) Update local database

Figure 10: Pseudo-code for Simulation Client.

6. Experiments

Figure 11 shows a general experiment setup scheme. The server is located behind the gateway A and the clients are located behind the gateway B. All attackers and both gateways are connected together to the same network.

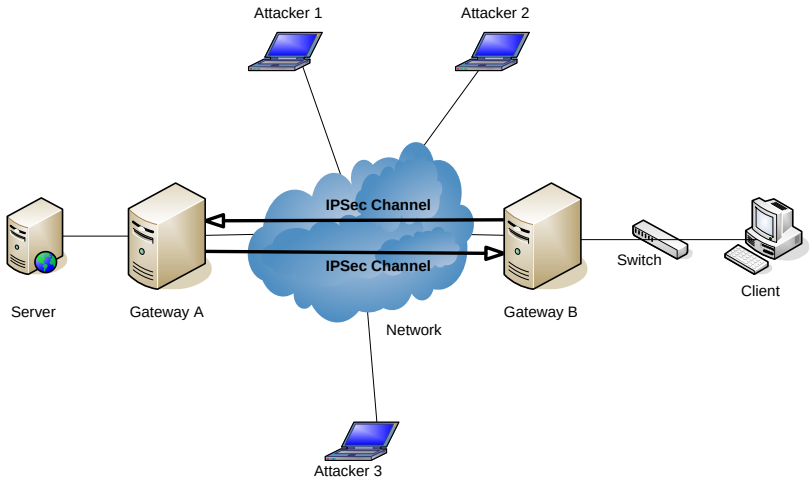


Figure 11: Experiment setup.

The gateways run Linux with IPSec in tunnel mode. The gateways have a Pentium III 650 MHz CPU and 256MB of RAM. The server and clients have a Pentium III 550 MHz CPU and 256MB of RAM. The attackers have a Pentium IV 1.8 GHz CPU and 512MB of RAM. All computers have an Intel PRO\1000 MT Desktop Adapter network card. All clients are connected to the gateway via a 3Com 10/100 switch.

The purpose of our experiments was to see the influence of the attacking power (in thousands of requests per second) on the average time for getting the response (latency) and on the number of successful responses received (delivery probability).

The results of next scenarios were compared in different experiments:

1. *No Protection:* the server has no DoS protection at all.
2. *IPSec, Valid SPI:* the gateways run IPSec in Encapsulating Security Payload (ESP) mode without encryption (authentication only), and the attacker knows the SPI used.
3. *IPSec, Invalid SPI:* the gateways run IPSec in Encapsulating Security Payload (ESP) mode without encryption (authentication only), and the attacker does not know the SPI used.
4. *IPSec + Φ-Hopper:* the gateways run IPSec in Encapsulating Security Payload (ESP) mode without encryption (authentication only) with Φ-Hopper.
5. *IPSec + Φ-Hopper (No Auth):* the gateways run IPSec in Encapsulating Security Payload (ESP) mode without encryption and without authentication with Φ-Hopper (only hopping).

When attacking, the adversary sends bogus requests at a constant rate. In scenario (2), the bogus requests carry the correct SPI field, but fail authentication. In scenarios (3), (4) and (5), the bogus requests carry an incorrect SPI field (with high probability), and so the bogus requests do not reach the authentication phase (or the server, for scenario (5)).

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 is long, an attacker has enough time to discover the SPI, e.g., 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. Scenario (5) is included to show how DoS-protection can be employed even when packet contents do not get authenticated (other than the SPI check). This scheme is faster than the one used in scenario (4), as it requires less processing time for valid traffic.

Section 6.1 presents an attacker implementation. Section 6.2 presents the results of experiments done with UDP communication. Section 6.3 presents the results of experiments done with TCP communication. Section 6.4 presents the results of experiments on Rate-Limiting mechanisms.

6.1 Simulation Attacker Implementation

Our simulation attacker was written in the C language and it runs under Linux as a command-line utility. An attacker uses RAW sockets, which allow constructing any packet with any header. This way, an attacker, for example, can construct a TCP packet and send it to the server behind the gateway so that the server will think that the packet arrived from a client. An attacker receives as arguments: the name of the protocol (can be TCP or UDP for the scenario (1) above, or can be ESP over TCP, ESP over UDP, AH over TCP, AH over UDP for the scenarios (2) - (5) above), an IP address of the source we are trying to impersonate, the port of the source, an IP address of the destination we are trying to attack, the port of the destination, the IP address of the gateway that the source is behind of, the IP of gateway that the destination is behind of, an average rate of sending requests, and optionally the SPI (used for constructing packets for IPSec protocols - ESP or AH only).

Figure 12 shows the pseudo-code for the Simulation Attacker implementation. When the simulation starts, an attacker initializes the local database, creates a callback for the “kill” signal, constructs the needed packet and sets the timeout value according to required sending rate (lines 1-4).

An attacker sends the requests to the desired destination according to the needed average rate of sending requests until it is terminated by receiving the “kill” signal or until an error occurs on sending a request (lines 5-6). The local database is updated after each sent request (line 7). An attacker waits for the timeout before sending the next request (line 8).

When an attacker receives a “kill” signal, it terminates the communication and performs some statistics calculations (lines 9-11).

```
(1)   Initialize local database
(2)   Initialize callbacks for alarm / kill signals
(3)   Construct needed packet
(4)   Set timeout value according to required sending rate

(5)   While not terminated and no error during sending packets
(6)     Send packet to destination
(7)     Update local database
(8)     Wait timeout

(9)   On receiving “kill” signal
(10)    Terminate communication
(11)    Perform statistics calculations
```

Figure 12: Pseudo-code for Simulation Attacker.

6.2 UDP Results

6.2.1 Receiving a response to a single request

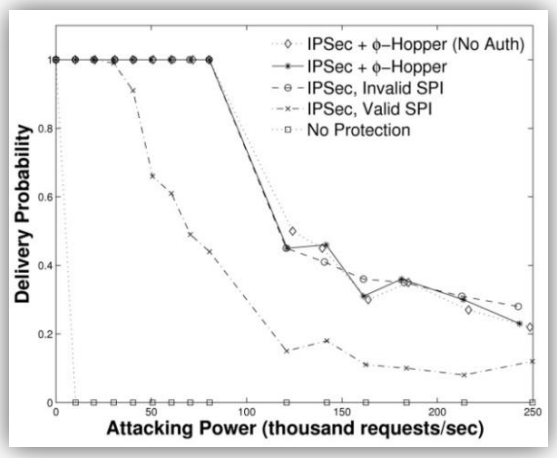
A single request was sent from the client to the server 100 times (1 request per second) for each attacking power (in thousands of requests per second) in each simulation. Every request was sent only after receiving a response to the previously sent request, or after timeout was occurred.

Figure 13(a) shows the delivery probability as the attacker's strength increases. We see that Φ-Hopper achieves the same delivery probability as when an attacker does not know the SPI used, as filtering in these cases is based on a simple comparison of a header field. We can see that when Φ-Hopper is used, the effect of authentication on the system's load is insignificant, as bogus requests do not reach the authentication phase at all. 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 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.

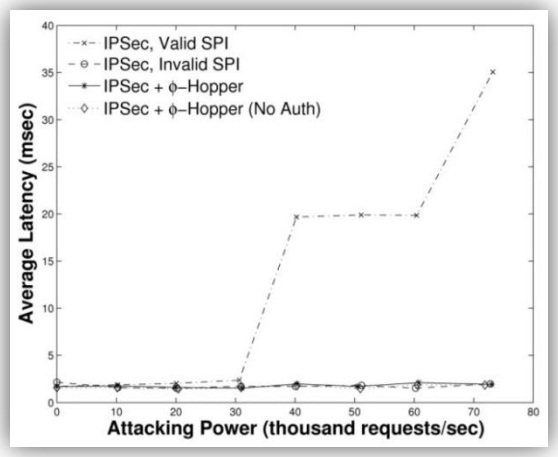
Figure 13(b) shows the effect of increasing-strength attacks on latency. In this experiment the server does not really process the request, but rather returns a reply immediately. We measure this parameter since we want to isolate the effect the algorithms run by the gateways have on latency. We can see that unless the SPI is known, the latency stays the same even when the attack strength increases. Additionally, the latency is virtually equal for Φ-Hopper with and without authentication, 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). Conversely, when only IPSec is used and the SPI is known, the latency is increased by tenfold and more even for mild attacks. Since the delivery probability is low for

attacks stronger than the ones plotted, it is meaningless to calculate the latency for such attacks.

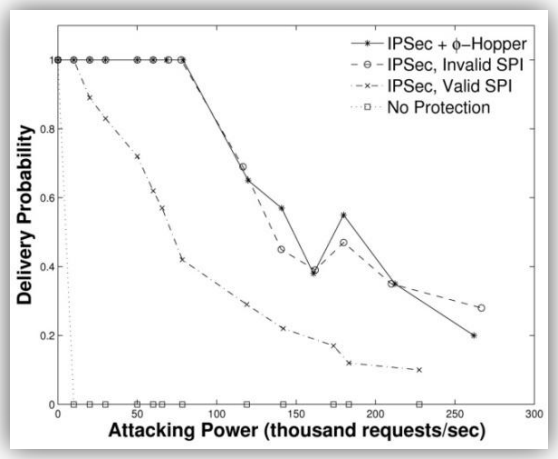
Figure 13(c) displays the delivery probability under a bursty DoS attack, where bogus requests are not sent at constant intervals, but rather as bursts. The attack strength is measured as the average number of bogus requests per second. Comparing these results to Figure 13(a), we observe that a bursty 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. We compare our results to analytical results for the delivery probability under DoS attacks. Figure 13(d) shows when the total sending rate (attacker + client) is k times the server's capacity, the delivery probability is $1/k$. The theoretical analysis assumes the attacker's sending rates are uniformly distributed, and thus the results shown in this figure can be compared to Figure 13(a). Indeed, we can see that the actual measurements closely match the theoretical analysis.



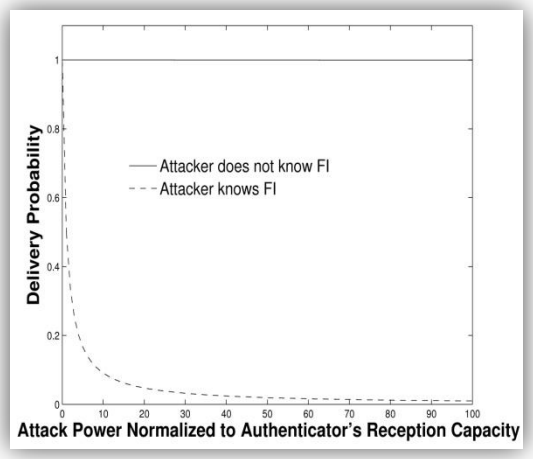
(a) Delivery probability.



(b) Latency (RTT).



(c) Delivery probability, bursty attacker.



(d) Theoretical values

Figure 13: Dos Attacks on IPSec with and without Φ-Hopper (UDP).

6.3 TCP Results

6.3.1 TCP socket connection establishment

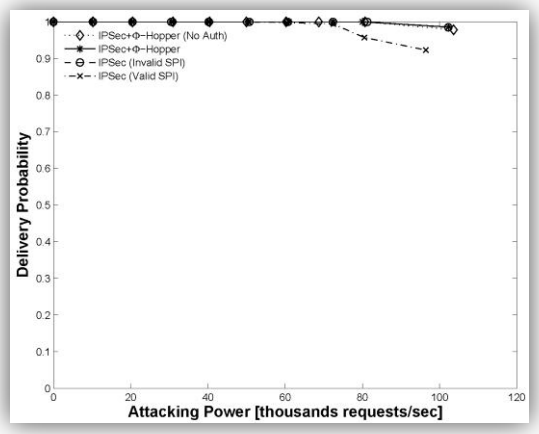
The purpose of this simulation was to see the influence of the attacking power on the average time for establishing a TCP socket connection (TCP's 3-way handshake) and on the percentage of connections that are successfully established.

A client tries to establish TCP socket connection with server 1000 times for each attacking power (in thousands of requests per second) in each simulation for scenarios (2) - (5). In scenario (1), when the server has no DoS protection at all, it crashes even when the attack is very weak (10,000 requests per second), reducing delivery probability to 0.

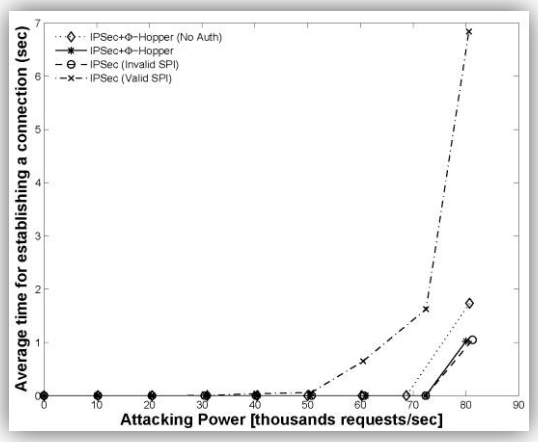
From Figure 14(a) we can see that when the gateways run IPSec in Encapsulating Security Payload (ESP) mode without encryption (authentication only), and the attacker knows the SPI used (IPSec - valid SPI), at a rate equal to 80,000 messages per second, the percentage of connections established falls by about 5%. In all other experiments, 100% of sessions are established.

From Figure 14(b) we can see that when the gateways run IPSec in Encapsulating Security Payload (ESP) mode without encryption (authentication only), and the attacker knows the SPI used (IPSec - valid SPI), at a rate equal to 80,000 messages per second the average time for establishing a connection is about 7 seconds.

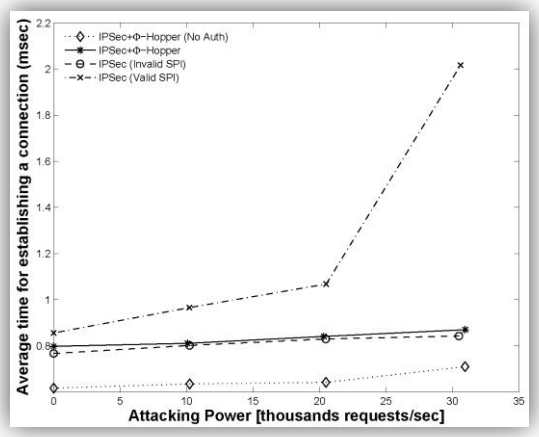
Figure 14(c) zooms in on lower attack rates. We can see that when the gateways run IPSec in Encapsulating Security Payload (ESP) mode without encryption (authentication only), and the attacker knows the SPI used (IPSec - valid SPI), already at the rate of 30,000 messages per second, the average time for establishing a connection starts growing. We see that the overhead of SPI hopping is negligible as the graph of IPSec with an invalid (unknown) SPI coincides with the graph of Φ-Hopper. On the other hand, we see that when authentication is not performed, and protection relies only on SPI hopping, the latency is lower.



(a) Percentage of connections established.



(b) Average time for establishing a connection.



(c) Average time for establishing a connection (zoomed in).

Figure 14: TCP socket connection establishment, with and without Φ-Hopper.

6.3.2 An attempt to break TCP communication

The purpose of this simulation was to check the ability of attacker to break the TCP communication between a client and the server.

We used our simulation attacker, which can construct TCP, UDP, ESP, and AH packets and send them to the destination. In order to break a communication, an attacker sends TCP packet with the TH_RST flag enabled. Upon receiving a packet with this flag, the receiver should close the TCP communication according to the TCP protocol.

Our experiments show that when the server has no DoS protection at all (TCP communication), an attacker causes the server to close the communication with the client without too much effort. The attacker only needs to know the IP address and

port of the server, the IP address and port of the client, and the IP addresses of the client's and server's gateways (if they exists) in order to encapsulate the packet. In addition, the adversary needs to discover or guess the sequence numbers used in TCP, but this can be done easily by sending many packets with increasing sequence numbers.

In case the gateways run IPSec in Encapsulating Security Payload (ESP) mode without encryption (authentication only), it becomes hard to break the communication, because the attacker needs to add authentication information on the sent packet exactly the same way the client does and this is almost impossible. We did not succeed to break the communication in that case, even when an attacker knows the SPI used.

6.3.3 Receiving a response to the single request

The purpose of this simulation was to see the influence of the attacking power on the average time for getting a response and on the percentage of successful responses received in communication in an on-going TCP session. A single request was sent from a client to the server 100 times (1 request per second) for each attacking power (in thousands of requests per second) in each simulation. Every request was sent only after receiving a response to the previously sent request, or after timeout was occurred.

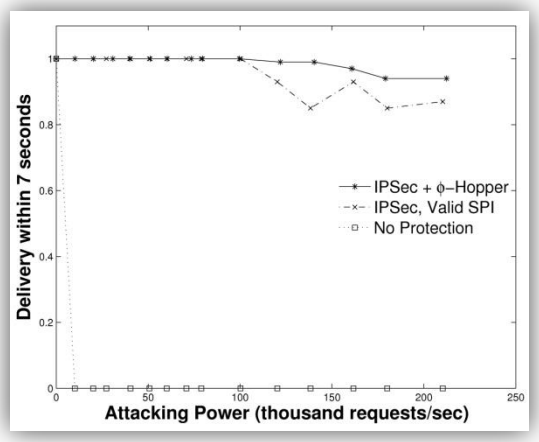
Using TCP with no IPSec protection is problematic for two reasons: First, if the adversary discovers or guesses the sequence numbers used in TCP, it can bring down the connection by sending a single RST packet (see section 6.3.2). The second problem when using TCP without client authentication is that 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 ESP, as in our UDP setting.

Figure 15(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.

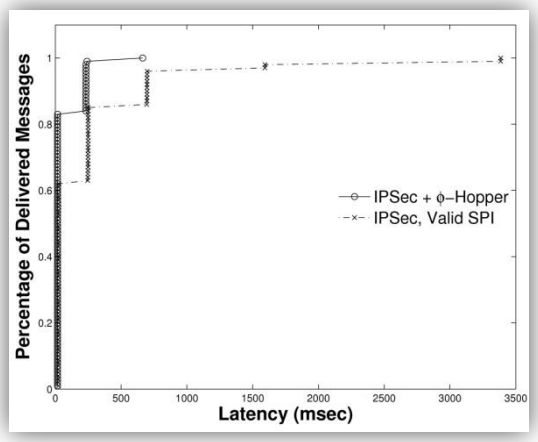
Figure 15(b) shows the cumulative distribution function (CDF) of TCP latencies (RTT) for Φ-Hopper and IPSec with a compromised SPI, for an attack power of 100,000 requests per second. We can see that Φ-Hopper provides better RTTs than IPSec with a compromised SPI. While over 80% of the messages passing through Φ-Hopper had no latency penalty (cf. data point 0 in Figure 15(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 15(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 - the probability for loss should be the same in every retransmission, therefore if the first bar is at 60%, the next should be at 60% + (60% * 40%) = 84%, as delays get about 2 times longer for each retransmission.

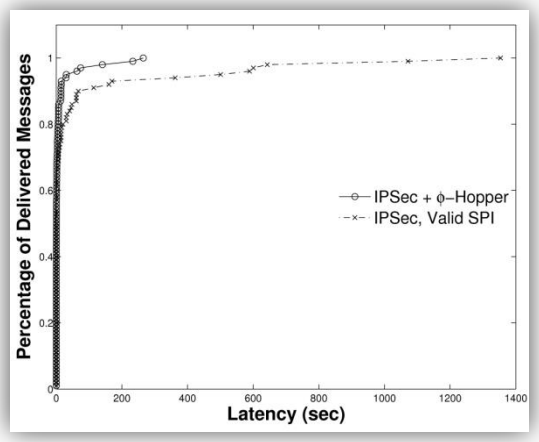
Figure 15(c) shows the CDF of TCP latencies for a stronger attack, of 240,000 requests per second. Notice that the latency in the figure is given in *seconds*, and not in *msecs*, as before. The figure clearly shows that Φ-Hopper provides reasonable latency for 85% to 90% of the messages, while IPSec's 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. Φ-Hopper degrades under strong attacks due to network congestion and not due to gateway CPU load - the network became overloaded by attacker's requests.



(a) Delivery within 7 seconds.



(b) Latency - Attack 100,000 requests/sec.



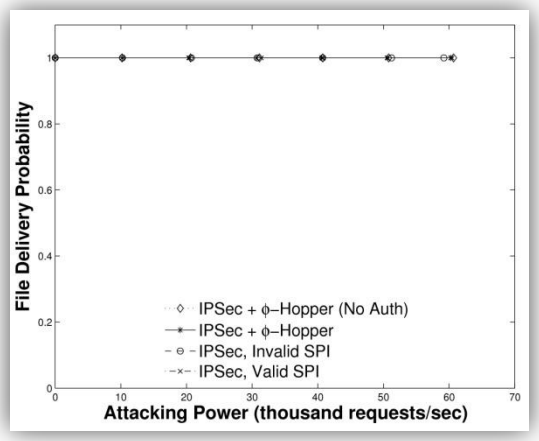
(c) Latency - Attack 240,000 requests/sec.

Figure 15: Dos Attacks on IPSec with and without Φ-Hopper (TCP).

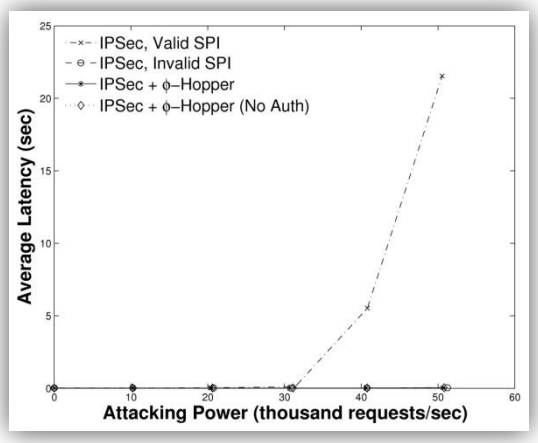
6.3.4 File Transfer via TCP

The purpose of this simulation was to see the influence of the attacking power on the average time for getting a response and on the percentage of successful responses received. A 100Kb file was sent from a client to the server 1000 times for each attacking power (in thousands of requests per second) in each simulation. Every file was sent only after receiving a response to the previously sent file, or after timeout was occurred.

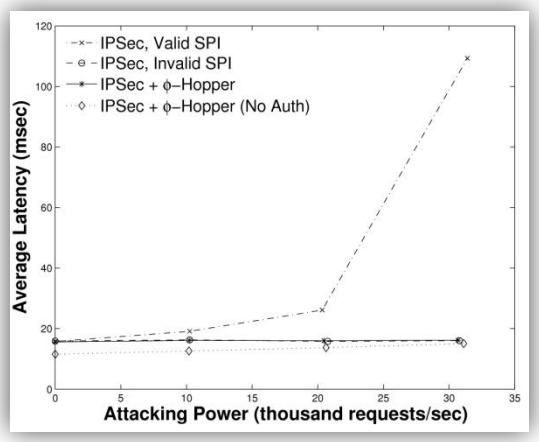
We measure the latency and probability of complete delivery of a 100Kb file over a TCP connection, in much the same way as an FTP transfer is performed. Figure 16(a) shows that all implementations manage to deliver the file to the destination when under a DoS attack. This is due to TCP's retransmission mechanism. However, Figure 16(b) shows that the latency measured when using IPSec with a known SPI is as large as 25 seconds, as opposed to a latency of a few milliseconds, exhibited by the other protocols. Moreover, all other protocols maintain roughly the same latency as the one measured when there is no attack at all. Figure 16(c) is the first part of Figure 16(b). The figure shows that even for milder attacks, IPSec with a known SPI entails latency 5 times larger than the latency measured using the other protocols.



(a) Probability to complete transfer.



(b) Latency of transfer.



(c) Latency of transfer (zoomed in).

Figure 16: TCP 100KB file transfer over IPSec, with and without Φ-Hopper.

6.4 Rate Limiting

6.4.1 Sending Rates

We experimented with three sending patterns:

- Constant rate:
  The delay between two message transmissions is constant and calculated as follows:

$$delay = \exp ected\_time - actual\_time$$

  Where

$$\exp ected\_time = \frac{number\_of\_sent\_messages}{average\_number\_of\_requests\_to\_send\_per\_second}$$
$$actual\_time = (current\_time - start\_time)$$

- Poisson rate:
  If the delay time is distributed exponentially, then the rate of sending messages is a Poisson process. Therefore, the delay between sending every two messages is calculated as follows:

$$delay = -\frac{1}{average\_number\_of\_requests\_to\_send\_per\_second} * \log(1 - drand48())$$

  Where drand48() - function that generates uniformly distributed pseudo-random numbers.

- Bursty rate:
  The messages are not sent at constant intervals, but rather at bursts. The delay between sending bursts is calculated as follows:

$$delay = bursty\_period - (current\_time - start\_time)$$

If the calculated delay is negative, which can be caused by OS timing, two packets are sent one after another without any delay between them. That way, it is ensured that on average, with high probability, the messages are sent at the needed rate.

6.4.2 Experimenting Results

We first study the effect of rate-limiting on one valid client, when the server is overloaded by one compromised client (see Figure 17).

The valid client sends requests at a constant rate of 10 requests per second. The compromised client tries to load the server by sending requests at rates between 0-1000 requests per second. The purpose of this experiment is to measure the delivery probability and the average time for getting a response at the valid client as a function of the rate of requests sent by the compromised client.

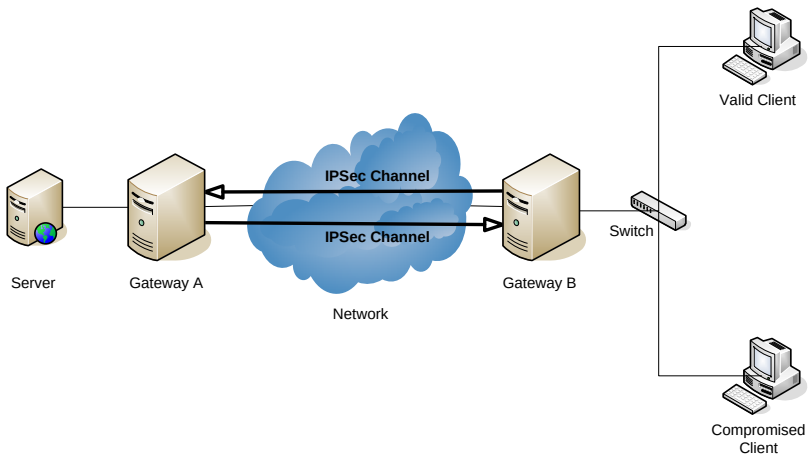


Figure 17: FQ rate-limiting - valid and compromised clients experiment.

Figure 18 shows the effect of FQ rate-limiting on the delivery probability. It can be seen 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.

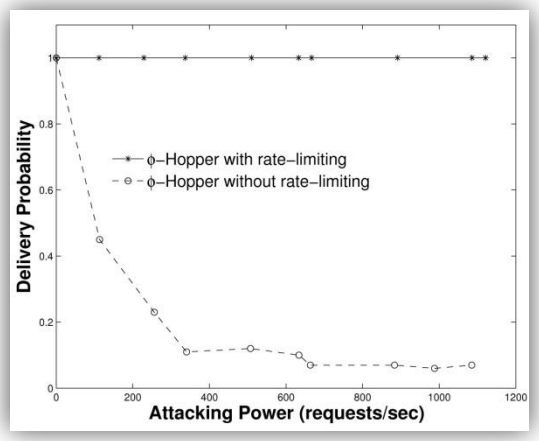


Figure 18: FQ rate-limiting (delivery probability) - 1 compromised / 1 valid client (IPSec).

Next, we study the impact of different rate-limiting schemes on valid clients in the absence of on attack. Three valid clients participate in this experiment. Each valid client sends requests with an average rate of 100 requests per second.

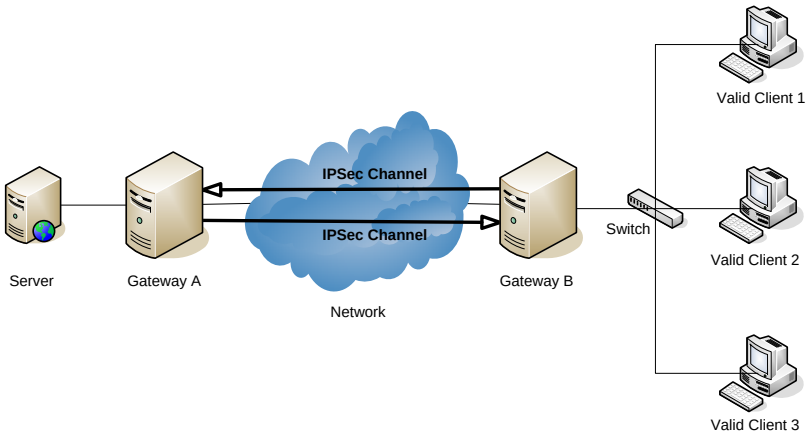


Figure 19: Rate Limiting - 3 valid clients experiment - a general scheme.

The purpose of this simulation was to compare the fixed-quota (FQ) rate-limiting to the Round-Robin-based (RR) rate-limiting algorithm. We examine the average time for getting a response and the percentage of successful responses received.

All clients send their messages either at constant intervals, or as a Poisson process, or as bursts. 1000 requests were sent from client to server in each simulation for each sending rate and for each rate-limiting algorithm.

The effectiveness of the FQ rate-limiting and the RR rate-limiting techniques are measured in these 3 scenarios (Constant rate, Poisson rate, Bursty rate), for a total of 6 experiments.

The total rate allowed by the server is set to 315 messages per second. When using FQ 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 msec. 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.

		Constant rate	Poisson rate	Bursty rate
Fixed-Quota Rate Limiting	Client 1	0.001127	0.001100	0.003060
	Client 2	0.000925	0.000883	0.002997
	Client 3	0.000919	0.000887	0.002496
	Average	**0.000990**	**0.000957**	**0.002851**
Round-Robin Rate Limiting	Client 1	0.157834	0.150451	0.632827
	Client 2	0.149934	0.150326	0.617601
	Client 3	0.148824	0.149161	0.641877
	Average	**0.152197**	**0.149979**	**0.630768**

Table 1: Rate Limiting - Average time from sending a request until getting a response [seconds].

		Constant rate	Poisson rate	Bursty rate
Fixed-Quota Rate Limiting	Client 1	100%	100%	11%
	Client 2	100%	100%	11%
	Client 3	100%	100%	11%
	Average	**100%**	**100%**	**11%**
Round-Robin Rate Limiting	Client 1	100%	100%	100%
	Client 2	100%	100%	100%
	Client 3	100%	100%	100%
	Average	**100%**	**100%**	**100%**

Table 2: Rate Limiting - Percentage of Responses received.

Table 1 and Table 2 show the average times from sending a request until getting response and the percentage of responses received respectively. We can see that, although RR rate-limiting imposes a higher average time from sending a request until getting a response due to its periodic and cyclic nature, it handles bursty traffic much better than FQ rate-limiting. While the delivery probability drops down to 11% (110 out of 1000 responses are received) for FQ 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 because RR allows all queues to share a single pool of resources, and so if one queue is empty, the other flows gain better maximum rates.

Our rate-limiting experiments show the flexibility and modularity of Φ-Hopper. Φ-Hopper can work well with different rate-limiting approaches suitable for various systems.

7. Summary

We performed an empirical evaluation of two techniques that mitigate the effects of DoS attacks on client-server communication: per-packet authentication, and FI hopping. We presented Φ-Hopper, a FI hopping protocol that supports client-server communication, and measured its resilience to DoS attacks compared to a per-packet authentication protocol, IPSec. Our empirical results provide insights to the efficiency of various client-server DoS protection schemes. For example, they show that using IPSec alone helps to some extent, but is insufficient when dealing with DoS attacks of at least moderate strength, or with corrupted clients.

In contrast, Φ-Hopper protects the communication even for much stronger DoS attacks. Our work illustrates that Φ-Hopper is robust, efficient, and easy to implement and deploy. Moreover, it can be used in conjunction with IPSec, to improve IPSec's resilience to DoS attacks.

Φ-Hopper without the authentication works better than Φ-Hopper with authentication (same DoS protection, less overhead) and thus if there is a need for DDoS protection only (not security), then the former should be used. The latter has better security properties in case a FI leaks and / or in case the authentication information is longer.

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:

- 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 separate entity and so provide separation 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 clients cannot consume much of the server's resources, at the expense of other clients.

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.

We implemented and tested our system in real conditions, and provided measurements that show that indeed Beaver is a promising solution.

Bibliography

[1] B. Adoba and W. Dixon, *RFC 3715 – IPSec-Network Address Translation (NAT) Compatibility Requirements*. Mar. 2004.

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

[3] R. Atkinson, *Security Architecture for the Internet Protocol*. IETF, 1998.

[4] R. Atkinson. (1998) Encapsulating Security Payload (ESP). RFC 2406, IETF.

[5] R. Atkinson. (1998) IP Authentication Header. RFC 2402, IETF.

[6] G. Badishi, A. Herzberg, I. Keidar, O. Romanov, and A. Yachin, "Denial of Service? Leave it to the Beaver.," Technion TR CCIT 595, July 2006.

[7] G. Badishi, A. Herzberg, and I. Keidar, "Keeping Denial-of-Service Attackers in the Dark," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 4, no. 3, pp. 191-204, Jul. 2007.

[8] J. R. Chertov, S. Fahmy, and N. B. Shroff, "Emulation versus Simulation: A Case Study of TCP-Targeted Denial of Service Attacks," in *International IEEE/CreateNet Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom)*, Mar. 2006.

[9] CSI/FBI, *Computer Crime and Security Survey*. 2003.

[10] L. Garber, "Denial-of-service attacks rip the Internet," *Computer*, vol. 33, no. 4, pp. 12-17, 2000.

[11] O. Goldreich, S. Goldwasser, and S. Micali, "How to construct random functions," *Journal of the Association for Computing Machinery*, vol. 33, no. 4, pp. 792-807, 1986.

[12] N. M. Haller, "The S/KEY One-Time Password System," in *the ISOC Symposium on Network and Distributed System Security*, Feb. 1994.

[13] G. Insolvibile, "The IP Security Protocol," *Linux Journal (http://www.linuxjournal.com/article/6117)*, Sep. 2002.

[14] IPSec_Tools. [Online]. HYPERLINK "http://ipsec-tools.sourceforge.net/" http://ipsec-tools.sourceforge.net/

[15] A. D. Keromytis, V. Misra, and D. Rubenstein, "SOS: An Architecture for Mitigating DDoS Attacks," *Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 176-188, 2004.

[16] A. Kuzmanovic and E. W. Knightly, "Low-Rate TCP-Targeted Denial of Service Attacks (The Shrew vs. the Mice and Elephants)," in *SIGCOMM*, 2003.

[17] H. C. J. Lee and V. L. L. Thing, "Port Hopping for Resilient Networks," in *the 60th IEEE Vehicular Technology Conference*, Sep. 2004.

[18] J. Li, N. Li, X. Wang, and T. Yu, "Denial of Service Attacks and Defenses in Decentralized Trust Management," in *The 2nd IEEE International Conference on Security and Privacy in Communication Networks (SecureComm)*, Aug. 2006, pp. 1-12.

[19] D. McDonald, C. Metz, and B. Phan. (1998) PF_KEY Key Management API, Version 2. RFC 2367, IETF.

[20] D. Moore, G. Voelker, and S. Savage, "Inferring Internet denial-of-service activity," in *10th USENIX Security Symposium*, August 2001, pp. 9-22.

[21] National Institute for Standards and Technology, "Secure Hash Standard (SHS)," *FIPS Publication 180-2*, Aug. 2002.

[22] Riverhead Networks (Cisco), *Products Overview*.

[23] R. Spenneberg. (2003) IPsec HOWTO (http://www.ipsec-howto.org/).

[24] S. Staniford, V. Paxson, and N. Weaver, "How to Own the Internet in Your Spare Time," in *Proceedings of the 11th USENIX Security Symposium*, Aug. 2002, pp. 149-167.

[25] A. Stavrou and A. D. Keromytis, "Countering DoS attacks with stateless multipath overlays," in *CCS*, Nov. 2005.

[26] J. Wang, X. Liu, and A. A. Chien, "Empirical Study of Tolerating Denial-of-Service Attacks with a Proxy Network," in *Usenix Security*, 2005.

[27] P. Watson, "Slipping in the Window: TCP Reset Attacks," in *CanSecWest*, 2004.

Beaver מעל IPSec - הגנה מפני התקפות מניעת שירות

אולג רומנוב

Beaver מעל IPSec - הגנה מפני התקפות מניעת שירות

חיבור על עבודת גמר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים בהנדסת חשמל

אולג רומנוב

הוגש לסנט הטכניון - מכון טכנולוגי לישראל

אלול תשס"ח חיפה ספטמבר 2008

עבודת גמר נעשתה בהנחיית
פרופסור חבר עידית קידר
בפקולטה להנדסת חשמל.

הכרת תודה

ברצוני להודות מקרב לב למנחה שלי, פרופ"ח עידית קידר, על האפשרות לבצע את המחקר הזה תחת ההנחיה המסורה שלה, ועל התמיכה הרבה לה זכיתי.

אני מודה לשותפיי למחקר, פרופ"ח אמיר הרצברג וד"ר גל בדישי, על התרומה הרבה מניסיונם האישי.

אני מודה לד"ר יצחק קסלסי על ההערות המועילות לעבודה זו.

הרבה תודות לד"ר אילנה דוד, למר ויקטור קוליקוב, ולכל צוות המעבדה לתוכנה על העזרה והתמיכה לה זכיתי.

אני רוצה להודות למשפחתי היקרה על התמיכה והעידוד התמידיים להם אני זוכה.

תודה מיוחדת לארוסתי סבטלנה על התמיכה לאורך כל תקופת הלימודים שלי.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

תוכן עניינים

תקציר

הרבה מערכות שרת-לקוח באינטרנט רגישות להתקפות מניעת שרות (DoS - Denial of Service). בהתקפות מסוג זה מטרת התוקף היא למנוע מתן שרות חלקי או מלא ע"י המחשב המותקף. ישנן הרבה דרכים לעשות זאת, אנחנו התרכזנו במקרה בו התוקף שולח כמות גדולה של בקשות מזויפות למחשב המותקף וע"י כך מעמיס אותו. בדרך כלל, התקפת DoS כוללת מספר רב של מחשבים עליהם התוקף השתלט מבעוד מועד. מחשבים אלו מכונים "זומבים", והתוקף משתמש בהם על מנת להפעיל התקפה מתואמת כנגד מטרה כלשהי בעוצמה הרבה יותר גדולה מעוצמת התוקף הבודד.

סינון הודעות והגבלת קצב התעבורה הנכנסת הינן דוגמאות לשיטות ההגנה מפני התקפות מניעת שרות. סינון הודעות מסתמך על ערכים מסוימים שנקבעים מראש (למשל, כתובת השולח) וכל פעם שמגיעה הודעה כלשהי, המסנן משווה בין השדה הספציפי בכותרת ההודעה לבין הערך שנקבע מראש. אם הערכים זהים - ההודעה ממשיכה ליעדה, אחרת, ההודעה נזרקת. הגבלת קצב התעבורה הנכנסת אינה מבדילה בין ההודעות המזויפות לבין ההודעות הטובות, כך שההודעות הטובות יכולות להיזרק ולא להגיע ליעדן. התוקף יכול לבנות את החבילות המזויפות בעצמו, בפרט את כותרת ההודעה, ולשים שם את הערך אותו מחפש המסנן ובכך לגרום למסנן להעביר את ההודעות המזויפות ליעדן. יתרה מכך, אם האפליקציה במחשב היעד שהחבילות מגיעות אליה מבצעת פענוח ו/או הצפנה עבור כל הודעה שמתקבלת, אזי התוקף יכול לגרום לה למיצוי משאבים ע"י שליחת הבקשות בקצב שאינו מעמיס את הרשת.

ניתן להגן מפני התקפות מניעת שרות ע"י שימוש באימות קריפטוגרפי של כל חבילה שעוברת ברשת. אחת הדוגמאות הוא IPSec (IP Security) - אוסף פרוטוקולים עבור הגנה על תקשורת IP (Internet Protocol) ע"י אימות ו/או הצפנה של כל חבילת IP אשר עוברת בערוץ התקשורת. מנגנון ה-IPSec מגן על אפליקציה מפני התקפות DoS, כיוון שהחבילות המזויפות לא יכולות לעבור את המחשב המאמת. פעולת האימות היא יקרה יחסית, לכן המאמת נחשף להתקפה וממצה את משאביו באימות החבילות המזויפות. תוך כדי אימות החבילות המזויפות, החבילות הטובות יכולות להיזרק עקב חוסר משאבים לטיפול בהן. לכן, כפי שאנו מראים בעבודה זו, השימוש ב-IPSec בלבד אינו מספיק לצורך טיפול יעיל בהתקפות מניעת שרות.

אחת מהנחות העבודה שלנו היא שהתוקף מוגבל בקצב שליחת ההודעות. הקצב המקסימאלי הוא למעשה סכום הקצבים בהם יכולים לשלוח הודעות המחשבים שעליהם הוא שולט. התוקף יכול ליצור כל הודעה אפשרית, עם כל כותרת אפשרית, אך התוקף אינו מסוגל ליצור חבילה מזויפת שתעבור אימות קריפטוגרפי במחשב המאמת. התוקף יכול לראות כל החבילות העוברות ברשת, אך דרוש לו פרק זמן מסוים לשנות את התקפתו כתוצאה מניתוח החבילות העוברות ברשת, וזאת כיוון ששינוי ההתקפה מחייב מתן פקודות למחשבים הנשלטים וזה לוקח זמן.

עבודה זו מציגה את *Beaver* - ארכיטקטורת שרת-לקוח שחסינה נגד התקפות מניעת שרות. מטרתה העיקרית היא להגן על תקשורת שרת-לקוח מפני התקפות מניעת שרות, במיוחד מפני הצפת השרת בהודעות, בין אם הן מזויפות ובין אם הן טובות.

Beaver מפעיל שני מנגנוני הגנה נגד התקפות מניעת שרות: אחד עבור יצירת שיחות ע"י לקוחות חדשים, והשני עבור הגנה על שיחות שרת-לקוח שנוצרו. המנגנון הראשון משתמש בשרתי בקרת כניסה (ADMs) מיוחדים. השימוש בשרתי בקרת כניסה מוריד את העומס מהשרת שיכול להיווצר תוך כדי הכניסה, כך שהשרת לא מודאג מהתקפות מניעת שרות של הלקוחות הרוצים להיכנס למערכת. המנגנון השני הוא Φ-Hopper - פרוטוקול תקשורת דו-צדדי אשר מגן מהתקפות מניעת שרות ע"י סינון חבילות. מנגנון ה-Φ-Hopper מגן על שיחת שרת-לקוח מהתקפות מניעת שרות, אבל לא מאמת תקשורת בעצמו, Φ-Hopper רק מספק סינון דינמי ומגביל את כמות התעבורה. ביחד, ADMs ו-Φ-Hopper מאוד יעילים נגד התקפות מניעת שרות.

על מנת לספק אימות עבור כל חבילה בנפרד, IPSec משתמש במפתחות משותפים סודיים. מנגנון ה-IPSec מקטין את האפקט של התקפת מניעת שרות, כיוון שהשרתים בד"כ מבצעים הרבה יותר עבודה מאשר IPSec במהלך אימות ההודעה. בניסויים שלנו אנחנו מראים, כי שרת HTTP פשוט שלא מוגן מפני התקפות DoS כלל, מתמוטט כאשר מתבצעת עליו התקפה בקצב 10,000 הודעות לשנייה. כאשר IPSec מותקן ומבצע אימות של התקשורת הנכנסת, המערכת יכולה להתמודד עם התקפה של כמעט 30,000 הודעות לשנייה ועדיין לענות לכל ההודעות הטובות שנשלחות ע"י הלקוחות.

למרות ש-IPSec עוזר להגנה נגד התקפות בקצב בינוני, החיסרון העיקרי שלו הוא שחישוב המידע עבור האימות דורש חלק ניכר מכוח המאבד עבור כמות גדולה של התעבורה, ואפילו, יכול להזיז את בעיית מניעת שרות מהשרת לכיוון המחשב המאמת. לדוגמה, בניסויים שלנו, כאשר IPSec מותקן ומבצע אימות של התקשורת הנכנסת, ומתבצעת עליו התקפה בקצב 80,000 הודעות לשנייה, השרת מצליח לענות רק על כ-45% מכל ההודעות הטובות. בנוסף, הרשת בקצב 100 Mbit נסתמת כאשר מתבצעת בה התקפה בקצב של כ-150,000 הודעות לשנייה.

כאשר IPSec מותקן, הכותרת של כל הודעה נכנסת מכילה גם כותרת השייכת לאחד מפרוטוקולי ה-IPSec, והיא מכילה שדה בגודל 32 ביט הנקרא SPI (Security Parameter Index). ערך השדה הנ"ל הינו ייחודי עבור כל שיחה בין שרת ללקוח. כל חבילה שערך ה-SPI שלה אינו תקין נזרקת, ואילו חבילה שמכילה את הערך הנכון מועברת דרך מנגנון האימות של ה-IPSec במידה והוא מופעל. בעבודה זו אנו מראים שניתן להגביר את עמידותו של ה-IPSec כנגד התקפות מניעת שרות ע"י שימוש בערך SPI אקראי ולא ידוע לתוקף, ובצורה זאת, להקטין משמעותית את כמות הפעולות הקריפטוגרפיות הנעשות בתהליך האימות. למרות זאת, אפילו השימוש בערך אקראי של ה-SPI מספק רק הגנה זמנית כל עוד ערך ה-SPI הינו קבוע למשך כל שיחת ה-IPSec. במציאות, התוקפים יכולים לגלות את ערך ה-SPI ע"י ניתוח ההודעות העוברות ברשת או ע"י תגובה לפעולות כלשהן שיזמו (למשל, התקפת DoS הצליחה כאשר נוחש ה-SPI הנכון).

כדי להימנע משימוש ב-SPI קבוע ועדיין להוריד את כמות הפעולות הקריפטוגרפיות במהלך התקפות כבדות במיוחד, אנחנו מציעים להפעיל על כל חבילה מידע מאומת שמשתנה עם הזמן. כל חבילה מכילה (FI) *filtering identifier* שנלקח מסדרה סודית פסיאודו-אקראית, אשר ידועה רק לצדדים המתקשרים. בעבודה זו, אנו משתמשים בשדה ה-SPI בתור FI. הסדרה הפסיאודו-אקראית נוצרת ע"י השרת והלקוח בנפרד תוך שימוש במפתח סודי משותף (כמו ב-IPSec). כל לקוח משתף מפתח סודי שונה עם השרת. כל פרק זמן קבוע, המספר הבא בסדרה נבחר להיות ה-FI. הצד שמקבל את החבילה, משווה את ערך ה-FI עם הערך הצפוי. אם ערך ה-FI שונה, החבילה נזרקת. גישה זו נקראת *FI-hopping*. מנגנון ה-FI hopping דורש פחות זמן עיבוד כאשר עוסקים עם כמויות גדולות של תעבורה (כמו בהתקפות DoS), מפני שצריך לחשב את ה-FI מחדש רק פעם בפרק זמן מסוים (למשל, פעם ב-5 שניות) ולא עבור כל חבילה. בשיטה זו, הרבה חבילות יכולות להישלח תוך אותו פרק זמן, ומצד שני, פרק הזמן הנ"ל יכול להיות קצר מספיק כך שהתוקף לא יספיק לגלות את ערך ה-FI ולהגיב בהתקפה מתאימה.

בשלב הראשון, תכננו ומימשנו את ה-Φ-Hopper. המימוש כלל הרחבת IPSec בתוך הגרעין של לינוקס. בתום המימוש נעשו ניסויים אשר חקרו את השפעת כוח ההתקפה על המערכת עם הפרוטוקולים המקוריים והמורחבים. הניסויים נעשו בעזרת השרת, הלקוח והתוקף שמומשו על ידינו.

בשלב הבא תכננו ומימשנו את המנגנון הגבלת כמות התעבורה גם על ידי הרחבת IPSec בתוך הגרעין של לינוקס. בשלב זה, העבודה גם כללה תכנון ומימוש של שרת בקרת כניסה.

לסיום, בנינו את המערכת כולה ע"י חיבור בין כל החלקים, והרצנו ניסוים בהם אנחנו מראים שהמערכת חסינה אפילו כאשר ישנן התקפות מניעת שרות וקיימים לקוחות אשר מעמיסים את הרשת.

אנחנו מציגים מדידות עבור זרימת ה-HTTP מעל תקשורת TCP או UDP ומדידות עבור העברת קבצים מעל TCP. כאשר התקשורת אינה מאומתת, אנחנו מציגים שהשרת מתמוטט אפילו כאשר קצב ההתקפה מאוד נמוך. בנוסף, כאשר התקשורת אינה מאומתת, קל יחסית לגרום לשרת לסגור את שיחת

ה-TCP שלו עם הלקוח. עבור תקשורת מאומתת, אנחנו מראים ש-IPSec לבד יכול להגן נגד ההתקפות מניעת שרות בצורה מוגבלת, כאשר Φ-Hopper נותן הגנה כמעט מושלמת אפילו מפני התקפות החזקות פי 3. עבור העברת קבצים דרך TCP, אפילו כאשר IPSec נותן הגנה במובן של הסתברות הגעה, הוא מגדיל בצורה חמורה את זמן ההעברה, כאשר זמני ההעברה נעים בין פי 5 ל-1,000 יותר מאשר זמן ההעברה שהתקבל ע"י שימוש ב-Φ-Hopper, וזאת עבור ההתקפות שנעו בין 30,000 ל-50,000 הודעות לשנייה בהתאם. עבור התקפות אלו, זמן ההעברה שהתקבל ע"י שימוש ב-Φ-Hopper, שווה לזמן ההעברה כאשר ההתקפה לא בוצעה כלל.

<u>להלן מספר מסקנות הנובעות מהמדידות שלנו:</u>

- שרת שלא מוגן כלל מפני התקפות מניעת שרות מתמוטט, אפילו כאשר קצב ההתקפה מאוד נמוך.
- אימות פר חבילה הינו אפקטיבי נגד התקפות בקצב בינוני, אבל נכשל נגד התקפות בקצב גבוה יותר.
- חשוב מאוד לשמור כל הזמן את שדה ה-SPI של ה-IPSec בסוד מפני התוקף. לצורך השגת מטרה זו, הערך ההתחלתי של ה-SPI חייב להיות אקראי.
- מנגנון ה-FI hopping יכול להבטיח ששדה ה-SPI של ה-IPSec יישאר בסוד מפני התוקף בהסתברות גבוהה ולכן יכול לשדרג את יכולות ה-IPSec ע"מ לספק הגנה טובה יותר מפני התקפות מניעת שרות.
- מנגנון הגבלת כמות התעבורה הינו חשוב אפילו כאשר מתבצע האימות.