An Empirical Study of Denial of Service Mitigation Techniques

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

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

Abstract

We present an empirical study of the resistance of several protocols to denial of service (DoS) attacks on client-server communication. We show that protocols that use authentication alone, e.g., IPSec, provide protection to some extent, but are still susceptible to DoS attacks, even when the network is not congested. In contrast, a protocol that uses a changing filtering identifier (FI) is usually immune to DoS attacks, as long as the network itself is not congested. This approach is called FI hopping. We build and experiment with two prototype implementations of FI hopping. One implementation is a modification of IPSec in a Linux kernel, and a second implementation comes as an NDIS hook driver on a Windows machine. We present results of experiments in which client-server communication is subject to a DoS-attack. Our measurements illustrate that FI hopping withstands severe DoS attacks without hampering the client-server communication. Moreover, our implementations show that FI hopping is simple, practical, and easy to deploy.

1 Introduction

Denial of service (DoS) attacks, in which an attacker attempts to deplete the target's resources, are common over the Internet [6]. 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, the 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 paper we are not concerned with the source of an attack, we simply use the term DoS to refer to either DoS or DDoS.

*Supported by the Israeli Ministry of Science.

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 [16]. 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 [13]. 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 paper, 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 [7] 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. Therefore, CPU exhaustion in the authentication phase constitutes a real DoS problem that needs to be addressed.

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 paper, 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 [4, 12]. Each packet contains a *filtering identifier* (FI), taken from a secret pseudorandom sequence, known only to the two communicating parties. Previous work proposed using a UDP [4] or TCP [12] port as the FI; in this paper, we use IPSec's SPI field or an application-added 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., 2 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 either a standard Linux IPSec implementation, or simply an authentication the server itself performs on each incoming packet (as done, e.g., in SSL). The ϕ-Hopper implementation presented here is a refinement of the ideas presented in [4, 12]. 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 in two variations: (1) by modifying a Linux kernel's IPSec [3] implementation, and (2) by inserting code in a Windows NDIS layer. Both IPSec and ϕ-Hopper use SHA-1 [15] as a pseudorandom function (PRF) [8] 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 [18], and these clients may behave unexpectedly, possibly overloading the server. Previous work on hopping as a way to resist DoS attacks used idealistic, impractical rate-limiters [4], or employed no rate-limiting at all [12].

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 [11] or a single RST packet [20]. 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 [4].

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 w.h.p., 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 generated by 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.

The paper proceeds as follows: Section 2 presents a detailed description of ϕ-Hopper's mechanisms and implementation. Section 3 provides extensive measurements of IPSec and ϕ-Hopper under DoS attacks. Section 4 discusses related work. Section 5 concludes.

2 ϕ-Hopper

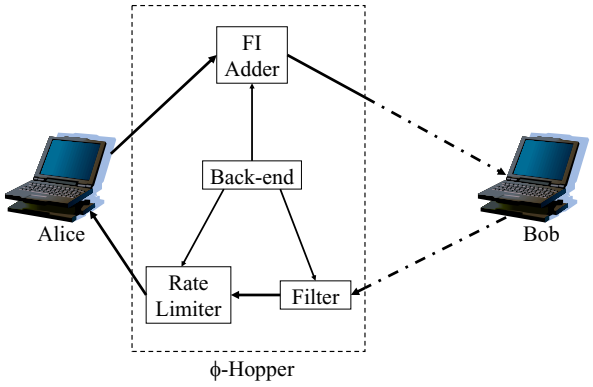


Figure 1. Communicating using ϕ-Hopper (Alice's view).

ϕ-Hopper leverages existing, cheap, network-level packet-filtering and rate-limiting solutions, along with more complex algorithms at a higher layer, which determine the filtering criteria and rate limits. Filtering is based on a *filtering identifier* (FI, or ϕ), which is some message field value that can be changed by the communicating parties, and is preserved en route. For example, it can be a combination of IP address and ports [12], as shown in [4], 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 1 shows the decomposition of ϕ-Hopper and the interaction between its various components.

The two parties wishing to communicate share a secret. This secret is used to create 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 [4]).

The generic nature of this protection scheme allows to use it to protect communication in different layers, e.g., by embedding the mechanism in IPSec, or in SSL/TLS. It is important to note that this simple idea of packet-filtering based on a changing packet field can be easily implemented in hardware. A hardware implementation of such a mechanism is naturally preferred, as it is faster and can provide better resilience to DoS attacks.

2.1 The front-end

The front-end can be a gateway or firewall, a layer in the end host's protocol stack, or even a dynamically programmable NIC that allows fast filtering at wire-speed [21]. 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. Messages arriving to a full queue are dropped. ϕ-Hopper sends messages from the queues to their destination in a RR fashion, provided that the total maximum allowed rate of messages is not exceeded. If a queue is empty, it is skipped for that RR cycle. This is very similar to Fair Queuing, which uses RR at the byte level [17]. 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.

2.2 The back-end

Figure 2 shows the pseudocode for ϕ-Hopper's back-end. Each party communicating via ϕ-Hopper maintains a *virtual time* (line 5), which determines its current position in the pseudorandom sequence for outgoing messages (lines 7 and 16), and for incoming messages (lines 8 and 17). Every fixed time interval δ, ϕ-Hopper performs a *hop* (line 13), which locally changes the virtual time (line 15). A ϕ-Hopper session between two parties is initialized using a seed that is used as the initial virtual time, and a shared secret key used for generating the pseudorandom sequence (line 4).

During session initialization, each party allocates bounded resources for communication in this session. ϕ-Hopper allocates separate resources for each active client (line 10), which are freed when the session for that client ends (line 22). Whenever a client becomes active/inactive, resources allocated to other clients might change, e.g., to achieve fairness or better utilization of the server. We note that, in general, since the server 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.

```
(1)  Initially:
(2)     ∀B  out(B) ← ⊥
(3)     ∀B  in(B) ← ⊥

(4)  initHopperSession(seed, key, B)
(5)     virt(B) ← seed
(6)     key(B) ← key
(7)     out(B) ← PRF_key(B)(virt(B)||A||B)
(8)     in(B) ← in(B) ∪ PRF_key(B)(virt(B)||B||A)
(9)     Set timer('close', B, virt(B)) to closeTimeout
(10)    Inform rate-limiter of 'initSession(B)'

(11) On wakeup of timer('close', B, virt)
(12)    in(B) ← in(B)\PRF_key(B)(virt||B||A))

(13) every δ time units
(14)    for all B s.t. out(B) ≠ ⊥ do
(15)       virt(B)++
(16)       out(B) ← PRF_key(B)(virt(B)||A||B)
(17)       in(B) ← in(B) ∪ PRF_key(B)(virt(B)||B||A))
(18)       Set timer('close', B, virt(B)) to closeTimeout

(19) endHopperSession(B)
(20)    out(B) ← ⊥
(21)    in(B) ← ⊥
(22)    Inform rate-limiter of 'endSession(B)'
```

Figure 2. ϕ-Hopper's back-end protocol for A (communicating with B).

We say that each party *opens FIs* for communication when these FIs are added to the list of valid FIs (lines 8 and 17), and *closes FIs* when these FIs are invalidated (line 12), $closeTimout$ time after they were created (lines 9 and 18). ϕ-Hopper uses two parameters that determine $closeTimeout$: Δ, the message latency, and Φ, representing the *synchronization gap* between the parties. Roughly speaking, the synchronization gap is the maximum differential between the times at which the parties decide to open the same FI in the pseudorandom sequence. It is the sum of the difference between the session starting time and the maximum clock drift during a ϕ-Hopper session. If the session time is so long that the clock drift might become a problem, i.e., Φ is too big, reinitialization is needed.

To compensate for the loose time synchronization between the parties, each party keeps multiple open FIs at the receiving end, corresponding to all virtual times the other

party might be in. The recipient opens a new FI every δ time, and closes a FI $4\Phi + \Delta$ time after opening it, i.e., $closeTimeout = 4\Phi + \Delta$. For example, if $\Delta = 100$ms, $\delta = 200$ms, and $\Phi = 250$ms, we get that there are at most 6 open FIs at a given time. A simple optimization that reduces the number of open FIs is closing a FI (if it is still open) Δ time after receiving a message on the next FI in the sequence.

3 Implementation and Measurements

3.1 Implementation

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

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

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

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

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

3.2 Measurements

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

Additionally, we measure the latency and the probability of delivery of a 100KB file over TCP, much like an FTP file transfer. The file delivery probability is the probability to successfully deliver the entire file. The latency is measured from the moment the first data packet is sent until the last acknowledgment is received. In all our results, each data point represents 100 experiments.

3.2.1 UDP/Linux

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

We compare 5 different scenarios: (1) The server has no DoS protection at all; (2) the gateways run IPSec in Authenticated Header (AH) mode, and the adversary knows the SPI used; (3) the gateways run IPSec in AH mode, and the adversary does not know the SPI used; (4) the gateways run IPSec in AH mode with ϕ-Hopper; and (5) the gateways run IPSec with ϕ-Hopper, but IPSec's authentication mechanisms are not used (only hopping). When attacking, the attacker sends bogus requests at a constant rate. In scenario (2), the bogus requests carry the correct SPI field, but fail authentication. In scenarios (3), (4) and (5), the bogus requests carry an incorrect (arbitrary) SPI field (w.h.p., for scenarios (4) and (5)), 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 exceeds the exposure delay $\mathcal{E}$, the adversary has ample time to discover the SPI, e.g., by ARP-poisoning a LAN, or by sniffing packets in intermediate routers. Once the adversary obtains the SPI, scenario (3) transforms into scenario (2). Since we assume relatively long sessions, we include scenario (3) mainly to quantify the overhead of port hopping. 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.

Figure 3(a) depicts the delivery probability as the attacker's strength increases. We see that ϕ-Hopper achieves the same delivery probability exhibited when the adversary does not know the SPI used, as filtering in these cases is based on a simple comparison of a header field. However, ϕ-Hopper does not rely on keeping the SPI, which is sent in the clear, secret. 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. ϕ-Hopper significantly outperforms IPSec when the SPI is compromised. The delivery probability is much lower when the SPI is known to the attacker, since this case requires complete authentication of every packet. This difference is most evident for relatively weak attacks ($80{,}000$ requests/sec), where ϕ-Hopper maintains 100% delivery, but the delivery for IPSec with a known SPI drops sharply to 44%. We can further see that having *any* form of protection is better than having no protection at all. When the server has no protection, it crashes even when the attack is very weak, reducing delivery probability to 0.

Figure 3(b) shows the effect of increasing-strength attacks on latency. In this experiment the server does not really process the request, but rather returns a reply immediately. We measure this parameter since we want to isolate the effect the algorithms run by the gateways have on latency. We can see that unless the SPI is known, the latency stays the same even when the attack strength increases. 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). Thus, as opposed to overlay networks, ϕ-Hopper ensures DoS-resilience with no or small penalty in latency. Conversely, when only IPSec is used and the SPI is known, the latency exhibited 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 3(c) displays the delivery probability under a bursty DoS attack, where bogus requests are not sent at constant intervals, but rather as bursts. The attack strength is measured as the average number of bogus requests per second. Comparing these results to Figure 3(a), we observe that a bursty attacker induces less damage than an attacker whose sending times are uniformly distributed over time. This can be explained by the fact that at times in which the attacker does not send any bogus message, the client's requests can be easily processed.

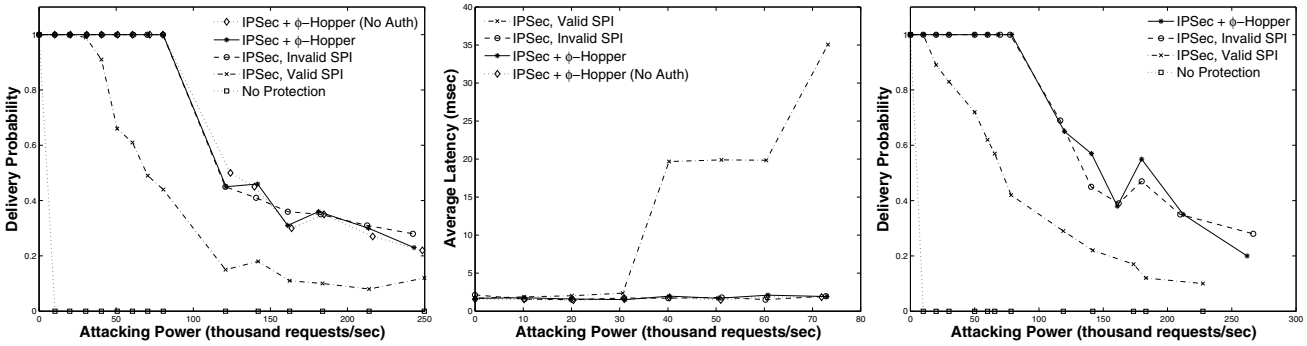
3.2.2 TCP/Linux

Having seen the benefits ϕ-Hopper offers for UDP traffic, we turn to test its effects on TCP traffic. We start by noting that using TCP with no IPSec protection is problematic for two reasons: First, if the adversary discovers or guesses the sequence numbers used in TCP, it can bring down the connection by sending a single RST packet [20]. We validated this in an experiment in which our attacker software sniffed a single packet sent by the client to the server over a TCP connection. The attacker then discovered the sequence numbers used in the TCP connection, and sent a single RST packet to the server with an appropriate sequence number. This packet brought down the connection immediately, as that is the purpose of an RST packet, when the sequence number falls inside the range of numbers the server expects.

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

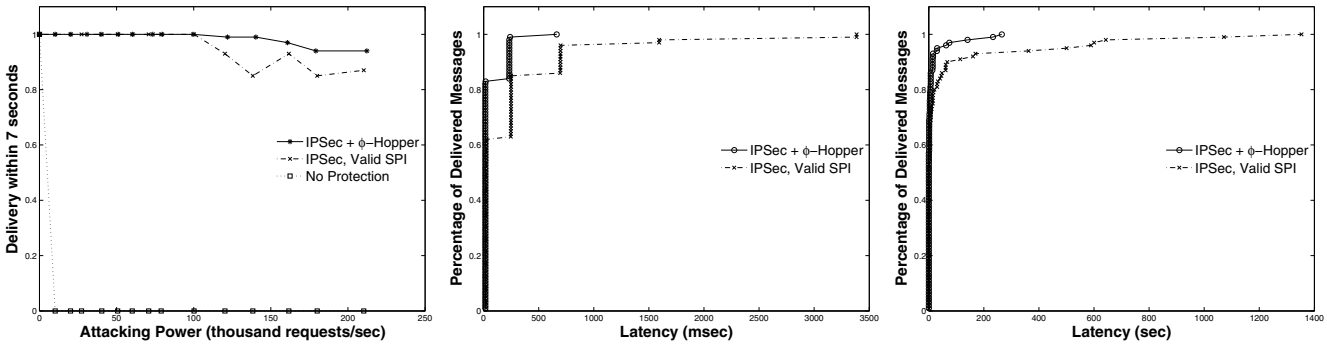
Figure 4(a) shows the delivery probability of TCP traffic over IPSec, with and without ϕ-Hopper. TCP's retransmission mechanism ensures that all messages eventually arrived to their destination. The figure shows the percentage of requests for which the client receives a response within 7 seconds of the moment the request was sent. As expected, when no protection is in use, the server crashes due to the heavy load. We can see that using ϕ-Hopper provides better delivery probability compared to IPSec with a compromised SPI, for attacks stronger than $100{,}000$ requests per second. For weaker attacks, all packets are delivered within 7 seconds in both scenarios.

Figure 4(b) shows the cumulative distribution function (CDF) of TCP latencies (RTT) for ϕ-Hopper and IPSec with a compromised SPI, for an attack power of $100{,}000$ requests per second. We can see that ϕ-Hopper provides better RTTs than IPSec with a compromised SPI. While over 80% of the messages passing through ϕ-Hopper had no latency penalty (cf. data point 0 in Figure 3(b)), IPSec managed to deliver only 60% of the messages with no delay. This corresponds to about 20% message loss in the first transmission when using ϕ-Hopper, compared to about 40% message loss in the first transmission for IPSec with a known SPI (cf. Figure 3(a).) Furthermore, ϕ-Hopper managed to deliver 99%



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

Figure 3. DoS attacks on IPSec on Linux, with and without ϕ-Hopper (UDP). ϕ-Hopper achieves the same results as IPSec with an invalid SPI, i.e., the attacker does not know the SPI value in use, *without* requiring the cleartext SPI to remain secret. When the attacker uses a valid SPI, IPSec only withstands attacks of less than half the strength before it is hampered by performing many authentication operations. When no defense is used, even very mild attacks cannot be tolerated.



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

Figure 4. DoS attacks on IPSec on Linux, with and without ϕ-Hopper (TCP).

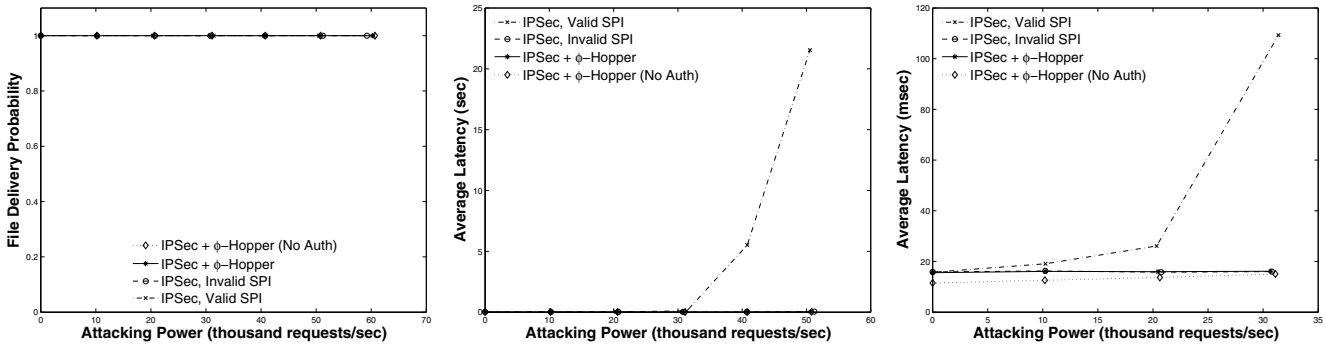
of the messages within 250 msecs, while IPSec delivered only about 82% of the messages by that time, and had overall delays of up to 3.5 seconds in delivery. We can clearly see TCP's exponential backoff in action, as delays get about 2 times longer for each retransmission.

Figure 4(c) depicts the CDF of TCP latencies for a stronger attack, of $240{,}000$ requests per second. Notice that the latency in the figure is given in *secs*, and not in msecs, as before. The figure clearly shows that ϕ-Hopper provides reasonable latency for 85% to 90% of the messages, while IPSec's latency starts deteriorating at about 75% to 80%. Moreover, the delivery of some messages in IPSec takes over 20 minutes – about 4.5 times worse than the longest delay in ϕ-Hopper.

Finally, 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 5 (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 5 (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 5 (c) is the first part of Figure 5 (b). The figure shows that even for milder attacks, IPSec with a known SPI entails a latency 5 times larger than the latency measured using the other protocols.

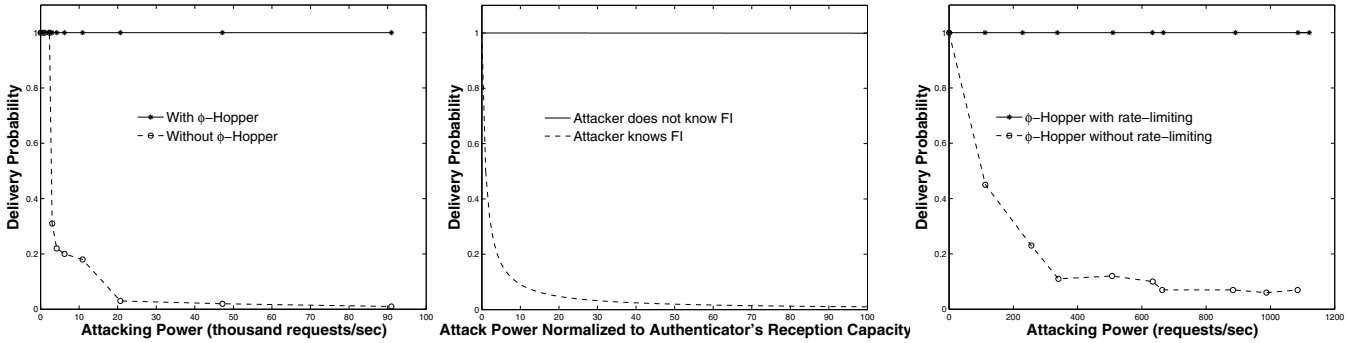
3.2.3 UDP/Windows

In our second setting, the client communicates directly with the web server, and we measure the effect ϕ-Hopper has when it runs on the server's machine, and not on a dedicated machine. The server runs on a Windows machine along



(a) Probability to complete transfer. (b) Latency of transfer. (c) Latency of transfer (zoomed in).

Figure 5. TCP 100KB file transfer over IPSec on Linux, with and without ϕ-Hopper. Again, ϕ-Hopper achieves the same performance under severe attacks as IPSec when the attacker uses an invalid SPI, without relying on the attacker's inability to discover the cleartext SPI. When the attacker discovers the SPI (uses a valid one), IPSec is burdened with cryptographic operations, and performance deteriorates significantly under heavy attacks.



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

Figure 6. Delivery probability under DoS attacks.

with ϕ-Hopper (in the appropriate experiments), which performs user-level filtering of packets through a kernel-level NDIS hook driver. The server machine has a Pentium 4 3.2GHz CPU, and 1GB of RAM.

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

3.2.4 Theoretical Values

We compare our results to analytical results for the delivery probability under DoS attacks, as taken from [4] (see Figure 6(b)). The theoretical analysis assumes the attacker's sending times are uniformly distributed, and thus the results shown in the figure can be compared to figures 3(a) and 6(a). Indeed, we can see that the actual measurements closely match the theoretical analysis.

3.2.5 Rate-Limiting

Figure 6(c) shows the effect of rate-limiting on the delivery probability for UDP traffic on IPSec/Linux. In this experiment, we have two clients: one valid client, and one compromised client. The valid client sends requests at a rate of 10 requests per second. The compromised client tries to load the server. We measure the delivery probability for the valid client as a function of the rate of requests sent by the compromised client. We can see that when rate-limiting is not enforced, the delivery probability drops rapidly due to the load on the server. Limiting the rate of each client to at most 12 requests per second suffices to ensure a delivery probability of 1.

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

Table 1 shows the difference in delivery probability and latency (RTT) for a client chosen arbitrarily from the 3 clients communicating with the server. We can see that, although RR rate-limiting imposes higher latency due to its periodic and cyclic nature, it handles bursty traffic much better than FQ rate-limiting. While the delivery probability drops down to 11% 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 a queue is empty, the other flows gain better maximum rates.

Our rate-limiting experiments show that, although rate-limiting in general is crucial to ensure a system's resilience to DoS attacks, it is important to choose the appropriate rate limiter. Our measurements show that using a round-robin rate-limiter that has more flexible quotas is superior to using a fixed-quota rate-limiter, when the clients send packets in bursts.

4 Related Work

Other empirical studies of DoS attacks include an empirical study of proxy networks [19] such as SOS [10] and Mayday [2]. This study 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 for direct client-server communication, without incurring a penalty on latency. Other work focuses on quantifying DoS activity over the Internet [14], while our focus is on DoS protection.

It has already been shown that DoS attacks can be harmful even when they are low-rate and do not congest the network [11, 13]. 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. For that, a simulation is not enough, and a thorough empirical study is needed [5]. 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 [9]. ϕ-Hopper is based on ideas that have been suggested in [4] and in [12]. However, these previous suggestions lacked in several areas, and so ϕ-Hopper differs from them in the following ways:

1. Instead of using the current time as the seed to the pseudorandom sequence, the initial seed used to start the sequence is given to the protocol and used as virtual time. Thus, there should only be means for the parties to share the seed (which is not secret), and no time synchronization between the communicating parties is needed, but rather a bounded clock drift.

2. ϕ-Hopper supports communication between many clients and a single server, and not just two-party communication.

3. ϕ-Hopper uses realistic rate-limiting techniques, as opposed to the purely theoretical analysis in [4] 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 [12] uses no rate-limiting at all.

4. ϕ-Hopper is implemented in two variations, and we provide measurements of the actual protocol implementation, and not of its simulated behavior [12] or of an analytical analysis of the protocol (as given in [4]).

The analysis in [4] shows that the basic idea of hopping is very effective against DoS attacks, but does so under simplified network and rate-limiting models. Other work *simulates* the effect port-hopping has on the delivery probability under attack, and shows that using it is expected to decrease the load on the server [12]. In Section 3 we have shown that the analysis in [4] gives a good estimate of realistic results, using a real implementation of all of ϕ-Hopper's components. Our results not only show that ϕ-Hopper provides strong resistance against DoS attacks, they also show that relying merely on authentication to provide DoS protection is futile.

5 Conclusions

We performed an empirical evaluation of two techniques that mitigate the effects of DoS attacks on client-server

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

Table 1. Fixed-quota vs. RR rate-limiting.

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.

References

[1] B. Adoba and W. Dixon. RFC 3715 – IPSec-network address translation (NAT) compatibility requirements, March 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. RFC 2401, IETF, 1998.

[4] G. Badishi, A. Herzberg, and I. Keidar. Keeping denial-of-service attackers in the dark. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 4(3):191–204, July–September 2007.

[5] 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)*, March 2006.

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

[7] L. Garber. Denial-of-service attacks rip the Internet. *Computer*, 33(4):12–17, 2000.

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

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

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

[11] A. Kuzmanovic and E. W. Knightly. Low-rate tcp-targeted denial of service attacks (the shrew vs. the mice and elephants). In *SIGCOMM*, 2003.

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

[13] 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)*, pages 1–12, August 2006.

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

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

[16] Riverhead Networks (Cisco). Products overview.

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

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

[19] J. Wang, X. Liu, and A. A. Chien. Empirical study of tolerating denial-of-service attacks with a proxy network. In *The 14th conference on USENIX Security Symposium*, pages 51–64, 2005.

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

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