Denial of Service Protection with Beaver

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

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

Abstract

We present Beaver, a method and architecture to "build dams" to protect servers from Denial of Service (DoS) attacks. Beaver allows efficient filtering of DoS traffic using low-cost, high-performance, readily-available packet filtering mechanisms. Beaver improves on previous solutions by not requiring cryptographic processing of messages, allowing the use of efficient routing (avoiding overlays), and establishing keys and state as needed. We present two prototype implementations of Beaver, one as part of IPSec in a Linux kernel, and a second as an NDIS hook driver on a Windows machine. Preliminary measurements illustrate that Beaver withstands severe DoS attacks without hampering the client-server communication. Moreover, Beaver is simple and easy to deploy.

1 Introduction

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

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

The first step in providing DoS resistance is protecting the network layer. Readily-available DoS solutions deployed in firewalls or gateways typically use two methods: filtering according to packet header fields like addresses and ports, and rate-limiting traffic. These simple methods are very efficient, but are insufficient: header fields can be spoofed to match filtering criteria, and rate-limiting, while good for the network, is of little use to the application, as valid packets are indiscriminately discarded. Our measurements show that even when the network is not loaded, a large number of bogus requests can kill a server that does not require authentication, and can virtually drop to zero the reliability of client-server communication when the server does require authenticated requests. Thus, network-level protection needs to be complemented by application-level mechanisms.

Our goal is therefore to provide simple and practical DoS protection for the application layer, using readily-available network-level solutions. We present *Beaver* – a robust architecture and method for application-level DoS protection. Beaver's components can be implemented in several different ways, depending on the required deployment scenario and capabilities. We describe an initial implementation of our system in two variations: (1) by modifying a Linux kernel's IPSec [2] implementation, and (2) by inserting code in a Windows NDIS layer.

The effectiveness of a DoS-mitigation solution can be quantified through mathematical analysis and empirical results. These two methods complement each other, as the analysis can provide results for *all* possible attacks, but these results are only applicable for a *model* of the system, which may or may not correctly reflect the real world. In this paper, we focus on the latter. Our initial findings validate earlier theoretical results [3].

2 Design Goals

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

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

*Supported by the Israeli Ministry of Science.

tions. DoS attacks on these services may degrade the service so much that clients lose money due to its unavailability.

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

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

One might argue that authenticating client-server communication alone is enough to filter out invalid packets sent by DoS attackers. But although authentication is enough to discriminate bogus messages from valid ones, the validation itself is costly. This is especially a problem if the server is the one performing the validation, as happens when using SSL. Since the server should be mainly busy with answering requests, we would like to minimize the number of invalid packets that reach the server and cause extra processing.

3 Beaver's Architecture

We present *Beaver* – a robust architecture and method to protect servers from DoS attacks. Beaver employs two DoS-protection mechanisms: one for registration and admission of new client sessions, and another for protecting ongoing sessions. The former uses dedicated *admission servers* (ADMs). The latter is *ϕ-Hopper* – a two-party communication protocol that mitigates DoS attacks by avoiding the use of fixed packet-filtering criteria (e.g., IP addresses and ports) at a firewall or gateway, and performing "pseudorandom hopping" on the filtering criteria instead [3, 7].

The ADMs can be provided as a common Internet service to multiple legacy servers, and therefore, they are not all trusted. The use of ADMs takes the registration and admission load off the server, so that the server is not concerned with DoS attacks on clients trying to be admitted into the system. This approach resembles the use of overlay networks in SOS [6] and Mayday [1]. However, SOS and Mayday also screen DoS attacks by hiding the server's identity and making it known only to a few nodes in the overlay network. Thus, in these solutions, all client messages, including those for ongoing sessions, are routed through the overlay, causing the latency of the client-server communication to increase by a factor of 5 or even 10 [6]. Additionally, this is a form of security-by-obscurity. Once the filtering criteria are revealed, spoofed packets that match the server's filtering criteria can penetrate the system's defenses and reach the server. Another drawback of Mayday and SOS is that overlay networks are more complex to set up and update.

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

Together, the ADMs and ϕ-Hopper are very effective against DoS attacks. In Section 6 we give experimental results for two implementations of ϕ-Hopper, and show that an ongoing session is almost unaffected by severe DoS attacks. These measurements validate a formal analysis of port-hopping and its guarantees given in previous work [3]. In the full paper [4], we give an analysis of the entire Beaver architecture and its guarantees, proving, among other things, that even when a heavy DoS attack is launched, clients can still start new sessions with the server, and that compromised clients cannot severly harm the server's performance.

4 ϕ-Hopper

Beaver's two-party communication component, ϕ-Hopper, leverages existing, cheap, network-level packet-filtering solutions, along with more complex algorithms at a higher layer, which determine the filtering criteria. At each communication party, ϕ-Hopper has two parts: a *front-end* that performs fast packet-filtering, and a *back-end* that controls the front-end's filtering criteria. The front-end can be a gateway or firewall, or a layer in the end host's protocol stack. Filtering is based on a *filtering identifier* (FI, or ϕ), which is some message field value that can be changed by the communicating parties, and is preserved en route. For example, it can be a combination of IP address and ports [3, 7], or IPSec's security parameter index (SPI) field [2]. The FI can also be an artificial field appended to the message. The FI size can be set to meet the wanted DoS-resistance guarantees.

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. The second implementation installs the front-end on the communication endpoints as an NDIS hook driver on a Windows system and, checks packets for an appended 160-bit FI. In this case, the hook only filters packets, and authentication is performed by the server.

The two parties wishing to communicate share a secret.

This secret is used to create pseudorandom sequences of FIs. Each message transmitted between the parties carries a FI taken from an appropriate pseudorandom sequence. The receiver's front-end anticipates the FI according to the pseudorandom sequence, and filters out all messages carrying invalid FIs. The FIs change in order to maintain DoS-resilience. Otherwise, the adversary could eavesdrop on messages and discover the FI in use. Hopping using an appropriate FI size ensures that with high probability, the adversary cannot discover the FI [3]. This scheme has some similarities with the S/KEY authentication method [5].

Each party communicating via ϕ-Hopper maintains a *virtual time*, which determines its current position in the pseudorandom sequence. Every fixed time interval, Beaver performs a *hop*, which locally changes the virtual time. A ϕ-Hopper session between two parties is initialized using a seed that is used as the initial virtual time, and a shared secret used for generating the pseudorandom sequence. In Beaver, the former is provided at admission time (by an ADM), and the latter – during the registration process.

During initialization, each party allocates bounded resources for communication in this session. ϕ-Hopper allocates separate resources for each active FI. We say that each party *opens FIs* for communication when resources are allocated for them, and *closes FIs* when these resources are freed. Changing the FIs (when the virtual time advances) closes old FIs and opens new ones, and the resources are effectively shifted between them. The synchronization between the two parties is loose, since they do not simultaneously receive the seed from the ADM, and their clocks may drift. Therefore, each party keeps multiple open FIs at the receiving end, corresponding to all virtual times the other party might be in.

When we have multiple on-going sessions from different clients, it may be probabilistically possible that two or more clients use the same FI at a specific point in time. To deal with this low probability case, we maintain a list of all clients using each FI, and adjust that FI's resources accordingly. We note that, in general, since resources are bounded and separated between FIs, 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.

Our implementations use a shared secret of 160 bit. At each hop, we increase the virtual time by 1, and calculate the 160-bit SHA-1 hash of the current virtual time concatenated with the shared secret. The virtual time is floored according to the time granularity that we wish to use. If the required FI's length is shorter than 160 bits, we calculate the xor of the bytes in chunks that have the same size as the FI.

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 uses the ideas presented in [3, 7], but differs in the following ways [4]:

1. Instead of using the current time as the seed to the pseudorandom sequence, the initial seed used to start the sequence is decided on during the admission process, and is used as virtual time. Thus, no time synchronization between the communicating parties is needed, but rather a bounded clock drift. If the session time is so large that the clock drift might become a problem, reinitialization should occur.
2. We do not assume pre-shared keys, and provide a key distribution mechanism. The shared secret is distributed during the registration process.
3. ϕ-Hopper supports communication between many clients and a single server, and not just two-party communication.
4. ϕ-Hopper is implemented in two variations, and we provide measurements of the actual protocol implementation, and not of its simulated behavior.

Our full paper contains the pseudocode of ϕ-Hopper [4]. The analysis we performed in previous work [3] shows that ϕ-Hopper has a tremendous effect on the probability of receiving valid packet while under a DoS attack. Additionally, it shows that port-hopping, as done in ϕ-Hopper, maintains high delivery probability even for an attacker $10{,}000$ times stronger than the two communicating parties, and for FIs as short as 16 bits, e.g., TCP/UDP port numbers. 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 [7].

5 Admission Servers

The ADM has two roles. First, it allows clients to *register* to the service, whereby the client receives a shared secret with the ADM and with the server. The ADMs do *not* know the latter, as it is encrypted with the client's public key. Different clients have different secrets, and the same client may have different secrets with multiple ADMs. In order to register, the client provides a *certificate* from some trusted authority, that binds the client's public key to the client's identity. This certificate can be based on authentication as complex as a biometrics match, or as simple as a credit card number, as long as it is hard for the same client to obtain many valid identities.

Second, the ADM authenticates registered clients before authorizing them to communicate with the server. The latter is called *the admission process*. There may be multiple admission servers, and all of them are identical, except for a unique secret each of them shares with the server. The use of many admission servers protects the admission process from

DoS attacks, as the client can initiate the admission process with an arbitrary ADM. A DoS-attacker that wishes to severly harm the admission process needs to launch a massive attack that targets most, if not all of the ADMs. This idea is very similar to the one used for SOS SOAPs [6], and it can be employed here since replicating an ADM is cheap and easy, as opposed to, say, replicating the server.

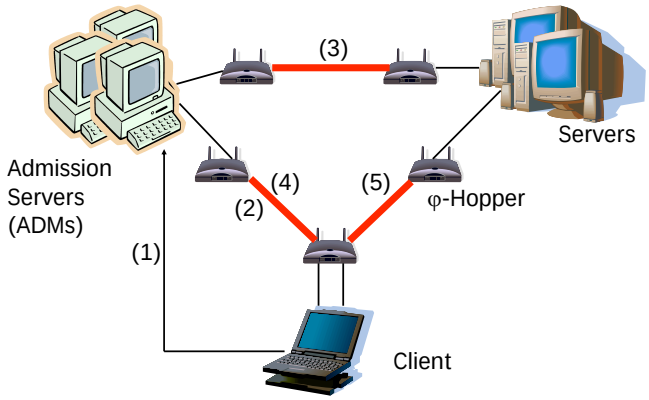


Figure 1: Beaver's architecture, where ϕ-Hopper operates in tunnel mode (marked in bold lines).

Figure 1 illustrates Beaver's architecture, and shows the admission process in action: (1) A pre-registered client requests an ADM to start a new session with the server. The client can choose the ADM arbitrarily. Specifically, a client that fails to start a session through some ADM may choose a different ADM for the admission process. (2) The ADM communicates with the client via ϕ-Hopper and authenticates the client. Communication via ϕ-Hopper is marked in bold lines. The figure illustrates ϕ-Hopper in tunnel mode, i.e., hopping between gateways. (3) The ADM contacts the server through a constant ϕ-Hopper session that they share, and asks it to start a new session with the client. The server then opens FIs for the new session with the client. (4) The ADM notifies the client that it can start communicating with the server. (5) The client communicates with the server via ϕ-Hopper. More generally, there can be multiple servers (e.g., a server farm), and an ADM can direct the client to any one of them.

Our full paper [4] contains pseudo-code for all of Beaver's procedures, including the admission and registration processes. We further prove the following claims:

- If no ADM is malicious, then no server ever allocates resources for communicating with invalid clients.
- The number of open FIs the servers allocate for invalid connections is bounded.
- A DoS attack on the ADMs' incoming link from a server does not prevent a valid client from starting a session with that server.
- The extra load induced on a server due to an attack increases on average by at most $\frac{C}{2^{\ell-2}}$ messages per second, where C is the attacker's capacity in messages per second, and ℓ is the number of bits in a FI.
- If no ADM is malicious, a compromised valid client that does not impersonate other valid clients cannot load the server with more messages per second than the server rate-limits each session.

6 Preliminary Measurements

Our implementation of Beaver currently contains only the ϕ-Hopper component, in two different forms. To allow proper operation of ϕ-Hopper, we fix the secret shared between the client and the server, and fix the seed of the pseudorandom sequence for ϕ-Hopper.

We measure the effect authentication and hopping have on the resistance to DoS. We experiment with a standard HTTP server, an appropriate client, and an adversary (implemented using one to three machines). In each experiment, the adversary sends bogus requests at an average constant rate to the web server. At the same time, the client sends valid requests to the server. The server processes each request, and dynamically forms a response, consuming CPU power while doing that. We measure the latency, consisting of round-trip time and server processing time, and delivery probability, i.e., the probability that a client's valid request is processed by the web server, as a function of the attacker's strength. Each data point represents 100 experiments.

Most of our experiments consider authenticated communication. For completeness, we also measured the effect of an attack when no authentication is performed, i.e., when the server processes all of the attacker's requests. In this case, even for a small attacking power, the server crashed, rendering any comparison to Beaver irrelevant.

In our first setting, we measure the advantages ϕ-Hopper offers, as compared to IPSec [2], when deployed on gateways. For this setting, we have a client, connected to gateway A, where gateway A is connected to gateway B, which in turn is connected to a web server. The gateways run Linux with IPSec in tunnel mode, with or without ϕ-Hopper installed, according to the experiment. The gateway machines have a Pentium 3 650MHz CPU, and 256MB of RAM.

We compare 3 different scenarios: (1) the gateways run IPSec in Authenticated Header (AH) mode, and the adversary knows the SPI used; (2) the gateways run IPSec in AH mode, and the adversary does not know the SPI used; and (3) the gateways run IPSec in AH mode with ϕ-Hopper. When attacking, the attacker sends bogus requests at a constant rate. In scenario (1), the bogus requests carry the correct SPI field, but fail authentication. In scenarios (2) and (3), the bogus requests carry an incorrect (arbitrary) SPI field (w.h.p., for scenario (3)), and so the bogus requests do not reach the authentication phase. It is important to note that scenario (2) is not realistic for long sessions, as the SPI stays constant, and the adversary has ample time to discover it, e.g.,

by ARP-poisoning a LAN, or by sniffing packets in intermediate routers. Thus, scenario (2) is only used to quantify the overhead of port hopping.

Figure 2(a) depicts the delivery probability as the attacker's strength increases. We see that ϕ-Hopper achieves the same delivery probability exhibited when the adversary does not know the SPI used, as filtering in these cases is based on a simple comparison of a header field. The delivery probability is much lower when the SPI is known to the attacker, since this case requires complete authentication of every packet. This difference is most evident for relatively weak attacks ($80{,}000$ requests/sec), where ϕ-Hopper maintains 100% delivery, but the delivery for IPSec with a known SPI drops sharply to 44%.

Figure 2(b) shows the effect of increasing-strength attacks on latency. We can see that the latency stays the same even when the attack strength increases, and that all 3 scenarios exhibit the same latency. This is also the same latency measured when IPSec and Hopper do not run at all (not shown on graph). Thus, as opposed to overlay networks, ϕ-Hopper ensures DoS-resilience with no penalty in latency.

Figure 2(c) displays the delivery probability under a bursty DoS attack, where bogus requests are not sent at constant periodic intervals, but rather at bursts. The attack strength is measured as the average number of bogus requests per second. Comparing these results to Figure 2(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. However, more experiments are needed in order to fully understand this phenomenon.

In our second setting, the client communicates directly with the web server, and we measure the effect ϕ-Hopper has when it runs on the server's machine, and not on a dedicated machine. The server runs on a Windows machine along with ϕ-Hopper (in the appropriate experiments), which performs user-level filtering of packets through a kernel-level NDIS hook driver. The server machine has a Pentium 4 3.2GHz CPU, and 1GB of RAM. Since now there is no IPSec layer to authenticate packets, the server itself authenticates requests using a simple SHA-1 hash attached to valid messages.

Figure 3(a) shows the delivery probability with and without ϕ-Hopper. We can see that 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.

Finally, we compare our results to analytical results for the delivery probability under DoS attacks, as taken from our previous work [3] (see Figure 3(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 2(a) and 3(a). Indeed, we can see that the actual measurements closely match the theoretical analysis.

7 Conclusions

We presented Beaver, a method and architecture to protect applications from DoS attacks. Our full paper [4] formally proves Beaver's good properties in withstanding DoS attacks. Preliminary measurements presented in this paper show that indeed Beaver is a promising solution. We are working to perform additional measurements and analysis of Beaver, to better understand phenomena such as the effect of attack burstiness on the delivery probability, and more generally, to understand Beaver's behavior under a wider range of attack strategies. We would like to explore the challenge of efficiently and scalably supporting many clients, while resisting DoS by authenticated clients. It would also be interesting to measure the effect of DoS attacks on the server's throughput. Finally, we intend to implement the admission servers and operate Beaver as a complete system.

References

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

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

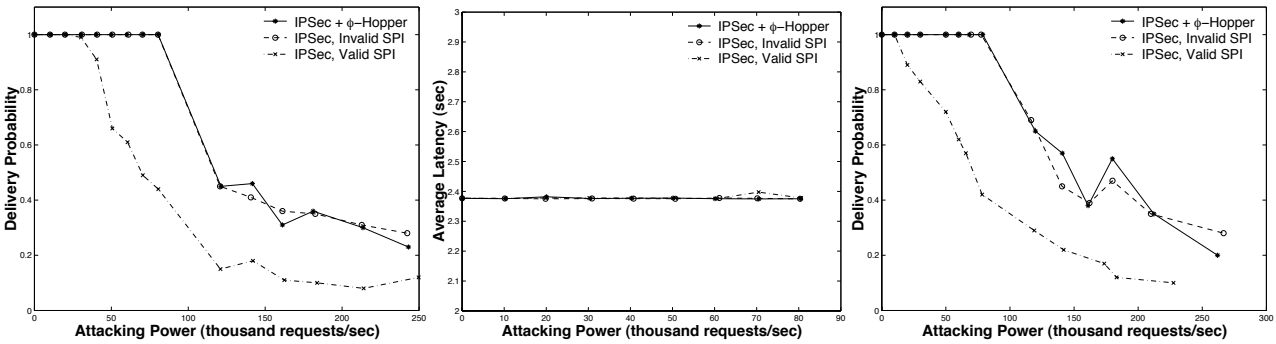
[3] G. Badishi, A. Herzberg, and I. Keidar. Keeping denial-of-service attackers in the dark. In *DISC*, volume 3724 of *LNCS*, pages 18–32, September 2005.

[4] G. Badishi, A. Herzberg, and I. Keidar. Denial of service? leave it to Beaver. TR CCIT 595, Department of Electrical Engineering, Technion, July 2006.

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

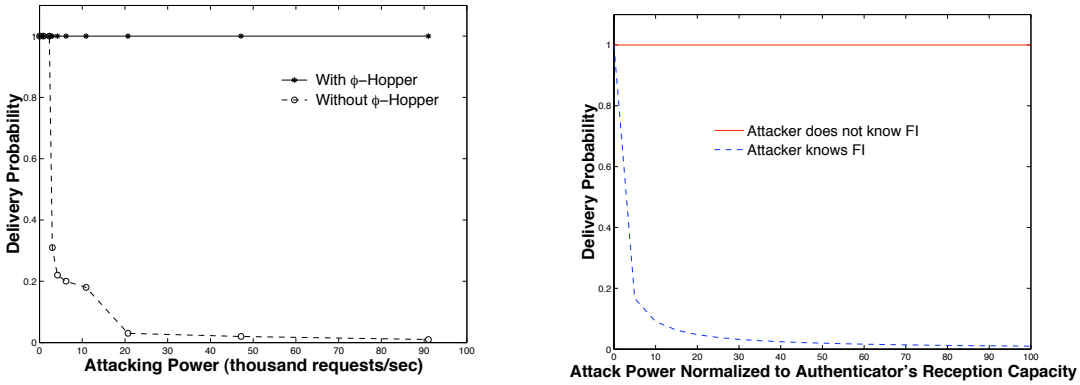
[6] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: An architecture for mitigating DDoS attacks. *JSAC*, 21(1):176–188, 2004.

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



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

Figure 2: DoS attacks on IPSec on Linux, with and without ϕ-Hopper.



(a) Windows NDIS layer. (b) Theoretical values.

Figure 3: Delivery probability under DoS attacks, with and without ϕ-Hopper.