Roaming Honeypots for Mitigating Service-level Denial-of-Service Attacks*

Sherif M. Khattab§ Chatree Sangpachatanaruk‡ Daniel Mossé§ Rami Melhem§ Taieb Znati§‡
§Department of Computer Science
‡Department of Information Science and Telecommunications
University of Pittsburgh, PA 15260
{skhattab, chatree, mosse, melhem, znati}@cs.pitt.edu

Abstract

Honeypots *have been proposed to act as traps for malicious attackers. However, because of their deployment at fixed (thus detectable) locations and on machines other than the ones they are supposed to protect, honeypots can be avoided by sophisticated attacks. We propose* roaming honeypots*, a mechanism that allows the locations of honeypots to be unpredictable, continuously changing, and disguised within a server pool. A (continuously changing) subset of the servers is active and providing service, while the rest of the server pool is idle and acting as honeypots.*

We utilize our roaming honeypots scheme to mitigate the effects of service-level *DoS attacks, in which many attack machines acquire service from a victim server at a high rate, against back-end servers of private services. The roaming honeypots scheme detects and filters attack traffic from outside a firewall (external attacks), and also mitigates attacks from behind a firewall (internal attacks) by dropping all connections when a server switches from acting as a honeypot into being active. Through ns-2 simulations, we show the effectiveness of our roaming honeypots scheme. In particular, against external attacks, our roaming honeypots scheme provides service response time that is independent of attack load for a fixed number of attack machines.*

1. Introduction

Network Denial-of-Service (DoS) attacks [26] pose an increasing threat to both public services, such as Google, and private services, such as subscription-based business services, deployed over the Internet. Typical private services consist of front-ends (e.g., web servers) and back-ends (e.g., database servers). Whereas service front-ends can be protected from DoS attacks by massive replication, as in Content Distribution Networks (CDNs) such as Akamai [1], service back-ends cannot tolerate the same level of replication, because of higher costs and tighter consistency constraints.

DoS attacks are difficult to prevent because of inevitable software vulnerabilities, which get exploited by attackers either to directly crash a victim or to compromise *zombie* machines, which are unwittingly used to launch the attack. Network-level DoS attacks aim at congesting network resources, such as link capacity and router buffers, by flooding them with bogus packets sometimes with spoofed (forged) source addresses. However, wide deployment of source-end DoS defense systems, such as *D-WARD* [19], which autonomously detects and stops abnormal one-way flows, and *Ingress Filtering* [9], which stops most spoofed attacks, would limit the pervasiveness and effectiveness of network-level and spoofed attacks, leaving floor to *service-level* DoS attacks. In service-level DoS attacks, a large number of attack machines manage to acquire service from a victim server, consuming both service-level resources, such as server memory and processing time, as well as network-level resources along the path outward from the server.

Honeypots [22,33], a proactive detection mechanism, are machines that are not supposed to receive any legitimate traffic and, thus, any traffic destined to a honeypot is most probably an ongoing attack and can be analyzed to reveal vulnerabilities targeted by attackers. Coupled with an Intrusion Detection System (IDS) (e.g., [4]), honeypots are effective in detecting hosts exploited by Internet worms [28] that perform random scanning [16]. However, since honeypots are deployed at fixed, detectable locations and on machines different than the ones they are supposed to protect, sophisticated attacks can avoid the honeypots.

In this paper, we propose *roaming honeypots*, a scheme for mitigating service-level DoS attacks against back-ends of private services. The locations of honeypots are continuously and unpredictably changing disguisedly within a pool of back-end servers. Each server alternates between providing the service and acting as a honeypot in a manner unpredictable to attackers.

* The authors were supported in part by NSF under grant ANI-0087609.

Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)
1063-6927/04 $20.00 © 2004 IEEE

1.1. Solution Approach

In [14], we presented the *proactive server roaming* mechanism, which is a secure and light-weight mechanism to proactively change the location of the active server within a server pool. Legitimate clients keep track of roaming times and location of the roaming server using light-weight, one-way hash functions.

In this paper, we build on the work in [14] to achieve roaming honeypots as follows: A (continuously and unpredictably changing) subset of k out of N servers is active and providing service, while the rest are acting as honeypots. Even if attackers obtain the identities of all servers, which of the servers are the active servers and which are the honeypots at a given moment is unknown to attackers.

Against service-level attacks, the benefit of our roaming honeypots scheme is two-fold: Firstly, idle servers (honeypots) detect attacker addresses so that all their subsequent requests are filtered out. Secondly, each time a server switches from idle to active, it drops all its current (attack) connections, opening a window of opportunity for legitimate requests before the attack re-builds up. We call these two benefits the *filtering effect* and the *connection-dropping effect*, respectively. Whereas the filtering effect defends the service against attacks launched from outside a firewall (external attacks), the connection-dropping effect mitigates attacks launched from behind the firewall (internal attacks).

The proactive nature of our scheme allows it to detect DoS attacks that are difficult to detect otherwise, for example *degrading attacks* [18], which consume only some portion of the victim's resources, and thus, could remain undetected for a significant time period. Although in this paper we focus on *physically* roaming honeypots, we note the potential of *logically* roaming honeypots (similar to IP hopping [11]), in which legitimate clients, in coordination with servers, randomly change the value of some field in their packets (e.g., destination address) rendering traffic with different field values illegitimate. Logical roaming increases the elusiveness and cost-effectiveness of the roaming honeypots, however, physical roaming is still necessary to defend against internal attacks. We leave the evaluation of logical roaming for future work.

We also utilize a layer of indirection between the server pool and legitimate clients, in the form of a network overlay of access gateways, to serve as front-ends, to make the mechanism transparent to clients, and to decouple authentication and authorization from service provision.

Through ns-2 simulation, we show the effectiveness of our roaming honeypots scheme against service-level DoS attacks launched from both external and internal networks. Against external attacks, our roaming honeypots scheme experiences a favorable attack-load-independent behavior, for a fixed number of attack machines. However, because of sacrificing some servers to act as honeypots, distributing the load on all the servers outperforms our roaming honeypots scheme in the case of a high legitimate client load combined with a low attack load.

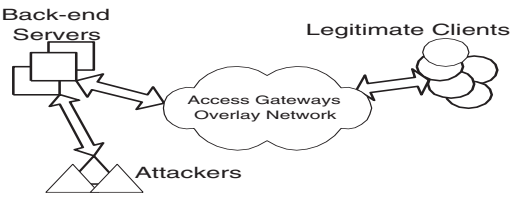

Figure 1. The service consists of a pool of back-end servers, an access gateways network (AGN) overlay acting as service front-end, and clients. Effective attackers send traffic directly to the servers.

In the next section, we describe the service and attack models. Section 3 reviews the proactive server roaming scheme and some of its limitations. In Section 4, we describe our roaming honeypots scheme and address some of the limitations of the proactive server roaming scheme. Section 5 presents an evaluation of our scheme against service-level DoS attacks. Section 6 expands on the overlay network, and discusses network-level attacks and overhead. We describe strongly related work in Section 7 and Section 8 concludes the paper.

2. Models

2.1. Service Model

We consider a private service that is deployed over the Internet and composed of front-ends, such as web servers, and back-ends, such as database servers. In this paper, we focus on protecting a pool of N back-end servers, referred to as just servers in the rest of the paper. The service is subscription-based; that is, clients need to subscribe through front-ends to gain access to the service. We assume a large client population with different access rights and both high service request rate and high membership subscription/expiration rate.

At the Point of Presence (PoP) of each ISP network hosting a server, a packet-filtering firewall and an on-line IDS, such as Snort [4], are deployed. The IDSs detect attack signatures and update the firewalls' filtering rules through a secure group communication mechanism (e.g., [34]).

Service front-ends form a layer of indirection between the server pool and legitimate clients in the form of a network overlay of access gateways (AGs), as depicted in Figure 1. We use the overlay network paradigm to provide flexibility in structuring the AGs (see Section 6.1), but not to

route data packets. The access gateways network (AGN) keeps track of the current active servers using the mechanism described in Section 4. Clients contact AGs to subscribe and request service, and after a client request is authenticated and authorized, AGs either contact the servers on behalf of the requesting client or tunnel authenticated client packets *to and from* the current active servers. The AGN can be operated by a third party entity in a business model similar to Akamai [1] or XenoService [36]. Servers do not perform per-request authentication/authorization and thus are relieved from costly authentication processing.

Finally, the service supports per-connection load balancing; AGs distribute clients' requests among servers using a distributed, dynamic load-balancing scheme.

2.2. Attack Model

In this paper, we assume that attackers select a number of servers, N_{att}, from the server pool and request service from the selected servers at a high rate.

We consider two types of attacks: *fixed-target* and *follower* attacks. In a fixed-target attack, the attacker selects N_{att} servers and attacks them continuously, whereas in a follower attack, the attacker tries to continuously direct the attack into N_{att} active servers. However, it takes some time (*follow delay*) for the attacker to detect that a server becomes idle, to find an active server, and to re-task its zombies to attack the active server.

Whereas public services are clearly vulnerable to service-level attacks, service-level attacks are also possible in our private service model. Although restricting service access to members of a list of current legitimate clients would eliminate service-level attacks, with a large client population and high join/leave and service request rates, maintenance of an up-to-date list of legitimate clients and per-request service authentication/authorization would incur high overhead causing both service degradation and attack amplification.

Defending against the service-level DoS attack is important as we envision this type of attack to be prevailing in the future after deploying source-end defenses, such as D-WARD [19], that filter out one-way flooding attacks at source networks. Also, service-level attacks are attractive to attackers because they are low-bandwidth attacks and can result in consuming both server and network resources.

It is difficult to launch a service-level attack using a spoofed source address; for instance an attacker needs to complete a three-way handshake protocol in order to request a TCP service. Thus, we assume service-level attacks use non-spoofed source addresses (we describe how to handle spoofed attacks in Section 6.2).

Finally, an attacker can eavesdrop on client traffic but cannot gather the identities of the current active servers by eavesdropping *within* the AGN or *at* the servers.

3. Proactive Server Roaming Background

The proactive server roaming scheme has been proposed in [14], where a prototype of the scheme is presented, and has been studied through simulation in [25]. In [14], one active server constantly changes its location within a pool of N homogeneous servers to proactively defend against unpredictable and undetectable attacks. Service time is divided into *epochs*; at the end of each epoch, the service migrates from one server to another in the server pool. A long hash chain is generated using a one-way hash function $H(\cdot)$, and used in a backward fashion similar to the *PayWord* scheme [24]. The last key in the chain, K_n, is randomly generated and each key, K_i $(0 < i < n)$, in the chain is computed as $H(K_{i+1})$ and used to calculate both the length, R_i, of service epoch E_i and the location, S_i, of the active server during E_i as follows: $R_i = \text{MSB}_m(H'(K_i))$ and $S_i = \textit{servers}[\text{MSB}_{\lfloor \lg N \rfloor}(H''(K_i))]$, where $\text{MSB}_j(x)$ are the j most significant bits of x, 2^m represents an upper bound on epoch length, N is the number of servers, and the array *servers* contains an ⟨IP address, TCP port⟩ pair for each server in the server pool. H' and H'' are public one-way hash functions, such as MD5 [23].

During *offline* subscription, each legitimate client is assigned a roaming key, K_t, from the hash chain, with a varying value of t according to each client's trust level and/or other policies. K_t allows the client to track the service up to and including epoch E_t. Clients also receive the *servers* list encrypted with a shared key, K_C, between the service and the client. The service registers the IP address of each subscribed legitimate client, a TCP port number for roaming update messages, the shared secret key K_C, and the index t of the roaming key assigned to each client.

The mechanism defines *roaming update* messages to serve three purposes: (1) to allow for changing the epoch length's upper bound, 2^m, to reflect the current threat level for example; (2) to keep the legitimate clients aware of the current service epoch; and (3) to provide a means for *online* extension of subscription durations of trusted clients; that is, to provide trusted clients with newer key values allowing them to track the service for a longer time.

TCP connection migration [27, 31] is used as a vehicle for the server roaming scheme. The proactive server roaming scheme proposes to use the clients for storing periodic state checkpoints of both TCP state and per-connection application state of the server. State update messages (containing state checkpoints) are sent securely to clients to protect from an attacker trying to fake erroneous state updates.

As a consequence of loose clock synchronization (i.e., the clock shift among system components is bounded by a constant, δ), each service epoch starts earlier by δ at the

new server and its firewall and ends later by $\delta + \gamma$ at the old server and its firewall, where γ is the estimated communication delay from clients to servers.

Although proactive server roaming represented a good first step toward the mitigation of DoS attacks, its limitations are addressed in this paper: (1) it handles only one server active at a time; (2) it requires offline service subscription, which is not a flexible service model; (3) servers keep track of the IP addresses of all subscribed clients in order to periodically send them roaming update messages; keeping such a list is not scalable, especially with a large and dynamic client population, and limits each client to use a fixed IP address, reducing flexibility; (4) the mechanism is not transparent to the clients and requires changes in client software; (5) it is easy to compromise a client machine with the possibility of either revealing the service secrets stored in the client or eavesdropping on client's traffic to discover the address of the current server (or both).

4. Roaming Honeypots

Our roaming honeypots mechanism allows for k out of N servers to be concurrently active. The locations of the current active servers (and thus the honeypots) are changed so as to be unpredictable to the attackers. Each legitimate request is tunneled through the AGN to an active server, whereas an illegitimate request sent directly to a randomly selected server has a probability of $\frac{N-k}{N}$ of hitting a honeypot. The source address of any request that hits a honeypot is recorded and all its future requests are dropped.

An idle server (honeypot) responds, in a contained manner, to (malicious) packets and service requests in order to hide its current status from attackers and to increase the follow delay (defined in Section 2.2). Before a server changes its status from idle to active, it cleans its state (e.g., flushes the service queue). From time to time an idle server performs source-code downloading from a secure read-only medium to defend against possible Trojan horses [37].

Servers, firewalls and AGs keep track of the long backward hash chain described in Section 3 and use it to change the current active servers as follows: Let K_i be a key in this chain. Let S represent the set of indexes of the *servers* array. Also, let $P_k(S)$ represent an ordered set of all possible subsets of S with cardinality k. The cardinality of $P_k(S)$ is $N_p = \binom{k}{N}$, where N is the number of servers in S. Then, for each service epoch E_i, the set of current active servers is $P_k(S)[\text{MSB}_{\lfloor \lg N_p \rfloor} H'(K_i)]$, where $H'(\cdot)$ is a one-way hash function and $\text{MSB}_x(y)$ are the x most significant bits of y. The length, R_i, of service epoch E_i is uniformly distributed in the interval $[m, m+u]$ seconds as follows: $R_i = m + \text{MSB}_{\lfloor \lg u \rfloor}(H''(K_i))$ seconds, where m and u are system parameters and can be changed adaptively through roaming update messages. The value of m represents a lower-bound on the idle time of a server and should be long enough for the IDS at the server's network to detect and analyze attacks. We add k, N, u, the *servers* list, and $P_k(S)$ to the roaming update message.

The roaming honeypots scheme allows for k servers to be active concurrently, solving the first limitation described in Section 3. We next describe how the roaming honeypots scheme addresses limitations (2)-(5): (2) Offline subscription is no longer required in the roaming honeypots scheme as clients can subscribe to the service on-line through the AGN. (3) On the other hand, periodic roaming update messages are still necessary, but they are only sent to AGs instead of legitimate clients. Section 6.1 describes how these update messages are sent, that servers need only keep track of a few IP addresses of AGs, and that servers no longer maintain client IP addresses. (4) & (5): Client-server traffic is tunneled in both directions through the AGs, whom clients perceive as the servers. Thus, the roaming mechanism is transparent to the clients and clients neither receive roaming keys nor update their software. Also, an eavesdropper on client traffic would only discover the AGs and not the current active servers' addresses.

Next, we describe our simulation experiments to study the behavior of the roaming honeypots scheme.

5. Experimental Results

We have developed a simplified model of our roaming honeypots scheme in ns-2 [5] and conducted a number of simulation experiments to study the effect of different parameters on our scheme. We compare our scheme to a full replication scheme, in which all the N servers are active with client load uniformly distributed among them.

Simulation Model: We created a wrapper for the ns-2 built-in FullTcp agent and added a socket layer to support roaming. In addition, we created a multi-threaded FTP server and client modules to be used as our testbed application for the simulation. The code for these modules [3] works on top of the socket layer, where both roaming and TCP agent management take place. An FTP connection remains alive until either the FTP request is fulfilled completely or roaming occurs. If a roaming event is scheduled to cause the server module to be idle in the middle of an active connection, the client module will use its socket layer to record the current FTP state (number of remaining bytes) of the connection, to drop the current TCP agent, to connect to one of the current active servers selected uniformly at random, and to send the recorded FTP state to the new server module in order to resume the FTP transfer (TCP state, such as congestion window, is not carried over because the path to the new server is likely to be different in general).

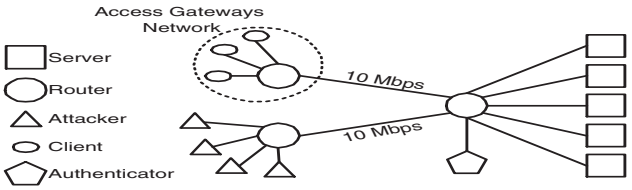

Figure 2. Simulation topology: all links are 1 Mbps unless noted.

Parameter(s)	Simulated Values
(N, k)	(5, 3)
Client load (CL)	0.9, 1.5 and 2.4 Mb/s
Attack load (AL)	1.5, 2.1, 3, 4, and 5 Mb/s
Migration interval (M)	3, 5, 7, 10, 20, and 30 sec
Attack type, N_{att}	Fixed-target, 5 and Follower, k
Follow delay	25%, 50%, and 75% of M

Table 1. Simulation Parameters

For simplicity, we fix the length, R_i, of all service epochs during a simulation run and term the fixed R_i value as *migration interval*. We also use a deterministic roaming pattern instead of using the hash chain.

As mentioned in Section 1, the benefit of our roaming honeypots scheme is two-fold; the filtering effect and the connection-dropping effect. If a servers receives a (malicious) request while acting as a honeypot, it records the requester's address into a list of attackers available to all servers. Servers filter out requests from nodes with addresses in the attackers list. To study the connection-dropping effect separately, we also model a roaming scheme in which no filtering takes place. In the description that follows, we refer to our roaming honeypots scheme as filter-roaming (or FR), the full replication scheme as non-roaming, and the scheme with no filtering as roaming (or R). We refer to the migration interval as M-interval (or just M).

Figure 2 depicts the simulated network topology. Nodes labeled as clients along with the router directly attached to them model the AGN depicted in Figure 1. The node labeled as authenticator replaces the functionality of roaming update messages (see Section 3). We use 1 and 10 Mbps links to model access and backbone links, respectively. For the sake of fast simulations, we do not use realistic link capacities nor realistic attack loads (although their relative values correspond to realistic cases).

Both clients and attackers request files of size 1 Mbits each with request inter-arrival times drawn from a Poisson distribution. We characterize both client and attack loads by the average bandwidth requested per second. For example, a load of 2.4 Mbps corresponds to requesting 2.4 files per second on average. When a new client is scheduled to request a file, it selects one server uniformly at random out of the N servers in the case of non-roaming, and out of the k current active servers in the case of both FR and R. Attack requests are distributed uniformly on the N_{att} targets. When an attack request is scheduled, an attack node is selected uniformly at random to launch the request. For fixed-target attacks, we set N_{att} to N, that is, attacking all the servers, to illustrate that the low-bandwidth requirement of service-level attacks allows the attackers to attack more servers. For follower attacks, we set $N_{att} = k$. Table 1 lists the simulation parameters and the values we use for them

Each simulation experiment has 10 runs (averaged in the graphs) with each run as follows: Legitimate clients send requests from time 0 to time 350 seconds and attackers from 50 to 250 seconds. A simulation run ends when all legitimate clients finish their requests. Figure 3 shows an example of how the average legitimate client response time changes with time throughout one simulation experiment.

Each point in Figure 3 represents the average response time (ART) of all legitimate requests issued within the previous 30 seconds. The ART of non-roaming keeps on increasing during attack. For both R and FR, dropping attack connections when roaming creates a short time-window, during which client requests can get serviced before the attack re-builds up. For FR, the ART increases slightly between time 50 and 180 seconds, that is, at the start of the attack. After all attack nodes are detected and filtered out, the ART is not affected by the attack anymore.

The relative performance of the different schemes (R, FR, and non-roaming) changes with different migration intervals, client loads, and attack loads. In the next paragraphs, we study the effect of varying these parameters. Results for both fixed-target and follower attacks exhibit the

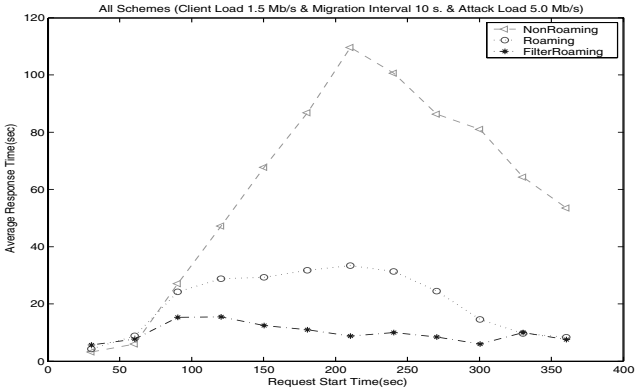

Figure 3. Average client response time (ART) throughout a simulation experiment. Fixed-target attack is from 50 to 250 seconds; k =3.

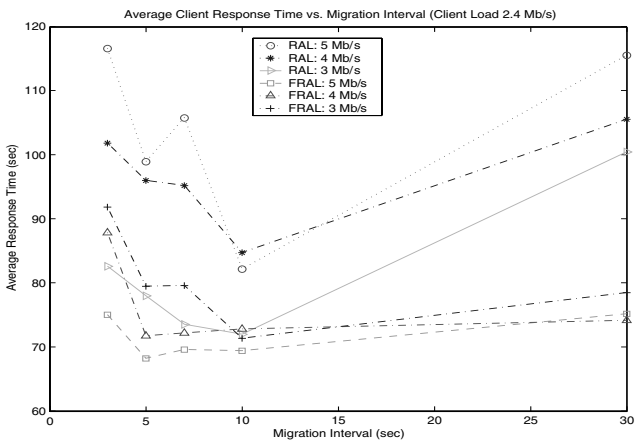

Figure 4. Effect of migration interval for different attack loads (AL)

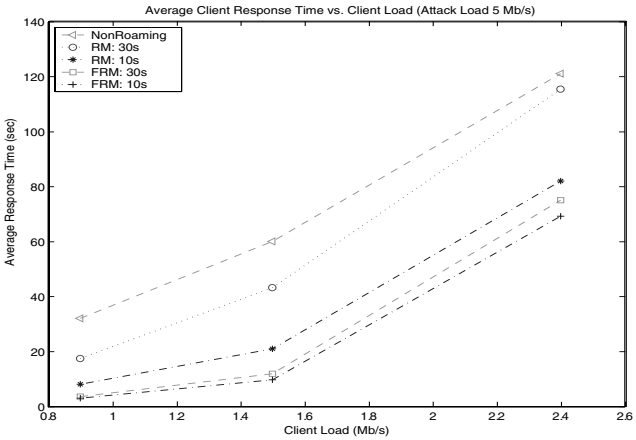

Figure 5. Effect of client load for different migration intervals

same behavior, except that for a specific attack load, fixed-target attacks (with $N_{att} = N$) cause more damage to the service, in terms of increased ART. Unless otherwise noted, we show the results for fixed-target attacks only.

Effect of Migration Interval (M): Figure 4 shows that there exists a critical value of M (10 seconds for R for this combination of parameters) that strikes a balance between roaming benefit and its overhead, which results mainly from forcing active legitimate connections into TCP connection re-establishment and consequently TCP slow start with each roaming event (in Section 6.3, we give more details on this overhead). For M below the critical value, roaming overhead is dominant; as M increases, the frequency of connection re-establishment (and slow start) decreases, resulting in a decreasing ART.

As M increases beyond the critical value, the ART increases. The reason is two-fold: First, as roaming happens less frequently, the connection-dropping effect of roaming occurs less frequently, as well. Second, the larger the M, the more client requests are issued to an attacked server and the more client requests will flock into the new server, diluting the effect of the reduced-attack time-window that roaming opens among them. Another (surprising) result is that, for FR, as attack load increases (i.e., attack request inter-arrival times decreases), the ART decreases, especially for small M, because of faster detection of attackers.

From the above, it is clear that the exact value of the critical M depends on service characteristics, such as average service time. The investigation of deriving this value is left for future work.

Effect of Client Load: Figure 5 shows the expected behavior of increasing ART with increasing client load for all schemes, under an attack load of 5 Mbps. The non-roaming scheme outperforms both R and FR under lower attack loads, as shown in the next section. Graphs with other attack loads (not shown here) exhibit a similar behavior.

Effect of Attack Load: Figure 6 shows that FR keeps the ART stable with increasing attack loads for a fixed number of attack nodes. This is because once the attack nodes are detected and filtered out, the attack has no impact. For non-roaming, distributing the load on all the servers helps in the case of low attack loads, but ART increases as attack load increases. For R, ART increases with increasing attack load, because the length of the time-window that roaming creates decreases with increasing attack load; note that the slope of the curve is lower for smaller M.

Effect of Follow Delay: As shown in Figure 7, as the follow delay increases, the ART decreases for both R and FR schemes. The effect of the follow delay is two-fold; during the follow delay, legitimate client requests enjoy an attack-free time-window (both R and FR benefit from this) and at-

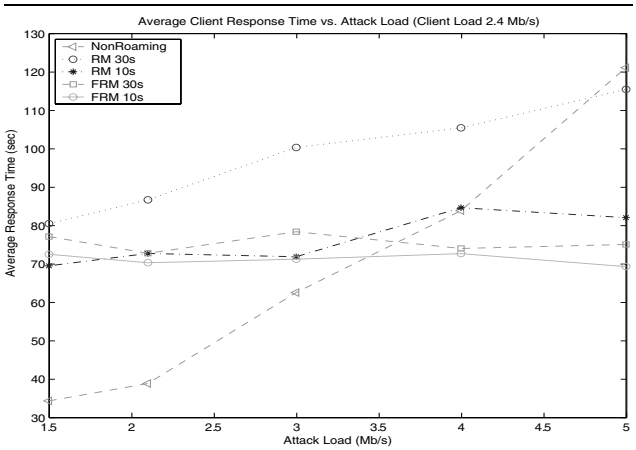

Figure 6. Effect of attack load

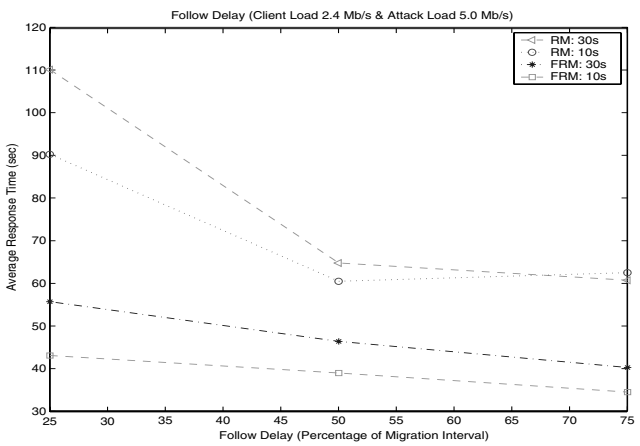

Figure 7. Effect of follow delay

tack requests arrive at each honeypot at a higher rate than in fixed-target attacks for the same attack load (attack load is distributed over 3 servers in follower attacks compared to 5 in fixed-target attacks), resulting in faster detection in FR. Because we plot the follow delay as a percentage of M, the scheme with 30 seconds M gets more (absolute) follow delay than the one with 10 seconds M, resulting in steeper slopes for higher M values. For non-roaming, the ART is the same for follower attacks and fixed-target attacks at about 120 seconds (not shown in this graph).

6. Discussion

6.1. Access Gateways Network (AGN)

The AGN acts as a layer of indirection between the server pool and legitimate clients; AGs keep track of the current active servers and forward client packets to them. In this section, we give more details about the AGN, discuss a scalable mechanism for distributing roaming update messages, and describe how migration of active connections, as described in Section 3, takes place.

We assume that the number of AGs assigned to a service is large and reconfigurations inside the AGN happen constantly in response to attacks and/or according to the policies of the operating entity. Also, we assume that the AGs are dispersed so that it is not possible to assume address aggregation (e.g., having a common IP prefix). Finally, we assume that the AGs vary in attached link capacity and intrusion resistance level.

Roaming update messages: AGs receive roaming update messages (see Section 3) to keep track of the current active servers. To improve the scalability of sending these messages, the group of AGs assigned to a service along with the service's server pool form a logical tree, with the server pool representing its logical root, and each node keeping track only of its immediate children. An AG may participate in more than one service tree. In the following description, without loss of generality, we consider a tree of one service and refer to it as "the tree". Next, we discuss how AGs are assigned to different levels of the tree, how an AG is inserted into the tree, and how roaming update messages are sent from the server pool to AGs.

As the link capacity of an AG and its intrusion resistance increase, the AG is assigned to a lower (closer to the root) level in the tree. With this policy, the AGs in the level immediately below the server pool exhibit the highest attack tolerance among AGs in the tree and are expected to feature the lowest reconfiguration rate. As a result, the number of AGs at this level can be kept small and servers need keep track of only a small number of AGs.

We utilize the Chord [30] lookup protocol to tolerate the high reconfiguration rate in the AGN as follows. Each AG (Chord node) is assigned a unique n-bit Chord identifier. To join a service tree, an AG performs both *address registration* and *parent registration*. For address registration, the AG registers the tuple $\langle id, address \rangle$ (similar to registering a trigger in the Internet Indirection Infrastructure ($i3$) [29]) at the Chord node responsible for storing the key id, where id is the n-bit string $([S_{ID}]_{n_1}||[\ell]_{n_2}||[index]_{n_3})$, where S_{ID} is a unique service identifier, $\ell(>0)$ is the AG's level in the tree, $index$ is the AG's index within level ℓ, the $||$ operator represents concatenation, $[x]_y$ means that x has a bit-length of y, $n_1 + n_2 + n_3 = n$, and $address$ is the AG's IP address.

The AG registers its IP address with its parent in the tree. The lowest-level AGs (at tree level 1) register their IP addresses with the server pool and every other AG registers its address with its parent as follows. The AG uses the string $([S_{ID}]_{n_1}||[\ell-1]_{n_2}||[index_{parent}]_{n_3})$ as the key identifier to lookup its parent's address registration entry, where $index_{parent}$ is the parent's index within its tree level $\ell - 1(\ell > 1)$. The AG then contacts its parent using IP to register its IP address at the parent. After registering with its parent, the AG receives roaming update messages using IP routing. We set the last n_3 bits of each AG's Chord identifier to zero (as in [29]), so that all AGs sharing the same tree level in the tree perform address registration with the same node, allowing the node to perform longest-prefix matching on the id field [29], and thus, allowing for selecting both $index$ and $index_{parent}$ at random.

An AG invalidates its address registration entry and performs a new address registration if, for instance, the AG leaves the tree or changes its level within the tree. Also, an AG invalidates its parent registration entry and performs a new parent registration if, for any reason, the AG changes its parent in the tree. If an AG changes its IP address, it updates its address and parent registration entries with the new address.

Roaming update messages traverse tree edges downward. As the tree level of an AG increases (and its attack tolerance decreases), the frequency of received roaming update messages increases and the indexes of roaming keys contained in the messages decrease, reducing the impact of compromising an AG with a low attack tolerance level and allowing the less-frequently reconfigured AGs to know future active servers' addresses for a longer time.

Connection migration: For each active connection it handles, an AG receives and stores periodic state update messages sent by the servers, as described in Section 3. At the end of each service epoch, a subset of the servers, S_{ai}, change their status from active to idle and another subset, S_{ia}, change from idle to active. The number of servers in both subsets is the same; that is, $|S_{ai}| = |S_{ia}|$. For each client connection, C, to a server in S_{ai}, its handling AG selects a server uniformly at random from S_{ia}, encapsulates the latest state update message for C in a TCP SYN packet with a *migrate* option [27], and establishes a connection with the selected server to take over C.

6.2. Network-level attacks

We have shown the effectiveness of our roaming honeypots scheme against service-level attacks. In this section, we give a qualitative analysis of the effectiveness of our mechanism against spoofed attacks, in which attackers use forged source addresses to hide their identity.

An attacker can launch a TCP sequence number guessing [20] attack and manage to request service using spoofed addresses. If such a spoofed request hits a honeypot, all future requests from the spoofed address will be dropped, denying service to a potentially legitimate address. However, because legitimate requests are tunneled through the AGN, an attacker needs to spoof an AG's address in order for this attack to be successful. An AG can easily detect that it is under such an attack (all its requests are being dropped) and can respond by changing its IP address. The AG then updates its address registration with the new IP address. The AG and all its immediate children should re-perform parent registration (as described in Section 6.1).

Defenses that tolerate packet source spoofing have been proposed elsewhere, such as Pushback [17] and Path Identifier (Pi) [35]. Our roaming honeypots scheme can work in combination with these defenses, to solve some of their limitations: (1) if the attack is mixed with legitimate flows and is comprised of many streams with a small bandwidth each, Pushback can cause collateral damage to legitimate traffic; (2) [15] describes how one low-bandwidth attack stream that exploits TCP's retransmission timeout (RTO) mechanism evades detection by Pushback and still causes severe throughput degradation; and (3) if attack packets are not accurately detected, Pi can cause collateral damage. Pushback can profit from our roaming honeypots scheme because it separates legitimate from attack traffic. In the case of Pi, a honeypot provides attack packets that can be used for filtering based on the Pi field. However, because the Pi field depends on the path traversed by a packet, we note that the Pi field value of a detected attack packet at one network cannot be readily used for packet filtering at another network.

6.3. Performance Shortcomings

Our roaming honeypots scheme incurs an overhead that causes performance degradation both in the absence of attacks and under low attack loads. This overhead results from three main factors: (1) The offered load, both legitimate and illegitimate, is distributed over k instead of N servers; (2) When a server switches from active to idle, all its current legitimate connections move to another server, re-establish TCP connections and re-enter TCP slow-start, losing their current TCP throughput; and (3) Legitimate connections flock simultaneously into new servers distorting the otherwise smoother request arrival distribution.

To offset the cost of the unused $N - k$ servers, we propose to add the typically available spare servers [6] into the server pool. Also, a scheme in which the value of k is varied adaptively depending on attack load would solve the first shortcoming and is left for future work.

7. Related Work

Although much work has been done in security, DoS attacks, intrusion detection, authentication, consistent hashing, and other related work, we only describe here work that is directly related to our roaming honeypots scheme.

At the cost of changing client software, *client puzzles* [12] allow for mitigating service-level attacks. Each client has to solve a cryptographic problem with varying complexity before the server allocates resources to the request and starts servicing it. *Puzzle auctions* [32] are implemented in Linux kernel as a framework to tune puzzle difficulty so as to minimize legitimate client cost in the presence of an adversary of unknown computing power.

Most solutions to the network-level DoS attacks try to develop a packet filter that can distinguish legitimate packets from illegitimate ones and hence drop illegitimate packets only [18]. The *D-WARD* defense system [19] is deployed at source-end networks, and autonomously detects and stops attacks originating from these networks.Wide deployment of D-WARD will motivate service-level attacks, on which we focused in this paper.

In the *Pushback* framework [17], once a router suffers from sustained congestion, it tries to detect flow aggregates that are contributing the most to congestion. The congested router rate-limits the detected flow aggregate(s) and sends

the aggregate signature (e.g., destination address) to upstream routers, which apply rate-limiting to the aggregate and recursively push the rate-limiting upstream toward attack sources. Pushback requires contiguous deployment; to overcome this limitation, *Selective Pushback* [21] proposes to send rate-limiting requests to routers sending traffic with higher than "normal" rates. The detection of these routers and the profiling of normal traffic are performed via an enhanced probabilistic packet marking scheme.

The *Path Identifier (Pi)* is a deterministic packet marking scheme that approximately identifies the path took by a packet. The attack victim uses the Pi mark to filter out malicious packets on a per-packet basis. Our scheme can be used in cooperation with either Pushback or Pi to filter attacks that use spoofed IP addresses.

DoS attacks have been classified as either single-source or multi-source, and it has been shown that the attack rate, as observed at the victim, of multi-source DoS attacks exhibit a ramp-up behavior, ranging from 200 ms to 14 seconds [10]. The staggered starting time of attack zombies causes this behavior and allows our roaming honeypots scheme to detect attack zombies and filter them out; efficiently throttling the attack rate before it reaches its peak.

IP hopping [11] protects a *public* server, whose clients use DNS to look up its IP address. In IP hopping, the server changes its IP address without changing its physical location. All packets destined to the old IP address are filtered at the network perimeter by a firewall. To avoid continuous server reconfiguration, a NAT (Network Address Translation) gateway can be used. IP hopping can be used both reactively and proactively. By using a virtual honeypot, such as [2], proactive IP hopping can provide the same effect that our scheme achieves in making it difficult for attackers to avoid hitting a honeypot and thus getting detected. However, IP hopping suffers from the following limitations. First, during the period of time in which the DNS entry of the old IP address is cached, all legitimate client requests using the old entry are filtered out. Second, IP hopping does not block a persistent attacker which looks up the new IP address using DNS. Although clients in our scheme utilize a public lookup service, such as DNS, to find the AGs, our scheme does not suffer from these limitations of IP hopping because the AGN is well provisioned, thus there is little chance that all AGs accessible to a client are down simultaneously. By physically moving the service and proactively cleaning server state, our scheme avoids the third limitation of IP hopping, keeping the server vulnerable to malicious state entries possibly implanted during the attack.

TCP-Migrate [27] and *Migratory-TCP* [31], which provide a framework for moving one end point of a live TCP connection from one location and reincarnating it at another location having a different IP address and/or a different port number, are used for mobility support and fault, or attack, tolerance. *Mutable Services* [8] is a framework to allow for reactively relocating service front-ends and informing only pre-registered clients of the new location through a secure DNS-like service. Our scheme builds on connection migration mechanisms and provides a secure framework for issuing the roaming trigger *proactively*.

The idea of introducing a layer of indirection to defend against DoS attacks was presented in the Secure Overlay Services (SOS) [13] architecture and the DoS Attack Mitigation (DAM) [7] framework. SOS uses the overlay network to hide the locations of a small number of proxy nodes (servlets) and allows only traffic from these servlets to enter the protected service's network. In order to gain access to the overlay network, a client has to authenticate itself with one of the replicated access points (SOAPs), which routes each client packet to one of the servlets using hash-based routing. The overhead of the overlay routing can be up to 10 times the direct communication latency. Our system model carries a large similarity with the SOS architecture, but uses one-hop tunneling through the AGs (typically close in proximity to clients) instead of hash-based routing for client-server traffic; thus it avoids this high latency.

DAM uses the network overlay to hide the locations of gateways to protect a CDN. Firewalls at the entry point of each replica in the CDN allow only traffic sourced at these secret gateways. The requirements of both continuous change of gateway locations, to avoid detection, and flexibility in the assignment of overlay nodes to services would cause an overhead of maintaining an up-to-date list of gateway addresses at the replica firewalls. Our scheme does not depend on this kind of whitelist-based filtering, and hence avoids this overhead.

8. Conclusion

Honeypots are either physical or virtual machines that are deployed to trap attackers. However, honeypots can be avoided by sophisticated attacks because of their fixed and detectable locations as well as being deployed on machines other than the ones they are supposed to protect. In this paper, we present roaming honeypots, a scheme for mitigating DoS attacks, in which honeypots are disguised within the protected server pool and are continuously and unpredictably changing. At any point of time, a subset of servers is active and providing service while the rest are acting as honeypots to capture attack packets and either record the address of the attackers (for non-spoofed packets) or use anti-spoofing defenses, such as Pi and Pushback, to filter or rate-limit attack streams. We presented distributed, randomized algorithms for changing the current active servers and allowing only legitimate clients to follow the roaming active servers.

Through simulations, we showed the effectiveness of our roaming honeypots scheme against service-level DoS attacks launched from behind firewalls and from external networks. As compared to a full replication scheme, in which all the servers are active, our roaming honeypots scheme shows a performance gain under the more realistic scenarios of high attack loads. A mechanism that adaptively changes the number of concurrent active servers depending on attack and client loads is a subject of future work.

References

[1] Akamai Corporation. http://www.akamai.com.

[2] Labrea - the tarpit. http://hts.dshield.org/LaBrea/.

[3] NetSec Group. http://www.cs.pitt.edu/NETSEC.

[4] Snort. http://www.snort.com.

[5] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[6] CERT. Denial of Service Attacks. http://www.cert.org/tech_tips/denial_of_service.html, 1997.

[7] B.-G. Chun, P. Mehra, and R. Fonseca. DAM: a DoS Attack Mitigation Infrastructure. *Under Submission*, January 2003.

[8] P. Dewan, P. Dasgupta, and V. Karamcheti. Defending against Denial of Service attacks using Secure Name resolution. In *Proceedings of SAM 2003*.

[9] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. In *RFC 2827*, May 2001.

[10] A. Hussain, J. Heidemann, and C. Papadopoulos. A Framework for Classifying Denial of Service Attacks. In *ACM SIGCOMM 2003*.

[11] J. Jones. Distributed Denial of Service Attacks: Defenses, A Special Publication. Technical report, Global Integrity, 2000.

[12] A. Juels and J. Brainard. Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks. In *Proceedings of NDSS'99*.

[13] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *ACM SIGCOMM*, 2002.

[14] S. M. Khattab, C. Sangpachatanaruk, R. Melhem, D. Mosse', and T. Znati. Proactive Server Roaming for Mitigating Denial-of-Service Attacks. In *Proceedings of ITRE'03*.

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

[16] J. Levine, R. LaBella, H. Owen, D. Contis, and B. Culver. The Use of Honeynets to Detect Exploited Systems Across Large Enterprise Networks. In *Proceedings of the 2002 IEEE Workshop on Information Assurance and Security*.

[17] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 62–73. ACM Press, 2002.

[18] J. Mirkovic, J. Martin, and P. Reiher. A Taxonomy of DDoS Attacks and DDoS Defense Mechanisms. Technical Report 020018, Computer Science Department, University of California, Los Angeles, 2002.

[19] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the Source. In *Proceedings of ICNP 2002*.

[20] R. Morris. A Weakness in the 4.2BSD Unix TCP/IP Software. Technical Report 117, AT&T Bell Labs Computer Science, 1985.

[21] T. Peng, C. Leckie, and K. Ramamohanarao. Defending against distributed denial of service attack using selective pushback. In *Proceedings of ICT 2002*.

[22] T. H. Project. *Know Your Enemy*. Addison-Wisley, Indianapolis, IN, 2002.

[23] R. Rivest. The MD5 message-digest algorithm. In *RFC 1321*, 1992.

[24] R. L. Rivest and A. Shamir. PayWord and MicroMint–Two Simple Micropayment Schemes. In M. Lomas, editor, *Proceedings of 1996 International Workshop on Security Protocols*, number 1189 in Lecture Notes in Computer Science, pages 69–87. Springer, 1996.

[25] C. Sangpachatanaruk, S. M. Khattab, T. Znati, R. Melhem, and D. Mosse'. A Simulation Study of the Proactive Server Roaming for Mitigating Denial of Service Attacks. In *Proceedings of ANSS'03*.

[26] C. Shields. What do we mean by Network Denial of Service? In *Proceedings of the 2002 IEEE Workshop on Information Assurance and Security*.

[27] A. C. Snoeren, H. Balakrishnan, and M. F. Kaashoek. The Migrate Approach to Internet Mobility. In *Proc. of the Oxygen Student Workshop*, July 2001.

[28] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[29] I. Stoica, D. Adkins, S. Zhaung, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *ACM SIGCOMM 2002*.

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*.

[31] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory TCP: Connection Migration for Service Continuity in the Internet. In *Proceedings of ICDCS 2002*.

[32] X. Wang and M. K. Reiter. Defending against denial-of-service attacks with puzzle auctions. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003.

[33] N. Weiler. Honeypots for distributed denial-of-service attacks. In *Proceedings of WET ICE 2002*.

[34] C. K. Wong, M. Gouda, and S. Lam. Secure Group Communications Using Key Graphs. In *ACM SIGCOMM 1998*.

[35] A. Yaar, A. Perrig, and D. Song. Pi: A Path Identification Mechanism to Defend against DDoS Attacks. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2003.

[36] J. Yan, S. Early, and R. Anderson. The XenoService - A Distributed Defeat for Distributed Denial of Service. In *Proceedings of ISW 2000*.

[37] L. Zhou, F. Schneider, and R. van Renesse. COCA: A Secure Distributed On-line Certification Authority. Technical report, Department of Computer Science, Cornell University, Ithaca, NY USA, 2000.