

Live Baiting for Service-Level DoS Attackers

Sherif Khattab, Sameh Gobriel, Rami Melhem, Daniel Mossé
Department of Computer Science,
University of Pittsburgh, PA 15260
{*skhattab, sameh, melhem, mosse*}@cs.pitt.edu

Abstract—Denial-of-Service (DoS) attacks remain a challenging problem in the Internet. By making resources unavailable to intended legitimate clients, DoS attacks have resulted in significant loss of time and money for many organizations, thus, many DoS defense mechanisms have been proposed.

In this paper we propose *live baiting*, a novel approach for detecting the identities of DoS attackers. Live baiting leverages group-testing theory, which aims at discovering defective members in a population using the minimum number of “tests”. This leverage allows live baiting to detect attackers using low state overhead without requiring models of legitimate requests nor anomalous behavior. The amount of state needed by live baiting is in the order of number of attackers not number of clients. This saving allows live baiting to scale to large services with millions of clients. We analyzed the coverage, effectiveness (detection time, false positive and false negative probabilities), and efficiency (memory, message overhead, and computational complexity) of our approach. We validated our analysis using NS-2 simulations modeled after real Web traces.

Keywords— Denial-of-Service, attacker detection, group testing.

I. INTRODUCTION

Denial-of-Service (DoS) is a major security problem in computer systems and networks [1], [2]. In a DoS attack, a group of attackers try to make a service unavailable to legitimate clients for unacceptably long periods of time [1]. *Service-level DoS attacks* target server resources by issuing legitimate-like service requests at a high rate to overwhelm the victim servers. This attack is particularly challenging because it usually does not trigger network-level alarms, and is thus hard to detect by network-level defenses [3], [4].

Many DoS defenses have been proposed to detect DoS attacks at the ISP [5], [6], [7], and many ISPs appreciate these proposals as a way to provide value-added services to their customers. Deploying defenses at the ISP level has other benefits, such as sharing block-lists of detected attackers among many services. A defense system has to be efficient and scalable to be deployable at the ISP level. Although detecting service-level DoS attack is always possible [2], detecting the identities of the *attackers* is much harder in general and requires high state overhead. Our focus is on the *efficient* detection of service-level DoS attackers.

Our main contribution is a novel detection algorithm, for which we coin the term *live baiting*, to accurately and efficiently discover service-level DoS attackers among clients of a public service. The algorithm is based on group-testing theory [8], [9], which aims at discovering defective members in a population with the minimum number of tests; each test is applied to a population subset. Our defense system divides

the service capacity into *virtual servers*, called *buckets*, and we employ group testing to carefully design client assignment to buckets to enable efficient pinpointing of DoS attackers with low false positive and almost zero false negative probabilities.

Two assumptions of group-testing theory limit its direct applicability in DoS-attacker detection. First, an estimate of the number of defective members is assumed to be known beforehand. It may be hard to come up with such an estimate a priori. Thus, we develop an adaptive technique that starts with an initial probing estimate, corrects the estimate as time proceeds, and adjusts the algorithm parameters accordingly. Second, group-testing assumes that defective members are “faithful,” that is, they cannot change test results maliciously to hide their existence. DoS attackers have the motivation to avoid detection and, thus, we develop a technique to detect DoS attackers even when they manipulate test results.

In a sense, we extend the idea of baiting, or setting traps for malicious clients [10], [11] by noting that the more hidden the traps, the harder they are to evade by attackers. For instance, honeypots, decoy systems deployed in unused IP address ranges, are effective against random-scanning worms [12], [13], but they actually represent “dead baits” that are easy to avoid. *Roaming honeypots* camouflages the honeypots as servers, but is limited to the detection of outsider attackers against private services [14], [15]. In this work, we use the protected service itself as the “live bait”, making our trap more attractive to attackers and much harder to evade.

The merits of live baiting are as follows.

- Live baiting can quickly and efficiently detect DoS attackers against a large service with a huge client population. In live baiting, detection of DoS attackers is done with manageable memory and computational overheads.
- Live baiting protects an open service whose clients are typically not known a priori, and it is not restricted to services with human clients who can solve puzzles such as CAPTCHA [4], [16].
- To our knowledge, live baiting is the first work to apply group-testing theory to the DoS attack problem. Moreover, live baiting can detect the actual number of attackers and adaptively adjust its parameters. This adaptability is a novel contribution to group-testing theory.
- Live baiting is an end-to-end mechanism, that is, it requires no modifications inside the network and only requires minor server and client modifications, thus reducing the introduction or addition of vulnerabilities.

We analyzed three aspects of detection: (a) *coverage*; (b) *effectiveness*: detection time, false positive and false negative

probabilities; and (c) *efficiency*: memory, message, and computational complexities. We compared our analytical results with NS-2 simulations that are based on real Web traces. We also developed techniques to discover the number of attackers, to adapt the scheme accordingly, and to handle complex system models and intelligent attack types tailored specifically to deceive our detection algorithm.

Paper organization. In the next section, we present a brief background on group-testing theory. In Section III, we define the problem of detecting service-level DoS attackers along with server and attack models. We then describe the live baiting algorithm for simple server and attack models in Section IV. In Section V, we summarize our analytical results and describe our NS-2 simulation study, in which we validated the analytical results and confirmed the applicability of the algorithm in a real scenario. In Section VI, we present our mechanism to detect and adapt to the number of attackers. Section VII presents techniques to extend our algorithm to more complex system and attack models. We overview current research in DoS defense in Section VIII, and finally we summarize our results and discuss their implications and limitations in Section IX.

II. GROUP TESTING OVERVIEW

The first application of group testing [8], [9] was during WWII; instead of testing every blood sample individually, groups of samples were pooled together and tested collectively [8]. If the outcome of the group test is negative, all samples in the group are good (disease-free). Although group testing has been used since then in many security and networking applications, such as data forensics [17], cryptography [18], multiple-access channels [19], and broadcast security against jamming [20], our work is the first to apply this powerful theory to the DoS attack problem.

Group testing aims mainly at identifying the defective (special) members of a population with few tests. There are two classes of group-testing mechanisms [21]. “*Non-adaptive*”, or single-stage, specifies all tests simultaneously without the benefit of using the outcomes of previous tests to determine the present test. *Adaptive* (multi-stage) group testing uses feedback from previous test results to determine subsequent tests. In this paper, we use a simple non-adaptive group-testing scheme as a proof-of-concept, but adaptive schemes can be used as well.

Non-adaptive group testing is modeled as a $T \times N$ matrix [21], where N is the total number of members, and T is the number of tests. Matrix rows represent tests and columns represent group members. When a matrix element (i, j) is set to 1, this means that member j participates in test i . An example of a group testing matrix for a population of 10 members with 4 tests is shown in Figure 1.

Test results are represented as a vector with an element for each test. For simplicity we assume that the test results are binary. So, a test result is set to 1 if the corresponding test returns a positive result, that is, if the test was applied to a group with at least one defective member. In the example shown in Figure 1 the 2nd, 6th, and 8th members are defective.

Detection of defective members. The detection algorithm discovers the defective members using the result vector and

	Members (defectives underlined)										<i>Test Results</i>
	1	<u>2</u>	3	4	<u>5</u>	<u>6</u>	7	<u>8</u>	9	10	
Tests	0	1	0	0	0	0	1	1	1	0	1
	1	0	1	0	0	0	1	0	1	1	0
	0	0	0	1	0	1	0	1	0	1	1
	1	0	0	1	1	0	1	0	0	0	0

Fig. 1. An example of a group-testing matrix.

the matrix. The algorithm we use in this paper works by excluding a negative (non-defective) member if it participates in a “large enough” number of tests with a negative result. For instance, if we assume that a member has to participate in only *one* negative test to be excluded, then in the above example, members 1, 3, 4, 5, 7, 9, and 10 will be excluded. This leaves us with the defective members 2, 6, and 8 as suspects. In this specific example, all defective elements are detected, and all non-defective members are cleared.

Matrix construction. Many matrix construction algorithms [21], [22] can be used with our live baiting scheme. For simplicity, we use the randomized construction method [17], [21], in which each bit in the matrix is set to 1 with a fixed probability p . As will be discussed later, the value of p is derived to maximize the detection accuracy.

False Positive and False Negative Probabilities. A false positive is when a non-defective member gets falsely identified as defective, while a false negative is when a defective member ends up being not detected. In the example above, both the false positive probability and the false negative probability are 0. In general, Goodrich *et al* [17] showed that the simple detection algorithm discussed above detects all defective members with the false positive probability $FP = [1 - p(1 - p)^d]^T$, where d is the number of defective members in the group and T is the number of tests used to detect defective members. By differentiating the above equation with respect to p , the optimal value of p , the value that yields the minimum false positive probability, is $\frac{1}{d+1}$ [17]. Thus, the minimum false positive probability for a given number of tests T is:

$$FP = [1 - \frac{1}{d+1}(1 - \frac{1}{d+1})^d]^T \quad (1)$$

Finally, from Eqn. 1 we derive the number of tests, T_{fp} , required to achieve a target false positive probability fp :

$$T_{fp} = \frac{\log(fp)}{\log(1 - \frac{1}{d+1}(1 - \frac{1}{d+1})^d)} \quad (2)$$

As we will show later, T_{fp} is $O(d)$, that is, the number of tests is in the order of number of defective members (attackers) not the total number of members (N). This observation is key to the scalability of our live baiting algorithm.

The above equations assume that an estimate of the number of defective members (d) is known a priori. It is also assumed that if a defective member participates in a test, the test result is always negative, that is, a defective member cannot arbitrarily alter the result of a test. Both these assumptions may not hold in DoS attacker detection. First, it may be hard

to estimate a priori the number of attackers out there. Second, DoS attackers want to avoid detection and may maliciously alter the results of tests to obscure their presence. We relax both these assumptions in Sections VI and VII.

III. PROBLEM DEFINITION

Our work applies group-testing theory to mitigate DoS attacks against an open public service, such as a Web service, where any host can access the service at any time from anywhere in the Internet. A number of attackers, x , launch service-level attack by issuing legitimate-like service requests at a high rate to overwhelm the victim servers.

The problem is then to detect the x attackers with high accuracy (low false positive and false negative probabilities), in a short time, and with small state overhead. The requirement of small overhead is particularly important if the detection system is to be deployed at an ISP to protect many customers.

Although each service can keep the state of its clients, the total number of clients over all the services may far exceed the capacity of a single ISP. Therefore, the per-client monitoring approach, although provides the most accurate DoS detection with the smallest detection time, requires state that grows linearly with the total number of clients and depends on models of legitimate or anomalous behavior. Thus, it can be prohibitively expensive. Our live baiting algorithm maintains $O(x)$ state, that is, state in the order of the number of attackers, which is much smaller than the number of clients. Even in the case of attacks launched from botnets, although botnet sizes have been estimated in the order of hundreds of thousands, recent studies question this estimate and report much smaller sizes (in the order of thousands) [23]. Moreover, live baiting does not require models for legitimate or attack behavior.

Our live baiting algorithm is also applicable to DoS attacks on resources other than server resources, but in this paper we focus on service-level or application-level attacks and assume that, as in [4], attacks cause no network-layer congestion.

Server and Attack Models. The service to be protected (e.g., Web or FTP) is provided by a server cluster with an aggregate request-processing capacity of C requests per second. A legitimate client sends requests at an average rate of R requests per second. Attackers are *misbehaving* clients that send requests at a rate higher than the legitimate to cause the aggregate service request rate to exceed the service capacity. In other words, each attacker sends requests at a rate $R_{attacker} > R$ ¹. We assume that the attackers aim at causing the *maximum damage* to the service by causing as many legitimate requests as possible to be dropped, and in Section VII we describe how to handle the case when attackers deviate from this behavior. Further, we assume that each client has a unique, un-spoofable identifier, which can be stored in an application-layer cookie in Web applications for instance.

It should be clear that live baiting does not defend against highly diffused, massively parallel DoS attacks, where the number of attackers is so large that they mimic legitimate clients with very low request rates. This problem is equivalent

¹Even though we assume that $R_{attacker} > R$, we do not need to know what are the values of these thresholds.

to differentiating such DoS attacks from flash crowds [24], [25] and is not the focus of this work.

We use this simple server and client model mainly to derive our analytical results, but our scheme extends to more complex models. For instance, in Section V we show the effectiveness of live baiting for a real Web service with bursty legitimate request rates and non-uniform request service times.

IV. LIVE BAITING FOR SIMPLE SERVER AND ATTACK MODELS

We present an overview of our live baiting algorithm followed by a description of its details. For simplicity, we describe the algorithm based on the simple service and attack models described in Section III and discuss more complex models in Section VII.

A. Live Baiting Overview

The service capacity is divided into a number of virtual servers, called *buckets*, and each client receives from the service a set of tokens that are valid to access only a set of buckets assigned to the client. The client then sends requests to the buckets assigned to it. To prevent attackers from accessing buckets other than those assigned to them, the tokens are securely generated and checked at the servers so that forged tokens/requests are dropped and clients using them are identified as attackers.

We use group testing to design the set of buckets assigned to each client. In the DoS-attacker detection domain, service clients are mapped to the population and DoS attackers are mapped to the defective members that we seek to detect.

As described in Section II, a group testing design is represented as a binary matrix with rows corresponding to tests and columns to population members. Buckets correspond to the tests or the matrix rows and clients are mapped to matrix columns. Each client is given *tokens* for each *1-bit* in its column, allowing it to send requests to the corresponding buckets. Clients piggyback a token on each request they send, so that requests are credited to the corresponding buckets.

Periodically, the number of requests in each bucket is counted, and if it exceeds a threshold that we call the High Water Mark (*HWM*), the test result corresponding to the overflowed bucket is labeled as positive, indicating that at least one attacker is assigned to the overflowed bucket.

Next, we describe the details of our algorithm at servers and at clients. We note that our algorithm requires no modifications inside the network. Moreover, the modifications in the servers and clients are minimal. Table I summarizes the terminology we use in the paper.

B. Live Baiting at Servers and Clients

We describe the building blocks of the live baiting algorithm at the servers and clients. More details can be found in [26].

Buckets and Tokens. The algorithm virtually divides the service capacity, C , into T buckets, where each bucket is implemented as a counter. Given an upper bound, N , on the total number of clients and an initial estimate of the number of attackers, d , we construct a $T \times N$ binary matrix in which an entry is 1 with probability $\frac{1}{d+1}$ and 0 otherwise. The matrix

TABLE I
TERMINOLOGY

Symbol	Meaning
FP	false positive probability
FN	false negative probability
N	maximum number of clients
C	server capacity (req./sec.)
T	number of buckets
HWM	high water mark of buckets
P	length of detection interval
R	legitimate client request rate (req./sec.)
$R_{attacker}$	attacker request rate (req./sec.)
d	initial estimate of the number of attackers
x	actual number of attackers
$B_{attackers}$	number of buckets attacked

depth, T , is determined based on the required false positive probability according to Eqn. 2.

Each new client is assigned one column in the matrix and is given one token for each 1 -bit (i.e., for each bucket) in its column. Thus, the average number of tokens per client is $\frac{T}{d+1}$. Each token is a tuple $\langle i, version, hash(i, version, K_{server}) \rangle$ that contains the row index i , the matrix version (to allow for matrix updates as will be described in Section VI), and a secure hash of length at least 160-bits computed using a secure hash function, such as SHA-2 [27]. The input to the hash function is the row number, the version, and a server secret, K_{server} , of length at least 80-bits, which is changed periodically (e.g., every day).

Attackers would want to generate their own tokens so that they can send their requests to different buckets than the ones assigned to them. If they manage to achieve that, our scheme would fail to detect them. Therefore, we employ the secure token generation described above to ensure, with high probability, that malicious clients cannot generate valid tokens by themselves. The probability of successfully guessing a valid token value is very low: $\frac{T}{2^{80}}$, where T is the number of buckets, that is, valid tokens, and 80 is the server secret length.

We emphasize that the service is open to the public, and tokens are available to all clients that request them (including attackers). However, we assume that each client is limited to a single column in the matrix. Enforcing this limit is typically straightforward. Token distribution is a simple service, whereby the legitimate token-request rate per client and the service time are known. Thus, it requires a simple rule for rate-limiting, and efficient algorithms can be used to detect misbehavers (e.g., the sample-and-hold algorithm [5]). On the other hand, live baiting protects a general service that requires more complex models for legitimate or anomalous behavior [33], which are not always easily defined.

High Water Mark (HWM). We calculate the expected number of requests each bucket is expected to receive in a detection interval of length P seconds, assuming that all clients are legitimate. We call this value the High Water Mark (HWM) of a bucket. Each legitimate client is expected to

send $R \cdot P$ requests every P seconds, distributed uniformly over its assigned buckets (on average, $\frac{T}{d+1}$ buckets per client). Also, clients are distributed uniformly over the buckets; each bucket has on average $\frac{N}{d+1}$ clients assigned to it. Thus, buckets are expected to receive almost the same number of requests $= \frac{N}{d+1} \cdot R \cdot P \cdot \frac{d+1}{T} = \frac{N \cdot R \cdot P}{T} = \frac{C \cdot P}{T}$ requests every P seconds. We thus set the HWM to this value.

For each bucket, only HWM requests out of the received requests are serviced at a high priority, and the rest are serviced at a lower priority. Low priority requests are dropped first when the servers are overloaded.

Detection. Every P seconds, if a bucket receives more than HWM requests, the corresponding test is marked as positive. The detection algorithm starts with all clients in the *suspects list*, and for each negative test, we remove the clients assigned to the corresponding bucket from the list. According to our simple attack model, if these clients were attackers, they would have caused *all* their assigned buckets to exceed their HWM (in Section VII we remove this assumption). The rest of the suspects list are labeled as attackers and are added to a *block-list* to block their future requests.

Request Filtering. When a request arrives at a server with a token $\langle i, version, hash_i \rangle$, the server drops the request if it is from an attacker already detected (in the block-list). The server also checks whether the token is not expired by checking its version against the current matrix version. If the token has expired, the server drops the request and sends to the client a new set of tokens. Otherwise, if the token is current, the server checks its validity by comparing the secure hash of $(i, version, K_{server})$ with the received $hash_i$. If they differ, the client sending the token is inserted into the block-list. Clearly, checking the token validity is a quick operation and can be made even quicker by reducing the size of the secret key, but with a higher probability of illegal token generation.

Mapping Buckets into Physical Servers. Each physical server maintains counters for all the buckets, and it increments the corresponding counters for each request it processes. These local counters contain partial counts because requests of a particular bucket may be serviced by more than one server. Each protected server sends its bucket counters to a central detection server at the ISP, which in turn runs the detection algorithm and updates the block-list. We note that requests are mapped to the physical servers using a variety of mapping functions, such as load-balancing and session-based routing (requests in the same session are routed to the same server to maintain session integrity). Live baiting is orthogonal to the request-mapping strategy, because its operation relies only on maintaining the bucket counters. Live baiting is also independent of the number of servers.

Live Baiting at Clients. Each client receives a set of tokens and piggybacks one token on each request it sends to the server. The client uses its tokens in a round-robin fashion (more details can be found in [26]). For now, we assume that attackers also use this algorithm and overflow all their assigned buckets; in Section VI, we consider other attack strategies.

V. EVALUATION

We evaluated our scheme analytically and conducted a simulation study to validate our analysis. We studied the following metrics: coverage; effectiveness, in terms of detection speed, false positive and false negative probabilities; and efficiency, in terms of storage, message, and computational complexities.

A. Summary of Analytical Evaluation

Due to lack of space, we only summarize our main analytical results. First, the number of buckets needed to achieve a specific false positive probability is linear in number of attackers. Second, the detection time is constant with respect to the number of attackers. Third, the memory overhead of the buckets is $O(d)$ and the matrix needs $O(N)$ memory. However, the matrix is accessed only every P seconds and not for each service request. Fourth, per-request processing has negligible ($O(\log(d))$) overhead, and the $O(N)$ detection algorithm invoked every period P . Finally, the message overhead caused by the tokens is small, and tokens can be piggybacked on service requests, such as HTTP GET. The complete analysis can be found in [26].

We highlight an important analytical result that will be used to estimate the number of attackers in Section VI. The average number of buckets assigned to attackers is:

$$B_{attackers} = T \left(1 - \left(1 - \frac{1}{d+1}\right)^x\right) \quad (3)$$

In the above equation $\left(1 - \frac{1}{d+1}\right)^x$ is the probability that a particular bucket has no attackers assigned to it. Thus, $\left(1 - \left(1 - \frac{1}{d+1}\right)^x\right)$ is the fraction of buckets to which at least one attacker is assigned, and thus, the bucket is acquired by attackers.

B. Simulation Study

The purpose of the simulation study is to validate our analytical results in more complex system and attack scenarios. In particular, we studied the effect of burstiness in request arrivals and irregularities in request processing times of real services on performance. Our results show that live baiting is indeed effective under real service models.

To this end, we developed a simulation model of live baiting in NS-2 [28]. Our simulation topology consisted of a client node modeling a “client cloud” and a server node, connected by a 100Mbps link with 10ms propagation delay. Our model extends the PackMime HTTP traffic generation module [29], which generates HTTP 1.1 sessions with session inter-arrival times, client waiting times, request sizes, and response sizes derived from empirical traces. We modified PackMime to include separate client and attacker distributions.

Instead of generating server delays independently from system load, we modeled the server CPU as a link, called “CPU-link”, with a bandwidth equivalent to the service capacity. The link is attached to a round-robin queue with sub-queues corresponding to the buckets. The queue buffer is shared fairly among the buckets, so that each bucket has a guaranteed share equal to total buffer size divided by number of buckets. We model request service by transmitting a packet in the CPU-link with size equal to the response size. When the packet is received at the end of the CPU-link, a packet is sent with 0

delay back to the server, who immediately sends the response to the client. If a packet is dropped from the CPU-link queue, the client of the dropped request is notified which in turn attempts a maximum of three retransmissions with one second delay between attempts.

Attack sessions have similar request size, response size, and client waiting time distributions as legitimate sessions, but they are generated at a higher rate. We slightly modified the detection algorithm to count request drops (more specifically, packet drops in the CPU-link’s buffer) in each bucket instead of the number of requests. We found the number of request drops to be a better metric than request count in this setting, because it is a more direct indicator of server overload; using the number of requests resulted in more false positives due to request burstiness of legitimate clients. We tried a few values for the HWM and got a good performance with a HWM value of 10 drops.

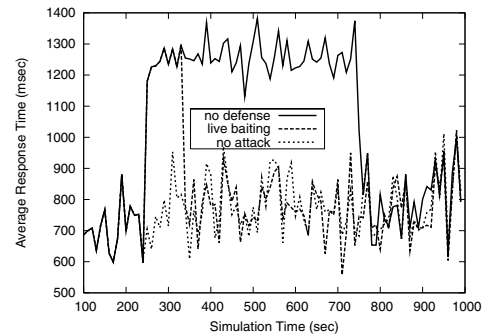


Fig. 2. Average response time for all schemes vs. simulation time. Attack = [250s, 750s]; detection interval = 90s.

To evaluate our algorithm, we used six metrics: average response time of legitimate requests, throughput of legitimate requests (legitimate requests completed per second) and legitimate sessions (legitimate sessions completed per second), false positive rate, false negative rate, and legitimate utilization; the latter is defined as the fraction of time the server’s CPU was processing legitimate requests. We used two performance baselines, namely *no attack* and *no defense*.

We simulated 10,000 legitimate clients with aggregate rate of 20 sessions per second. The CPU-link was 100Kbps, zero propagation delay, and 100Kbit total buffer size in one direction. The reverse direction was 100Mbps with zero propagation delay, provisioned to introduce negligible delays and no drops. The simulation time of each run was 1,000 seconds, the attack started at 250s, and ended at 750s. The warm-up period was 100 seconds, after which statistics were collected. The detection intervals started after the attack started, and the request-drop counters were reset every detection interval.

Figure 2 illustrates the effectiveness of live baiting in detecting attackers and reducing the attack effect on response time. In this figure, requests were grouped every 10 seconds based on their start time, and the average response time of each group was plotted. The aggregate attack rate was 100 sessions per second, number of buckets $T = 1,000$, number of attackers $x = 100$, and the estimate $d = 100$ as well. The detection interval was 90 seconds. With no defense, the

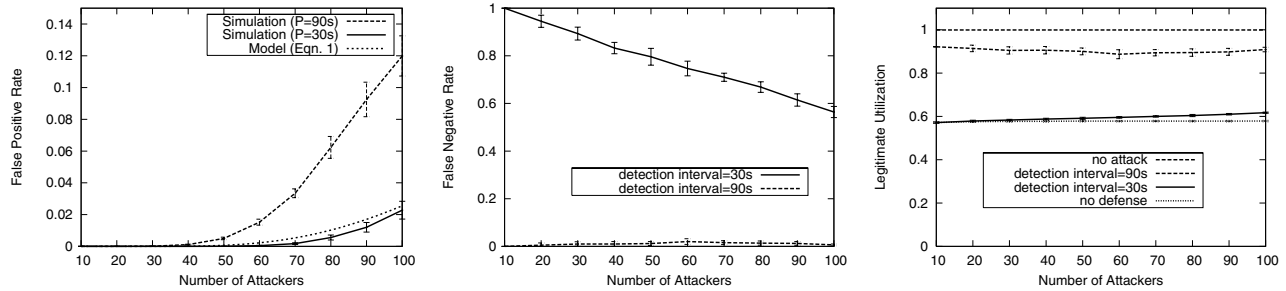


Fig. 3. Effect of number of attackers under accurate estimation.

response time almost doubled after the attack started. With live baiting, response time spiked after attack started, but quickly (at 340s, after the first detection interval) went down, after the attackers were detected and added to the block-list. After the attack, the response time with live baiting was slightly less than with no attack, because a few legitimate clients were blocked (false positives) and consequently, the load on the system was slightly reduced.

We also studied the false positive and false negative rates. First, to validate Eqn. 1, we fixed the aggregate attack rate at 100 sessions per second and increased the number of attackers with accurate estimation, that is, $d = x$. The error bars in the next figures represent 90% confidence intervals. As Figure 3 shows, live baiting with 90s detection interval and 1,000 buckets was successful in detecting up to 50 attackers with a false positive probability of less than 0.005 and almost zero false negatives. We found that simulation results matched Eqn. 1 when the detection interval was 30 seconds (Figure 3 (left)). However, 90s interval introduced more false positives than predicted by the model. This difference was due to bucket overflows resulting from bursty legitimate request arrivals. Although the model correctly predicted the trend of the false positive probability increase with increasing d and fixed number of buckets, it needs further enhancement to take the effect of the detection interval into consideration.

The false negative rate with $P = 90s$ is almost zero (Fig. 3 (middle)), because 90 seconds is enough time for each attacker in this scenario to overflow all its assigned buckets. On the other hand, the false negative rate with 30s interval decreased with increasing number of attackers. Although the rate per individual attacker decreased, resulting in more time needed for the attacker to overflow its buckets, the number of these buckets ($= \frac{T=1000}{d+1}$) decreased as well.

The effect of the number of attackers on the legitimate utilization depended on the detection interval length, as shown in Fig. 3 (right). When the interval was short, the decrease in false negatives and the small increase in false positives had two effects: the legitimate utilization was much smaller than at large intervals, and the utilization slightly increased. Conversely, when the interval was large, legitimate utilization slightly decreased due to the increase in false positive rate while the false negative rate was constant.

In summary, live baiting is a promising technique for detecting attackers, but special care should be taken in selecting the detection interval length and the *HWM*.

VI. ADAPTIVE LIVE BAITING

Live baiting relies on an estimate, d , of the number of attackers, which may be different than the actual number of attackers, x . Below we show how live baiting detects and adapts to a number of attackers different than the estimated one.

First, when $x \neq d$, Eqn. 1 becomes:

$$FP = [1 - \frac{1}{d+1}(1 - \frac{1}{d+1})^x]^T \quad (4)$$

We conducted a simulation experiment where we varied the number of attackers while fixing the estimate $d = 100$. The purpose is to study the effect of underestimating and overestimating the number of attackers and to validate Eqn. 3 and Eqn. 4. As Fig. 4 (left) shows, the false positive rate increased with increasing number of attackers. The 90s has more false positives than predicted by Eqn. 4 because of the burstiness reason previously discussed. Moreover, the false negative rate was almost zero until the number of attackers exceeded the estimate (right y-axis in Fig. 4 (left)). The false negative rate then increased because the rate per attacker decreased (fixed aggregate attack rate) and number of buckets per attacker remained fixed (because d was fixed), so the time that each attacker needed to overflow its buckets exceeded the 90s detection interval. The legitimate utilization (not shown) slightly dropped to about 0.93 until number of attackers exceeded the estimate, then dropped steeply due to increasing false negatives and approached the no defense baseline.

We use Eqn. 3 to estimate the number of attackers. When $x = d$, the fraction of attacked buckets (normalized to T) tends to $1 - \frac{1}{e} = 0.63$. However, as x increases beyond d , the fraction of attacked buckets increases. This is because now $\frac{1}{d+1}$ is constant relative to x , the number $1 - \frac{1}{d+1}$ is < 1 , and as x increases, $(1 - \frac{1}{d+1})^x$ decreases and $1 - (1 - \frac{1}{d+1})^x$ increases. We use this increase as a flag to increase our estimate of the number of attackers to \hat{d} :

$$\hat{d} = \frac{\log(1 - \frac{B_{attackers}}{T})}{\log(1 - \frac{1}{d+1})}, B_{attackers} < T$$

The above formula was derived from Eqn. 3 by taking the logarithm of both sides and solving for x .

We measured the maximum number of attacked buckets (buckets with more drops than *HWM*) over all detection intervals within each run. This number was plotted in Fig. 4 (right)

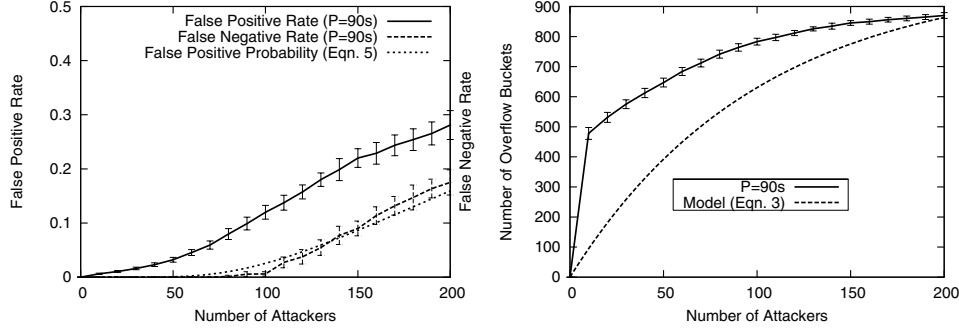


Fig. 4. Effect of under-estimation and over-estimation of the number of attackers. The number of buckets under attack can be used to detect both under-estimation and over-estimation situations.

together with Eqn. 3, which models the number of attacked buckets. With 90s detection interval and $HWM = 10$, the number of attacked buckets exceeded the model estimation. Thus, using Eqn. 3 to estimate the number of attackers would yield a larger number than the actual number of attackers. For example, at 700 attacked buckets, the model estimates the number of attackers as about 120, but the actual number of attackers would be about 70 attackers (this can be shown by drawing a horizontal line at $y = 700$ in Fig. 4 (right)).

Effectively, using Eqn. 3 would make the system over-estimate the number of attackers, which would cause slightly more state, but results in fewer false positives. For instance, at an estimate of $d = 100$, the actual number of attackers would be about 60 (this can be shown by drawing a horizontal line at $y = 630$ in Fig. 4 (right) that intersects the two curves at around $x = 60$ and $x = 100$, respectively). The false positive rate from Fig. 4 (left) would be 0.04 (at $x = 60$) instead of 0.12 if the estimate was accurate ($x = d = 100$).

Matrix regeneration. Once a new estimate, \hat{d} , is calculated, if it is larger or much smaller than the initial estimate, a new matrix is generated with the new estimate, and the number of buckets is updated according to Eqn. 2 to maintain the target FP . The *matrix version* is incremented every time the matrix is regenerated. Tokens with versions older than the current matrix version are dropped, and clients are assigned new tokens as described in Section IV.

VII. EXTENSIONS TO THE SIMPLE ATTACK MODEL

In this section we discuss extensions to the live baiting scheme to handle more intelligent attackers.

Relaxing the Maximum-Damage Assumption. In our previous discussion, we assumed that the attackers attack **all** buckets assigned to them. Although this attack scheme causes the *maximum damage* to the server (i.e., maximum number of dropped requests), a more intelligent attacker might choose to attack only a subset of the assigned buckets. In this case the attacker will be cleared (removed from the list of suspects) for some of the buckets, and hence, evades detection. To model this intelligent attack, we assume that each attacker attacks each assigned bucket with an *attack probability* ρ_{attack} . The false negative probability is then:

$$FN = 1 - [1 - \frac{1}{d+1}(1.0 - \rho_{attack})(1 - \frac{1}{d+1} \cdot \rho_{attack})^{d-1}]^T$$

In the above equation, $\frac{1}{d+1}(1.0 - \rho_{attack})(1 - \frac{1}{d+1} \cdot \rho_{attack})^{d-1}$ is the probability that a bucket assigned to an attacker ends up un-attacked. Note that using the above equation, $FN = 0$ when $\rho_{attack} = 1$ as expected. On the other hand, the number of attacked buckets, and hence, the damage caused by the attack, is reduced to:

$$B_{attackers} = T(1 - (1 - \frac{1}{d+1} \cdot \rho_{attack})^x) \quad (5)$$

Fig. 5 (left) plots the fraction of attacked buckets ($\frac{B_{attackers}}{T}$) at $d = 100$. It shows that this fraction decreases, and hence, the amount of damage caused by the attack reduces, with decreasing attack probability.

We note that this intelligent attack is not possible if the assignment of requests to buckets is not controlled by the clients (by attaching tokens to requests), and is done only by the servers. However, removing the client involvement requires the servers to look up the matrix for each request to assign it to one of the requesting client's buckets. Although the matrix size may be manageable, it is still proportional to the number of clients. Investigation of how to remove client involvement in bucket assignment is a subject of future research.

Meanwhile, we deal with this attack by clearing the client with a *clearing probability* ρ_{algo} for each of the client's assigned buckets found to be not attacked. In other words, the algorithm excludes a client from the suspects list if "enough" buckets assigned to the client are not attacked, instead of just one bucket. The false positive (FP) and false negative (FN) probabilities in this case are as follows.

$$FP = [1 - \frac{1}{d+1}(1 - \frac{1}{d+1} \cdot \rho_{attack})^d \cdot \rho_{algo}]^T \quad (6)$$

$$FN = 1.0 - [1 - \frac{1}{d+1}(1.0 - \rho_{attack}) \cdot (1 - \frac{1}{d+1} \cdot \rho_{attack})^{d-1} \cdot \rho_{algo}]^T \quad (7)$$

Our approach is to set the clearing probability to a fixed value that achieves acceptable false positive and false negative probabilities assuming a lower bound on ρ_{attack} . Fig. 5 (middle and right; note the two Y axes) shows the effect of changing ρ_{algo} when $\rho_{attack} = 0.9$ and 0.5 , respectively.

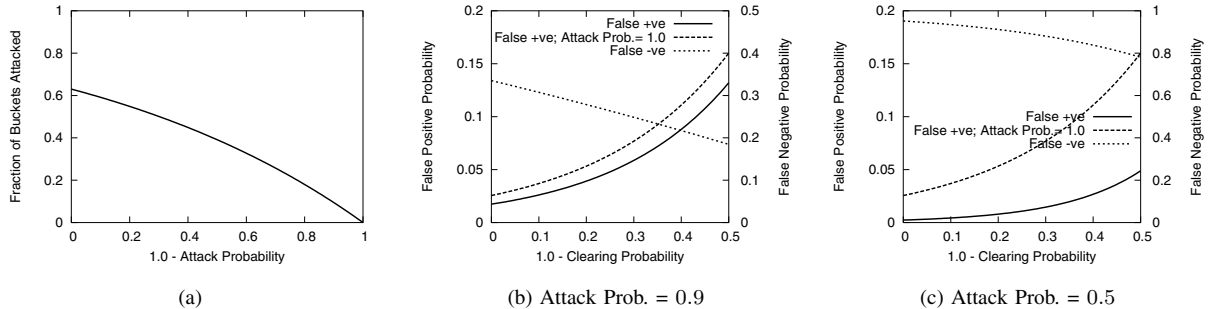


Fig. 5. Effect of clearing (ρ_{algo}) and attack (ρ_{attack}) probabilities .

VIII. STATE-OF-THE-ART IN DoS DETECTION

Although much work has been done in security, DoS attacks, intrusion detection, authentication, group-testing, and other related work, we only describe here work that is directly related to our live baiting algorithm. DoS defense approaches can be classified into prevention, detection and recovery, and mitigation. Mirkovic *et al.* has an extensive taxonomy of DDoS attack and defense techniques [30].

There are three attacker detection approaches, namely *misuse detection*, *anomaly detection*, and *specification-based detection*. Honeypots are effective detection tools against many DoS attack instances [10]. For example, in shadow honeypots, suspected requests are sent to instrumented servers to detect attacker attempts to exploit vulnerabilities [31]. This helps reduce the false alarm rate of intrusion detection systems. Sekar *et al.* developed a triggered, multi-staged automatic DDoS detection system for a large ISP to accurately discover customers under attack [32]. Kandula *et al.* has improved CAPTCHA-based defenses by detecting attackers that keep sending wrong solutions at a high rate [25]. Hussain *et al.* developed a detection algorithm based on spectral analysis to detect repeated attacks, e.g., attacks from the same set of attackers. Ranjan *et al.* enumerate a set of abnormal request behaviors that DDoS attackers use and propose a session scheduling algorithm based on suspicion level [33] that needs to keep state for each session. Our live baiting scales to the number of sessions by using an amount of state proportional only to the number of attackers, and it does not require models of legitimate sessions.

Estan and Varghese proposed an elegant algorithm to detect “heavy-hitter” network flows with low memory requirements to suit in-router implementation [5]. Their sample-and-hold technique is proposed for network-level detection and requires operators to set a threshold to define heavy-hitters. For instance, a heavy hitter may be defined as any flow that uses 10% or more of link capacity. We define a heavy hitter to be the client that causes overflows in its buckets. Although we require a threshold (HWM), in terms of number of request drops, our threshold is easier to design than sample-and-hold in the context of DoS detection. This relative ease is because in a well-provisioned server, there are typically very few request drops under no attack, so our threshold can be set to a small

number once and for many services, while a sample-and-hold threshold would require knowledge about legitimate client behavior for each service.

Chuah *et al.* [7] proposed an \sqrt{N} distributed algorithm for detecting flows that deviate from specifications. Jin *et al.* proposed the hop-count filter to detect and block packets with spoofed source addresses [34]. Kargl *et al.* proposed using class-based queuing and traffic monitoring to protect Web servers from DDoS attacks [3]. Their results (among results from other researchers) highlight the challenges introduced by service-level attacks, which target the CPU, and are harder to detect because they do not trigger any network-traffic alarms. Live baiting efficiently handles the service-level DoS attack.

In capability-based schemes, such as [35], [36], attack victims recover by revoking access from detected attackers. For instance, Kreibich *et al.* proposed that in-network filters enforce packet symmetry of passing flows, and the victim server blocks traffic from detected attackers by simply not replying to them [37]. Traceback is an approach for recovery from spoofing attacks (e.g., [38], [39]). Recovery schemes depend on the accuracy of attack detection. For instance, in-network filtering schemes (e.g., Pushback [40] and Max-min-fair throttling [41]) would not distinguish highly-diffused DoS attackers from legitimate clients and would propagate filters to block traffic from both [40].

IX. CONCLUSIONS

In this paper, we presented a novel approach for efficiently detecting service-level Denial-of-Service (DoS) attackers. Our solution, *live baiting*, is based on a novel application of group testing theory. Live baiting is a scalable scheme that can efficiently and quickly detect DoS attackers in a large service with millions of clients even when attackers send service requests that are indistinguishable from legitimate requests. We presented results from an analysis of the scheme effectiveness, in terms of false positive and false negative probabilities, and efficiency, in terms of memory, message, and computational complexity. Moreover, we introduced an adaptive method (which is also a contribution to the general group-testing theory domain) for discovering the actual number of attackers starting from an initial estimate. We validated our results using NS-2 simulations and presented techniques to extend our scheme to more complex service and attack models.

Our focus in this paper is on the presentation of live baiting via a proof-of-concept evaluation. When applied to web services, we were able to detect DoS attackers efficiently, with the assumption that clients have unique, un-spoofable IDs, such as cookies. The investigation of how to tune the scheme parameters, such as the detection interval, the high water mark, and the clearing probability, is left for future work. Although we described our solution in the context of attacks against servers (service-level attacks) our approach can be applied to protecting network-level resources as well. The investigation of this application is a subject of future work.

REFERENCES

- [1] Virgil D. Gligor, "A note on denial-of-service in operating systems," *IEEE Trans. Software Eng.*, vol. 10, no. 3, pp. 320–324, 1984.
- [2] Virgil D. Gligor, "On denial-of-service in computer networks," in *Proceedings of the Second International Conference on Data Engineering*, Washington, DC, USA, 1986, pp. 608–617, IEEE Computer Society.
- [3] Frank Kargl, Joern Maier, and Michael Weber, "Protecting web servers from distributed denial of service attacks," in *WWW '01: Proceedings of the 10th international conference on World Wide Web*, New York, NY, USA, 2001, pp. 514–524, ACM Press.
- [4] Virgil D. Gligor, "Guaranteeing access in spite of distributed service-flooding attacks," in *Security Protocols Workshop*, 2003, pp. 80–96.
- [5] Cristian Estan and George Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 270–313, 2003.
- [6] Rangarajan Vasudevan, Z. Morley Mao, Oliver Spatscheck, and Jacobus van der Merwe, "Reval: A tool for real-time evaluation of ddos mitigation strategies," in *Proceedings of the 2006 USENIX Annual Technical Conference*, 2006.
- [7] Chen-Nee Chuah, Lakshminarayanan Subramanian, and Randy H. Katz, "Dcap: detecting misbehaving flows via collaborative aggregate policing," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 5, pp. 5–18, 2003.
- [8] Andrew Sterrett, "On the detection of defective members of large populations," *The Annals of Mathematical Statistics*, vol. 28, no. 4, pp. 1033–1036, Dec. 1957.
- [9] Robert Dorfman, "The detection of defective members of large populations," *The Annals of Mathematical Statistics*, vol. 14, no. 4, pp. 436–440, Dec. 1943.
- [10] The HoneyNet Project, *Know Your Enemy*, Addison-Wisley, Indianapolis, IN, 2002.
- [11] N. Weiler, "Honeypots for distributed denial-of-service attacks," in *Proceedings of WET ICE 2002*.
- [12] John Levine, Richard LaBella, Henry Owen, Didier Contis, and Brian 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*.
- [13] Balachander Krishnamurthy, "Mohonk: mobile honeypots to trace unwanted traffic early," in *NetT '04: Proceedings of the ACM SIGCOMM workshop on Network troubleshooting*, New York, NY, USA, 2004, pp. 277–282, ACM Press.
- [14] Sherif M. Khattab, Chatee Sangpachatanaruk, Daniel Mossé, Rami Melhem, and Taieb Znati, "Roaming Honeypots for Mitigating Service-level Denial-of-Service Attacks," in *ICDCS 2004*.
- [15] Sherif Khattab, Rami Melhem, Daniel Mossé, and Taieb Znati, "Honey-pot back-propagation for mitigating spoofing distributed denial-of-service attacks," *Journal of Parallel and Distributed Computing (JPDC) Special Issue on Security in Grid and Distributed Systems*, vol. 66, no. 9, pp. 1152–1164, Sept 2006.
- [16] Luis von Ahn, Manuel Blum, and John Langford, "Telling humans and computers apart automatically," *Commun. ACM*, vol. 47, no. 2, pp. 56–60, 2004.
- [17] M. T. Goodrich, M. J. Atallah, and R. Tamassia, "Indexing information for data forensics," in *Proceedings of the Third International Conference on Applied Cryptography and Network Security (ACNS)*, J. Ioannidis, A. Keromytis, and M. Yung, Eds., 2005, pp. 206–221.
- [18] Benny Chor, Amos Fiat, and Moni Naor, "Tracing traitors," in *CRYPTO '94: Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, 1994, pp. 257–270, Springer-Verlag.
- [19] Annalisa De Bonis and Ugo Vaccaro, "Constructions of generalized superimposed codes with applications to group testing and conflict resolution in multiple access channels," *Theor. Comput. Sci.*, vol. 306, no. 1-3, pp. 223–243, 2003.
- [20] Yvo Desmedt, Rei Safavi-Naini, Huaxiong Wang, Lynn Batten, Chris Charnes, and Josef Pieprzyk, "Broadcast anti-jamming systems," *Comput. Networks*, vol. 35, no. 2-3, pp. 223–236, 2001.
- [21] Hung Q. Ngo and Ding-Zhu Du, "A survey on combinatorial group testing algorithms with applications to dna library screening," *Discrete mathematical problems with medical applications (New Brunswick, NJ) DIMACS Ser. Discrete Math. Theoret. Comput. Sci., Amer. Math. Soc.*, vol. 55, pp. 171–182, 2000.
- [22] D.Z. Du and F.K. Hwang, *Combinatorial Group Testing and its Applications*, World Scientific, Singapore, 2nd edition, 2000.
- [23] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis, "My botnet is bigger than yours (maybe, better than yours): Why size estimates remain challenging," in *Proceedings of USENIX/HotBots*, April 2007.
- [24] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich, "Flash crowds and denial of service attacks: characterization and implications for cdns and web sites," in *WWW '02: Proceedings of the 11th international conference on World Wide Web*, New York, NY, USA, 2002, pp. 293–304, ACM Press.
- [25] Srikanth Kandula, Dina Katabi, Matthias Jacob, and Arthur W. Berger, "Botz-4-Sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds," in *2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.
- [26] "Live Baiting for Service-level DoS Attackers," <http://www.cs.pitt.edu/~skhattab/livebait.pdf>, April 2006.
- [27] Federal Information Processing Standards Publication 180-2, "Secure hash standard," August 2002.
- [28] "The Network Simulator - ns-2," <http://www.isi.edu/nsnam/ns/>.
- [29] J. Cao, W.S. Cleveland, Y. Gao, K. Jeffay, F.D. Smith, and M.C. Weigle, "Stochastic models for generating synthetic http source traffic," in *IEEE Infocom 2004*, March 2004, vol. 3, pp. 1546–1557.
- [30] Jelena Mirkovic, Janice Martin, and Peter Reiher, "A Taxonomy of DDoS Attacks and DDoS Defense Mechanisms," Tech. Rep. 020018, Computer Science Department, University of California, Los Angeles, 2002.
- [31] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis, "Detecting targeted attacks using shadow honeypots," in *Proceedings of the 14th USENIX Security Symposium*, 2005.
- [32] Vyas Sekar, Nick Duffield, Oliver Spatscheck, Jacobus van der Merwe, and Hui Zhang, "Lads: Large-scale automated ddos detection system," in *Proceedings of the 2006 USENIX Annual Technical Conference*, 2006.
- [33] S. Ranjan, R. Swaminathan, M. Uysal, and E. Knightly, "Ddos-resilient scheduling to counter application layer attacks under imperfect detection," in *Proceedings of the IEEE Infocom*, Barcelona, Spain, April 2006, IEEE.
- [34] Cheng Jin, Haining Wang, and Kang G. Shin, "Hop-count filtering: an effective defense against spoofed DDoS traffic," in *Proceedings of the 10th ACM conference on Computer and communication security*, 2003, pp. 30–41.
- [35] Avi Yaar, Adrian Perrig, and Dawn Song, "An endhost capability mechanism to mitigate DDoS flooding attacks," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.
- [36] Katerina Argyraki and David Cheriton, "Network capabilities: The good, the bad and the ugly," in *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets-IV)*, 2005.
- [37] Christian Kreibich, Andrew Warfield, Jon Crowcroft, Steven Hand, and Ian Pratt, "Using packet symmetry to curtail malicious traffic," in *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets-IV)*, 2005.
- [38] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson, "Practical network support for IP traceback," in *ACM SIGCOMM 2000*.
- [39] Steven M. Bellovin, Marcus Leech, and Tom Taylor, "ICMP Traceback Messages," in *draft-ietf-itrace-01.txt, internet-draft, October 2001. Expired draft*.
- [40] Ratul Mahajan, Steven M. Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker, "Controlling high bandwidth aggregates in the network," vol. 32, no. 3, pp. 62–73, 2002.
- [41] David K. Y. Yau, John C. S. Lui, and Feng Liang, "Defending Against Distributed Denial-of-service Attacks with Max-min Fair Server-centric Router Throttles," in *Proceedings of the IEEE International Workshop on Quality of Service (IWQoS)*, May 2002.