Evaluating the Vulnerability of Network Mechanisms to Sophisticated DDoS Attacks

Udi Ben-Porat  
Tel-Aviv University, Israel  
Email: ehudben@post.tau.ac.il

Anat Bremler-Barr  
Interdisciplinary Center, Herzliya, Israel  
Email: bremler@idc.ac.il

Hanoch Levy  
Tel-Aviv University, Israel  
Email: hanoch@post.tau.ac.il

Abstract— The design of computer and communication systems has been based, for decades, on the fundamental assumption that the objective of all users is to improve their own performance. In recent years we have experienced a wave of DDoS attacks threatening the welfare of the internet. These are launched by malicious users whose pure incentive is to degrade the performance of other, innocent, users. The traditional systems turn out to be quite vulnerable to these attacks.

The objective of this work is to take a first step to close this fundamental gap, aiming at laying a foundation that can be used in future computer/network designs taking into account the malicious users. Our approach is based on proposing a metric that evaluates the vulnerability of a system. We then evaluate the commonly used data structure in network mechanisms, the hash data structure, using our vulnerability metric. We show that a Closed Hash is much more vulnerable than an Open Hash to DDoS attacks, even though the two systems are considered to be equivalent via traditional performance evaluation. We also apply the metric to queueing mechanisms common to computer and communications systems. Lastly we apply it to the practical case of a hash table whose requests are controlled by a queue, showing that even after the attack has ended, the regular users still suffer from performance degradation or even a total denial of service.

I. INTRODUCTION

In recent years the welfare of the Internet has been threatened by malicious attacks of distributed denial of service (DDoS). DDoS attackers consume the resources of the victim, a server or a network, causing a degradation in performance or even total failure of the victim. In DDoS attacks, attackers send a large amount of traffic, consuming the CPU or the bandwidth of the victim1. The ease with which such attacks are generated makes them very popular. According to a study, large backbone networks observe DDoS attacks on a daily basis [1].

The basic DDoS attack is a simple brute force flooding, where the attacker sends as much traffic as he can to consume the network resources, namely the bandwidth of the server’s incoming link, without any knowledge of the system design. This can be achieved by sending a huge amount of traffic over a raw socket; or by using an army of Zombies2 each mimicking the requests of a regular user.

More sophisticated attacks try to increase their effectiveness by aiming at hurting a weak point in the victim’s system design, i.e. the attacker sends traffic consisting of complicated requests to the system. An example of such attacks is an attack against http servers by requesting pages that are rarely requested (forcing the server to search in the disk). Similar attacks can be conducted on search engines or on DataBase servers by sending difficult queries that enforce them to spend much CPU time or disk access time. In fact, an example of such a sophisticated attack can be seen even in the classic SYN [2] attack which aims to hurt the TCP stack mechanism at its weakest point, the three-way-handshake process and its corresponding queue.

In this paper we concentrate on sophisticated attacks. We define Sophisticated DDoS attacks as attacks that increase their effectiveness by aiming at hurting a weak point in the victim’s system design, i.e. the attacker sends traffic consisting of complicated requests to the system. This generalized definition covers all the different attack types described in the recent papers [3], [4], [5], [6], [7], [8], [9] including the two new major breeds: the Complexity Attack [3] and the Reduction of Quality Attacks [5], [6]. In section II we give a short overview on the different DDoS attack types that fall into the category of sophisticated attacks.

Our first contribution in this paper (in Section III) is a proposal of a new metric that evaluates the vulnerability of a system for any kind of sophisticated attacks. Our approach is based on proposing a Vulnerability Factor that accounts for the maximal performance degradation (damage) that sophisticated malicious users can inflict on the system using a specific amount of resources (cost) normalized by the performance degradation attributed to regular users using the same resources. Having a metric for computer security [10] is an important step for improving security. As a well known phrase states : “If you cannot measure, you cannot manage. If you cannot manage, you cannot improve”. Our metric provides the ability to understand the resilience of systems to DDoS and, in the case that multiple algorithms/designs are available for the same problem, helps in selecting the most resilient one.

1While the terminology is close and somewhat misleading, we note that our paper does not deal with DoS attacks, where the attacker is able to attack by sending only a handful of packets, exploiting a bug in the application implementation or design (e.g. ping of death). We focus on DDoS where the attackers exploit the valid characteristics and rules of the system protocol and just send a large volume of traffic in order to consume the resources.

2A zombie is a compromised machine that is controlled by a remote hacker. The current estimate is that there are millions of machines today, in private homes and institutes, that are zombied.
Vulnerabilities to sophisticated DDoS attacks seem to be an inherent part of existing computer and network systems. Traditionally, these systems have been designed and operated under the underlying fundamental principle that each user aims at maximizing the performance he/she receives from the system. While operational strategies have been divided to social optimization (optimize operation to the benefit of the overall population) or individual optimization (each user acts to improve its performance), see e.g. [11], the basic paradigms were still based on this principle, namely that each user aims at maximizing its own performance. This paradigm does not hold any more in the new "DDoS environment" where some users do not aim to maximize their own performance, but rather to degrade the performance of other users. Thus, there is a major gap between traditional assumptions on user behavior and practical behavior in modern networks. Due to this gap it is both natural and inherent that traditional algorithms, protocols and mechanisms will be quite vulnerable in this environment.

Our second contribution, in Section V, is evaluating the Hash data structure, commonly used in network mechanisms, using our vulnerability metric. Using our vulnerability metric we show that a Closed Hash is much more vulnerable than an Open Hash. This demonstrates the importance of the metric as it indicates a major performance gap between two systems which are considered to be equivalent via traditional performance evaluation.

Our third contribution, in Section VI, is discussing the Vulnerability of Queueing policies, commonly used in network mechanisms. We demonstrate the issue by considering a very simple example of the First Come First Served (FCFS) queue. We show that though attackers, under this mechanism, "must wait like everyone else" and thus have no control of their place in the queue, they can still cause significant damage by using the same resources (traffic) as regular users. The strategy of the malicious users is simple - just submit large jobs from time to time. In fact the vulnerability level (using the proposed metric) of this system can be unbounded.

Our fourth contribution, in Section VII, shows that in the case of an attack on either Open or Closed Hash, the regular user still suffers from performance degradation or even a total denial of service, even after the attack has ended (note that the degradation is not due to the extra load of the attack but due to its sophistication). As far as we are aware, we are the first to point out post attack performance degradation. We demonstrate it by using the analysis of the practical case that combines the Hash table with a queue preceding it (post-attack degradation is demonstrated, for closed hash, also in Section V). Using this model, we further derive the precise size of the attack that will "harm" the Hash in such a way that, even after the attack has ended, it will make the regular users suffer from complete denial of service (and not only from performance degradation).

We conclude the paper by comparing the vulnerability of the open and closed hash (Section VIII) and a discussion (Section IX).

II. Sophisticated Attacks

We define Sophisticated DDoS attack as an attack in which the attacker sends traffic aiming to hurt a weak point in the system design in order to conduct denial of service3. The sophistication lies in the fact that the attack is tailored to the systems design, aiming to increase the effectiveness of the attack.

The motivation of the attacker is to minimize the amount of traffic it sends while achieving the same or even better results. Using sophisticated attacks the attacker reduces the cost of attacks i.e., reduces the number of required zombies4 and reduces the sophistication in coordinating the attack. Moreover, the use of sophisticated attacks increases the likelihood that the attack will succeed in going unnoticed (going Under the Radar) by the DDoS mitigation mechanisms, which are usually based on detecting the fact that the victim was subject to higher-than-normal traffic volume. Note that designing a sophisticated attack tool requires only a one-time effort of understanding the system design in order to find its weak point. However, after such a tool is constructed, other attackers can use the tool as an "off-the-shell" black box attack tool without any knowledge of the system. Similar evolutions were observed in other DDoS attack types such as "syn-attack" where general tools such as trinoo and tnf were designed and distributed over the web to layman attackers[12].

Due to the complexity of the applications, they are usually more vulnerable to Sophisticated DDoS attacks. However, the Sophisticated attacks are not just against applications and may also be conducted against protocols (see, for example, attacks against TCP [7]).

Roughly speaking, we can classify the methods of launching sophisticated attacks into three groups, based on the weak design point exploited by the attacker: Worst-Case Exploit, Traffic Pattern Exploit and Protocol Deviation exploit.

1) Worst-Case Exploit or Complexity attack [3], [13] - Attacker exploits the worst-case performance of the system which differs from the average case that the system was designed for. Paper [3] demonstrates the phenomenon on the commonly used Open Hash data structure, where attacker can design an attack that achieves the worst case of $O(n)$ complexity per insert operation instead of the average case of $O(1)$. Complexity attack problems were described in many different algorithms, such as quicksort [14] regular expression matcher [15], intrusion detection systems [16], [9] and the linux route-table cache [17].

2) Traffic Pattern Exploit - Attacker exploits the stochastic worst case traffic pattern to the system. This case is similar to the first one with the difference that the worst case scenario involves a specific traffic arrival pattern of requests from multiple users. This type of attack is demonstrated in the

3or just to significantly degrade the performance (such as Reduction of Quality attacks like [5], [16])

4Today, there is a market in zombie machines. Attackers can skip the time consuming process of taking control of new machines and just buy "off the shelf" zombies. Hence the number of machines required to launch the attack translate directly to the cost of money.
reduction of quality attack papers [5], [6], [9]. RoQ attacks target the adaptation mechanisms by hindering the adaptive component from converging to steady-state. This is done by sending, from time to time, a very short duration of surge demand imitating many users and thus pushing the system into overload condition. Using a similar technique the paper [7] presented the Shrew Attack [7] which is tailored and designed to exploit TCP’s deterministic retransmission timeout mechanism. Another example of an attack exploiting the stochastic worst case was presented in [4] where it is shown that WFQ, a commonly deployed mechanism to protect against DDoS attack, is ineffective in the environment of bursty applications including the Web application.

3) Protocol Deviation Exploit - Attacker forms his own protocol rules, exploiting the fact the protocol was designed using the assumption that all the users obey the rules of the protocol. We are not aware of previous papers that deal with this method of attack, however we have started investigating this method, using the example of Multiple Access protocol (and this is part of our future work). Multiple Access protocols [18] deal with the scenario of a shared channel, where a set of nodes send and receive frames over the same channel, while only one node can transmit at a time. In this type of algorithm each node runs some type of collision avoiding algorithm. In Sophisticated attacks the motivation of the attacker is to disturb the transmission over the channel as much as possible, and hence he runs his own protocol (not obeying the collision avoiding algorithm) with his own sets of rules which are designed to reduce the performance of the network the most, taking into account the fact that all other users obey the protocol rules.

III. THE VULNERABILITY FACTOR

Our work is based on the observation that the traditional performance analysis of computer systems and algorithms is not adequate to analyze the vulnerability of a system to sophisticated DDoS attack and a new metric is needed. Note that the analysis of the worst-case of a system versus that of the average-case is not sufficient in order to comprehend the vulnerability of a system. This is due to three major factors: First the worst case analysis does not take into consideration the cost of producing the worst case scenario. However, in the context of analyzing the vulnerability of a system, the cost is an important factor, since a scenario that is expensive to produce, does not really pose a threat to the system. Second, traditionally the assumption in the worst case analysis of a system is that all the participants follow the protocol rules. However, dealing with a malicious attacker, the attacker can form his own protocol, mainly designed to damage the system as much as possible. Third, usually the worst case analysis deals with the most difficult input to the system per user. However in the vulnerability assessment we also need to take into account the most difficult scenario of interaction between the users of the system. I.e., we also need to analyze the system using the ”stochastic” worst case scenario (as shown in the sophisticated attacks [5], [7]).

Our proposal for the definition of the Vulnerability Factor of a system is to account for the maximal performance degradation (damage) that sophisticated malicious users can inflict on the system using a specific amount of resources (cost) normalized by the performance degradation attributed to regular users using the same resources. Figure 1 demonstrates the concept of the Vulnerability Factor. Mechanisms that are vulnerable only to brute force attacks will have a Vulnerability Factor of one, while mechanisms that are vulnerable to sophisticated attacks will receive a factor of $K > 1$ indicating that the maximal attacker’s power is equivalent to that of $K$ regular users.\(^5\)

Formally, we define the Vulnerability Factor as follows: Let $S$ be a system in its regular status experiencing usual traffic (load). Let the $\text{usersType}$ parameter be equal to either regular users ($R$) or malicious attackers ($M_{st}$) with strategy $st$. Note that we use the plural terms since some of the attack types occur only in specific scenarios of multiple coordinated access of users to the system [5]. Let the $\text{budget}$ be the amount of resources that the users of $\text{usersType}$ spend on producing the additional traffic to the system, i.e., the cost of producing the attack is limited by the $\text{budget}$ parameter. The $\text{budget}$ can be measured differently in the various models, e.g. as the required bandwidth, or as the number of required computers or as the required amount of CPU and so on. Let $\Delta \text{Perf}_{S}(\text{usersType}, \text{budget})$ be the change in the performance of system $S$ due to being subject to additional traffic added to the regular traffic of the system, where the traffic is generated by users from type $\text{usersType}$ with resource $\text{budget}$. The performance can be measured by different means such as the CPU consumption, the delay experienced by a regular user, as the number of users the system can handle, and so on.

We define the Effectiveness of a strategy $st$ on system $S$ as

$$E_{st,S}(\text{budget} = b) = \frac{\Delta \text{Perf}_{S}(M_{st}, b)}{\Delta \text{Perf}_{S}(R, b)},$$

and the Vulnerability Factor $V$ of a system $S$ as:

$$V_{S}(\text{budget} = b) = \max_{st} E_{st,S}(b).$$

\(^5\)Putting the DoS attack on the same baseline, the DoS attack (bug of application that caused total failure) receives the factor of infinity.
A strategy \( st \) is said to be an **Optimal Malicious Users Strategy** for system \( S \) if the effectiveness of strategy \( st \) on \( S \) is equal to the **Vulnerability Factor** of \( S \). Note that there can be more than one **Optimal Malicious Users Strategy** and that an **Optimal Malicious Users Strategy** maximizes \( \Delta P_{e} \mu_{S}(M_{st}, b) \).

Note that we focus on the attacker's strategy that has the maximum impact in performance degradation, since the common assumption in the field of security is that the vulnerability of a system is measured as its weakest point.

**IV. Related Work**

The first step for measuring the vulnerability of system, was done in paper [5]\(^9\), concentrating only on measuring reduction of quality attacks. The paper suggests the measure of **attack potency** by evaluating the amount of performance degradation inflicted by an attacker with specific budget. The reader can observe that this measure is similar to our definition \( \Delta P_{e} \mu_{S}(M_{st}, b) \), and in fact the difference between the Effectiveness and the Potency measures, is that we normalize the performance degradation the system suffers due to an attacker by the performance degradation the system suffers due to a regular user. Clearly, the potency parameter can be generalized to measure any sophisticated attack, however we think that the **Effectiveness measure** is preferable since it allows to derive meaningful information based on this number alone. Consider the case of a system with Vulnerability Factor of \( K \). In this case we can deduce that the system can cope with a number of sophisticated attackers, which is only \( \frac{1}{N} \) of the number of regular users the system can handle. In contrast, if the potency level is \( P \) it means that the attacker obtains \( P \) units of damage per one unit of cost; this number is not meaningful without comparing it to the potency of other scenarios, such as the potency of a regular user, or the potency of other attacks. Moreover, the unit of measure (damage per cost) is not fully defined and left for the interpretation of whoever uses it.

Both the Potency and the Effectiveness concentrates on the power of a specific attack. We suggest that the more interesting metric is the **Vulnerability Factor** that analyzes the system and give bounds on its performance under any type of attack.

In the paper we present results relating to the vulnerability of Hash systems. The sophisticated attack against Hash algorithm was first introduced in [3] however, the paper concentrated only on Open Hash, using only experimental results on real-life Hash implementations. As opposed, we provide analytic results using our **Vulnerability Factor** metric on the two Hash types: Open Hash and Closed Hash. An important finding of this simple analysis, which can affect practical designs, is that these two common Hash types (which are known to be equivalent according to many performance aspects) drastically differ from each other in their vulnerability to sophisticated attacks. Moreover our paper, by analyzing the effect of queueing with Hash, shows that the regular users suffer from performance degradation also after the attack has ended.

**V. Sophisticated Attack on Hash Tables**

Many network applications, such as the Squid web proxy and the Bro intrusion detection system, use the common Hash data structure. This data structure supports the dictionary operations of Insert, Search and Delete of elements according to their keys. A Hash table is used when the number of keys (elements) actually stored is small relative to the total number of possible key values. A Hash is based on an array of size \( M \), where an entry in the array is called a **bucket**. The main idea of a Hash is to transform the key \( k \) of an element, using a Hash function \( h \), into an integer that is used as an index to locate a bucket (corresponding to the element) in the array. In the event that two or more keys are hashed to the same bucket, a collision occurs and there are two Hash strategies to handle the collision: Open Hash and Closed Hash. In an Open Hash each bucket in the array points to a linked list of the records that were hashed to that bucket. In a Closed Hash all elements are stored in the array itself; In the insert operation the array is repeatedly probed until an empty bucket is found. The Hash function in this case can be thought of as being extended to be a two parameter function depending on the key \( k \) and the number of the probe attempts \( i \) i.e. \( h(k, i) \) (for detailed explanations about Open and Closed Hash see [20]). The particular probing mechanism where \( h(k, i) = H(k) + i - 1 \) is called **linear probing**.

In the Hash data structure (Closed or Open), the average complexity of performing a Hash operation (insert, delete, find) is \( O(1) \) elementary operations\(^7\) while the worst case complexity is \( O(N) \) elementary operations per Hash operation, where \( N \) is the number of elements in the Hash table. The insert operation consists of conducting a search to check whether the element already exists in the Hash and, in the case it does not exist, of inserting the new element. In both Closed and Open Hash the insert operation of a new element, is the most time consuming operation.

In [3] it is shown that an attacker, in the common case, where the Hash function is publicly known\(^8\), can drive the system into the worst case performance, and thus dramatically degrade the performance of the server application\(^9\). Naturally, the Hash example belongs to the group of **Worst-Case Exploit** sophisticated attack.

---

\(^9\)The precise complexity is a function of the load in the Hash.

\(^8\)Due to the Open source paradigm.

\(^7\)Universal Hash functions were introduced [21] and are cited as a solution suitable for adversarial environments. However, it has not been standard practice to follow this advice (mainly due to fear of unacceptable performance overheads in critical regions of code). Moreover, Tim Peters and Solar Designer claim [13] that it may be possible to force the worst case scenario even in the case of universal hashing, by statistically identifying whether two items collide or not, using the detectable processing latency. Another counter-measure is to randomize the Hash Table. Paper [22] shows that there is an attack that can overcome this solution.
A. Model

Consider a Hash system consisting of $M$ buckets and $N$ elements in the Hash, where the Hash function is known to the attackers. In this model the resource-budget of the users is measured by the number of Hash operations ($k$) they can perform, where Hash operations can be either insert, delete or search. In our analysis we will use three metrics: i) The number of memory references the $k$ operations require in attack time (In Attack Resource Consumption) and ii) The complexity of performing an arbitrary Hash operation after the attack has ended, i.e., after these $k$ Hash-operations are completed (Post Attack Operation Complexity). Note that both metrics, (i) and (ii), are measured by the number of memory references\footnote{It is common practice to count memory references in network algorithms since memory is the bottleneck in most of the systems.} iii) the waiting time of a regular user while preforming an arbitrary Hash Operation after the attack has ended, i.e., after the $k$ Hash-operations are completed (Post Attack Waiting Time). The analysis of the third metric is in Section VII.

In order to carry out the analysis one needs to decide on what type of operations to base the computation the denominator of Equation 1, that is $\Delta \text{Perf}_S(R, b)$. In analyzing the Vulnerability Factor we take the conservative approach and assume that the regular users effect the system by conducting the most time consuming operation types. In both Closed and Open Hash this is the insert operation. Thus we receive a lower bound on the actual Vulnerability Factor. Note also that as opposed to the traditional elementary operation complexity analysis in the Vulnerability Factor analysis we are also interested into the specific constant and not only in the general trend. For ease of presentation we define $x(1)$ and $x(2)$ as the first and second moments of a random variable $X$.

B. Insert Strategy

Let the Insert Strategy (INS) be the malicious users’s strategy of performing $k$ insert operations of new elements where all the insert elements hash into the same bucket.

Note that in order to create Distributed DoS attacks, using the Insert Strategy multiple attackers only need to share one bit of information - the bucket that they wish to insert the elements to.

Lemma 1: In Open Hash and Closed Hash systems, the Insert Strategy is an Optimal Malicious Users Strategy under the In Attack Resource Consumption metric and Post Attack Operation Complexity metric.

Sketch of Proof: In the In Attack Resource Consumption metric the attacker preforms the operations that require the highest time\cite{20}. In both Open and Close Hash the time consuming operation is the insert operation and in both cases the highest time is achieved if all the elements where inserted into the same bucket, thus the search phase in the insert operation of the new element, requires to review at least all the previous inserted elements in the attack. See full proof in Appendix X. In Post Attack Operation Complexity metric, the attackers aim to preform an attack that after the attack is ended, the status of the Hash is such that regular user operations, will take the maximum time. In order to achieve this the attackers need to preform an attack which result in the worst possible state of the Hash. In Open Hash, as we show later on, there is no such case, all the possible Hash states, require on average the same time complexity. In Closed Hash, this is achieved by creating a big cluster, a long run of occupied buckets, which is the result of the Insert Strategy. In both cases, the detailed proof is carried out by using induction on the $k$ malicious user operation.

C. Open Hash

In this subsection we analyze the vulnerability of the Open Hash table (denoted by OH). We define the load factor\footnote{This is the load factor as defined in the literature for data structures \cite{20}; it is different from the traffic load.} $L$ of the Open Hash to be equal to $\frac{k}{M}$, which is the average number of elements in a bucket. We assume each bucket is maintained as a linked list. The insert operation of a new element, which is the most time consuming operation in Hash, requires an expected number of memory references of $L + 1$ \cite{20}. The use of the Insert Strategy of the malicious users on the Open Hash creates a growing list of elements in one bucket.

1) In Attack Resource Consumption Metric:

Theorem 2: Under the In Attack Resource Consumption metric the Vulnerability Factor of the Open Hash is $V_{OH}(k) = \frac{k + 2L + 1}{M + 2L + 2}$

Proof: We prove that $\Delta \text{Perf}_{OH}(R, k) = k(1 + L + \frac{k + 1}{M})$

After the $i - 1$st insertion of regular users the load of the table will be $L + \frac{i - 1}{M}$, therefore the $i$th insertion is expected to require $1 + L + \frac{i - 1}{M}$ memory references, resulting in total of $k(1 + L + \frac{k + 1}{M})$ memory references over $k$ insertions.

Next we prove that $\Delta \text{Perf}_{OH}(M_{INS}, k) = kL + \frac{k(k + 1)}{2}$. The expected memory references which is required by the $i$th insertion of malicious users is $L + i$, sums to $kL + \frac{k(k + 1)}{2}$ memory references for the entire insertions sequence.

Remark: Simplifying the Vulnerability Factor, we deduce that parameter $k$, the magnitude of the attack, is the most dominant parameter that influence the Vulnerability.

2) Post Attack Operation Complexity Metric:

Theorem 3: Under the Post Attack Operation Complexity metric $V_{OH}(k) = 1$.

Proof: The proof follows from the fact that the expected length of an arbitrarily linked list in the Hash, after the insertion of the $k$ elements, is always $L + k/M$. This is regardless of the bucket chosen by any of these elements.

Remark: This characteristic of the Open Hash whereby the expected damage caused by malicious users equals that of regular users is an important advantage in terms of vulnerability. As we can see, this advantage is well expressed in the measurement proposed above.
D. Closed Hash

In this subsection we analyze the vulnerability of the Closed Hash table (denoted by CH). We define the load factor \( \alpha \) as the percentage of occupied buckets, i.e. \( \frac{N}{M} \). For the analysis we assume that collision resolution is done via linear probing; nonetheless, we will explain later that the result derived for the In Attack Resource Consumption also holds for more sophisticated collision resolution schemes, such as the commonly used double hashing probing. As in Open Hash, in a Closed Hash the most time consuming operation is the insertion of a new element. Each such insertion requires approximately on average \( \frac{1}{2\alpha} \) elementary operations where \( \alpha \) is the current load in the Hash (see [20]).

The Vulnerability Factor in Closed Hash is influenced by the clustering effect (which also complicates the analysis). In Closed Hash long runs of occupied buckets build up [20]. These clusters of buckets increase the average insert time.

1) In Attack Resource Consumption Metric: The sophisticated attackers optimal strategy is the Insert Strategy, i.e., insert different keys that hash into the same bucket and thus create a long cluster of occupied buckets. This attack creates a cluster that contains the \( k \) inserted elements plus the union of multiple clusters that existed before the attack.

Note that in this metric we can avoid making the assumption that a linear probing technique is used. Formally, the attackers search for a pool of keys, so that for every two keys \( k_1 \) and \( k_2 \) in the pool \( h(k_1,i) = h(k_2,i) \) for all \( i \) where \( h(k,i) \) is the Closed Hash function. A commonly used open addressing Hash function is the double hashing: \( h(k, i) = h(k) + ih'(k) \). It easy to see that even with this double hashing function the attackers can easily find such as \( k_1 \) and \( k_2 \) since they have the knowledge of \( h \) and \( h' \).

**Theorem 4:** Under the In Attack Resource Consumption metric the Vulnerability Factor of the Closed Hash is

\[
V_{CH}(k) = \left( \sum_{i=1}^{k} \sum_{t=i}^{i+N} \frac{N}{M} \right) \cdot \frac{1}{\sum_{i=1}^{k} \frac{1}{1-(\alpha+\frac{N}{M})}}
\]

Proof: From the complexity of an insert operation of a new element, we get that \( k \) insertions of regular users require:

\[
\Delta P_{perCH}(R, k) = \sum_{i=1}^{k} \frac{1}{1-(\alpha+\frac{N}{M})}
\]

Next we prove that:

\[
\Delta P_{perCH}(M_{INS}, k) = \sum_{i=1}^{k} \sum_{t=i}^{i+N} \frac{N}{M} \cdot i
\]

Let \( B_j \) denote the \( j \)-th bucket.

The Insert Strategy is an Optimal Malicious Users Strategy. The malicious users create a long cluster by inserting \( k \) elements which are all hashed into the same arbitrary bucket, which, w.l.o.g we assume is \( B_1 \). Let \( X_i \) be a random variable denoting the number of memory references used to insert the \( i \)-th element into the table. In addition to passing over the \( i-1 \) elements previously inserted by the malicious users, the \( i \)-th insertion may also need to pass over some of the already existing elements in the table, therefore \( i+N \geq X_i \).

Let us examine the table before the malicious users start their insertions: \( X_i = t \) iff \( B_i \) is the \( i \)-th free bucket of \( B_1, B_2, \ldots, B_t \). This occurs iff \( B_1, \ldots, B_t \) contain exactly \( t-i \) elements which none of which is in \( B_i \). Therefore \( P(X_i = t) = P(Y_t = t-i) \cdot P(\text{empty} | Y_t = t-i) \) where \( Y_t \) is a random variable denoting the number of elements residing in \( B_1, \ldots, B_t \). \( Y_t \) follows the hypergeometric distribution with parameters \( M, N \) and \( t \) and thus, for \( i \leq t \):

\[
P(X_i = t) = \binom{N}{t-i} \binom{M-t}{N-t} \cdot \frac{1}{t}
\]

and \( P(B_t \text{ is empty} | Y_t = t-i) = \frac{i}{t} \) and we get that:

\[
P(X_i = t) = \binom{N}{t-i} \binom{M-t}{N-t} \cdot \frac{i}{t}
\]

The expected total complexity of the malicious users, \( \Delta P_{CH}(M_{INS}, k) \), is given by:

\[
E(\sum_{i=1}^{k} X_i) = \sum_{i=1}^{k} X_i \cdot \frac{1}{t}
\]

where \( X_i = \sum_{t=1}^{i+N} P(X_i = t) \cdot t \). And hence we get the required Equation.

Remark: Simplifying the Vulnerability Factor expression, we receive, similar to the Open Hash case, that the parameter \( k \) is the dominant parameter that influence the Vulnerability.

2) Post Attack Operation Complexity Metric: In Closed Hash the cluster that was created due to the attack, has a negative effect on the complexity of a user operation even after the attack has ended. Every insert element which is mapped into a bucket in the Cluster (not just specifically the bucket the attackers used) will need to search from its position to the end of the cluster [14]. Thus, unlike the case in Open Hash, the Vulnerability Factor will be greater than 1 under this metric, as the following theorem states:

**Theorem 5:** Under the Post Attack Operation Complexity metric the Vulnerability Factor of the Closed Hash is:

\[
\Delta V_{CH}(k) = \left( E[Q_M] - \frac{1}{N} \right) \left( \frac{1}{1-(\alpha+\frac{N}{M})} - \frac{1}{1-(\alpha+\frac{N}{M})} \right)
\]

where \( E[Q_M] = \sum_{c=k+1}^{N+k+1} \frac{1}{c+k} \) and after regular insertions it equals \( \frac{1}{N} \).

Proof:

As already mentioned, the expected insertion complexity to a normal closed hash table consisting \( R \) elements is given by \( \frac{1}{N} \), so we conclude that the expected complexity before the additional user insertions is \( \frac{1}{N} \), and after \( k \) regular insertions it equals \( \frac{1}{N} \).

We now calculate the performance degradation after the malicious insertion. This is retrieved from the following lemma 6, that calculate the complexity of operation priority after the malicious insertions. For \( d = 1 \), the value of 

\[
\sum_{s=1}^{N-k-c+2} P(Q_m = s| \text{Area B}) s^d
\]

can be replaced [20] by

\[
\frac{1}{N} \cdot \frac{1}{M}
\]
in order to produce a clearer expression as we used along the text. By doing so, we get the above value of $E[Q_M]$. Let us look at a closed hash table with $M$ bucket, $k \geq 1$ elements inserted by attacker (using the Inset Strategy) and $N$ existing elements before the attack. Let the random variable $1 \leq Q_M \leq N + k + 1$ be the complexity of one regular insertion into the table after the malicious insertions. In the following claim we calculate the value $E[Q_M^d]$. We will need to compute the 2nd moment of $Q_M$ in section VII.

**Lemma 6:** $E[Q_M^d] = \sum_{m=k+1}^{N+k+1} P(C = m) E[Q_M^d] = s|C = c$

\[
= \sum_{c=k+1}^{N+k+1} \left( \frac{N}{M-N-2} \right) \frac{k+2}{c+1} \left( \frac{c^2}{2M} \right) + (1 - \frac{1}{M}) \sum_{s=1}^{N-k-(c-1)(M-N-K-1-s)} \left( \frac{s}{M-c-s} \right)
\]

**Proof:** The malicious elements are all contained in one big chain of occupied buckets which may contain some or all of the exiting elements. If an element will be hashed into this chain, it will eventually rehashed into one of the two empty buckets trimming the chain. Define the malicious cluster as the chain (consisting the malicious insertions) of consecutive occupied buckets and the empty bucket that follows them (in the linear probing direction).

Let $1 \leq C \leq N + k + 1$ be the length of the malicious cluster (the number of elements in the chain is $C - 1$). An insertion can be hashed into the cluster area (Area A) or outside the cluster (Area B). With linear probing rehashing, it is impossible for an element to be hashed into one area and then rehash into the other.

The value of $Q_M$ for an insertion into Area A depends on the distance between the first bucket where the element was hashed and the empty bucket at the end of the cluster, therefore $P(Q_m = s|Area A, C = c) = \frac{1}{c}, (1 \leq s \leq C)$.

An insertion into Area B is equivalent to an insertion into a regular open hash table with the same number of buckets and elements as in Area B.

Therefore $\left( \prod_{i=1}^{s-1} \frac{N-i}{M-c-i} \right) = s|Area B, C = c = \left( \prod_{i=1}^{s-1} \frac{N-i}{M-c-i} \right) \cdot \frac{M-N-K-1-s}{M-c-s} (note that P(Q_m = 0|Area B) = \frac{M-N-K-1}{M-c-s})$.

When we combine these possibilities we get:

\[
E[Q_M^d] = s|C = c = \frac{c}{M} \sum_{s=1}^{s=c} P(Q_m = s|Area A) s^d + (1 - \frac{c}{M}) \sum_{s=1}^{s=c} P(Q_m = s|Area B) s^d
\]

Therefore the probability function of the cluster size is given by $P(C = c) = P(X_{k+2} = c + 1) = \frac{(\frac{c}{c+1})^{(\frac{N-k}{c+1})} k+2}{c+1}$.

Eventually we get:

\[
E[Q_M^d] = \sum_{s=c=k+1}^{N+k+1} P(C = c) E[Q_M^d] = s|C = c
\]

\[
= \sum_{c=k+1}^{N+k+1} \left( \frac{N}{M-N-2} \right) \frac{k+2}{c+1} \left( \frac{c^2}{2M} \right) + (1 - \frac{1}{M}) \sum_{s=1}^{N-k-(c-1)(M-N-K-1-s)} \left( \frac{s}{M-c-s} \right)
\]

**Remark:** In this case the parameters that influence the vulnerability are the second moment of $C$, the cluster size, and $M$. The Vulnerability is inversely proportional to $M$. This is the first time where the number of buckets play an important role in the Vulnerability.

VI. Sophisticated Attack on Queuing Mechanisms

Queueing mechanisms are major mechanisms used in variety of computer and communications systems. Their major objective is to control the system operation when it is under heavy traffic load in order to provide good and fair performance to the various users. Since these system are responsible for the resource allocation and taking care of the performance at times of high traffic load – it is natural that they become the target of attacks and their vulnerability should be examined.

To demonstrate the vulnerability of the queuing mechanism is sensitive to attacks, if not designed properly, we consider a simple case study consisting of a single server (with single queue) system that operates under the First-Come-First-Served (FCFS) scheduling. This analysis of FCFS is also an important step for us in order to evaluate the performance of applications that use Hash in the Internet (in Section VII). Consider the M/G/1 model with arrival rate $\lambda$ and service times distributed as random variable $X$ with first two moments $x(1)$ and $x(2)$ and the system utilization is $\rho = \lambda x(1)$. Let $W_{\text{scenario}}$ be the expected waiting time of an arbitrary arrival where scenario can be either of $C$, $R$ or $M$, denoting respectively the cases of i) The queue is subject only to normal load by regular arrivals (“clean”), ii) the queue is subject to additional load due additional regular arrivals, or iii) the queue is subject to additional load due to malicious arrivals. The performance of the system can be evaluated via the mean waiting time experienced by the jobs in equilibrium, which is given by $W_C = \lambda x(2)/(2(1 - \rho))$.

One way to attack a FCFS queue is to send large jobs. Consider attackers who behave like regular users who pose additional arrival rate $\lambda_1$ and whose job size is also distributed like $X$. Adding these users to the system changes the mean waiting time to $W_R(1) = (\lambda + \lambda_1) x(2)/(2(1 - \rho - \lambda_1 x(1)))$, and we have $\Delta \text{Perf}_{FCFS}(R, \lambda_1 x(1)) = (\lambda + \lambda_1) x(2)/(2(1 - \rho - \lambda_1 x(1))) - \lambda x(2)$. Now Consider attackers $A$ who send jobs of size $x = K x(1)$ at rate $\lambda_2 = \lambda_1/K$, that is the additional traffic load they pose is

\[15\text{The related graphs were built according to the exact expression.}
\[16\text{In In Attack Resource Consumption metric the number of buckets impact the load. However, the load parameter has only small impact on the vulnerability.}
identical to the additional load of the regular users, and thus the attack cost from this perspective is the same (identical budget). The delay in the system, nonetheless, now becomes $W_M^{(1)} = \frac{(\lambda x^{(2)} + \lambda_1 K(x^{(1)})^2)}{(2(1 - \rho - \lambda_1 x^{(1)}))}$. Thus, we have $\Delta \text{Perf}_{FCFS}(M, b = \lambda x^{(1)}) = \frac{\lambda x^{(2)} + \lambda_1 K(x^{(1)})^2}{2(1 - \rho - \lambda_1 x^{(1)})}$. Thus if $K$ is chosen to be large enough, $K(x^{(1)})^2 >> x^{(2)}$ and the Vulnerability Factor $V_{FCFS}(b = \lambda x^{(1)}) = \frac{\Delta \text{Perf}_{FCFS}(M, b = \lambda x^{(1)})}{\Delta \text{Perf}_{FCFS}(R, k)}$ can be made as large as one wants, if the system allows arbitrarily large jobs. Thus, if job sizes are not limited the simple FCFS mechanism is highly vulnerable. We therefore may conclude that queueing mechanisms can be quite sensitive to attacks, if not designed properly. Accounting for the Vulnerability Factor of these policies is important in designing them properly.

VII. COMBINING HASH WITH QUEUEING

In practical network systems, one would be interested in the combined effect of the hash table performance with that of queueing. This is the case where the requests to the hash get queued up in a queue and then be served by the hash table. To this end, the queueing analysis given above reveals that the mean waiting time of a request is affected not only by the mean processing time of the other requests but also by their second moment. Under the Post Attack Waiting Time metric one should revisit the analysis of the open hash using the Post Attack Operation Complexity metric (Theorem 3). Under that analysis the Vulnerability Factor of Open Hash was determined to be 1 since the mean length of the hash linked lists are the same regardless of in which list the items are added. Nonetheless, since in the Insert Strategy, the malicious users force all items into the same list they can increase, drastically, the second moment of the length of an arbitrary list. As a result, we show that they increase the the waiting time to pursue a regular user operation even in Open Hash.

Lemma 7: In Open Hash and Closed Hash systems, the Insert Strategy is an Optimal Malicious Users Strategy under the Post Attack Waiting Time metric.

Proof: Assume an M/G/1 queueing model where one processor handles insertion requests to an open hash table at rate $\lambda$. The Vulnerability is given as:

$$V_{OH}(k) = 1 + \frac{(1 - \lambda L)(k - 1)(1 - \frac{1}{M})}{(1 - \lambda L)(k - 1)(1 - \frac{1}{M}) + 2L + 1 - \lambda L(L + \frac{1}{M})} \quad (4)$$

Assume an Open Hash table of size $M$ with load factor of $L$. Let $W_{state}^{(1)}$ be the expected waiting time of a table operation according to the state of the hash table. Where state can be either of C, R or M, denoting respectively the cases of i) no insertions were made by additional users ("clean"), ii) $k$ elements were inserted by regular users, or iii) $k$ elements were inserted by malicious users. Let $Y_{state}$ be a random variable denoting the number of elements in an arbitrary Hash bucket. We will use $Y_C$, $Y_R$ and $Y_M$ to derive $W_C$, $W_R$ and $W_M$, respectively.

We assume that the service time for each operation equals the number of the elements in a bucket related to the operation, so according to the M/G/1 waiting time formula we get:

$$W_{state}^{(1)} = \frac{\lambda Y_{state}^{(2)}}{2(1 - \lambda Y_{state}^{(1)})}, \quad \text{which together with Eq. 4 yields:}$$

$$V_{OH}(k) = 1 + \frac{(1 - \lambda Y_C^{(1)})(Y_R^{(2)} - Y_C^{(2)})}{(1 - \lambda Y_R^{(1)})(Y_R^{(2)} - Y_C^{(2)}) + \lambda(Y_R^{(1)} - Y_C^{(1)}))Y_C^{(2)}} \quad (5)$$

For the individual models we get $Y_C^{(1)} = L$, $Y_R^{(2)} = L^2 + \frac{M-1}{M} L$, $Y_R^{(1)} - Y_C^{(1)} = \frac{k}{M}$, $Y_R^{(2)} - Y_C^{(2)} = \frac{k}{M}(\frac{k-1}{M} + 2L + 1)$, $Y_M^{(2)} - Y_R^{(2)} = \frac{k}{M}(k - 1)(1 - \frac{1}{M})$, which now lead to the proof.

Remark: Simplifying Eq. 3 we deduce that the Vulnerability Factor is proportional to $k$, namely attackers can be quite effective. The term $(1 - \lambda L)$ should be considered as a finite non-small number since in normal system $\lambda L$ is kept, for stability purposes below 0.8.

Next, consider calculate the vulnerability of Closed Hash. We define the random variables $Q_C$, $Q_R$, and $Q_M$ of the complexity of one insertion to a closed hash table in the same manner $Y_C$, $Y_R$, and $Y_M$ were defined for Open Hash.

The queueing model has not changed, and in the same way as in equation 5 we get that:

Under the Post Attack Waiting Time metric

$$V_{CH}(k) = 1 + \frac{(1 - \lambda Q_C^{(1)})(Q_M^{(2)} - Q_C^{(2)})}{(1 - \lambda Q_R^{(1)})(Q_R^{(2)} - Q_C^{(2)}) + \lambda(Q_R^{(1)} - Q_C^{(1)}))Q_C^{(2)}} \quad (6)$$

Assume a regular closed hash table with given table size $M$ and $R$ existing elements. Let the random variable $1 \leq Q_{reg} \leq r + 1$ be the complexity of one regular element insertion into the table. $P(Q_{reg} = s) = (\prod_{i=1}^{s-1} \frac{r-(i-1)}{M-(i-1)}) \cdot \frac{M-r}{M-r-(s-1)}$.

$$E[Q_{reg}^{(d)}] = \sum_{s=1}^{r+1} \left( \prod_{i=1}^{s-1} \frac{r-(i-1)}{M-(i-1)} \right) \cdot \frac{M-r}{M-r-(s-1)} s^d. \quad (7)$$

The fact that $E[Q_C^{(d)}] = E[Q_{reg}\overline{N}^{(d)}]$ and $E[Q_R^{(d)}] = E[Q_{reg}(N+k)^{d}]$ is used with equation 7 to calculate the moments of $Q_C$ and $Q_R$. The moments of $Q_M$ are calculated according to equation 3, and now we can use equation 6 to get the vulnerability value.
VIII. Open and Closed Hash: Vulnerability Comparison

We use the results derived above to compare the vulnerability of the Open and Closed Hash models. To conduct the comparison we will adopt the common approach that an Open Hash table with \( M \) buckets is performance-wise equivalent to a Closed Hash table with \( 2M \) buckets\(^{17}\). Unless otherwise stated, we use the parameters \( M = 500 \) and \( k = N \), meaning that the additional user always doubles the number of the existing values \((k = N)\) in the Hash.

Figures 2(a) and 2(b) and 2(c) depict the vulnerability of the Open and Closed Hash as a function of the number of existing elements \((N)\) under the different metrics. The figures demonstrate a striking and very significant difference between the Open and the Closed Hash: The Closed Hash is much more vulnerable to attacks than the Open Hash.

The performance of the open and closed Hash for the regular users waiting time metric is provided in Figure 2(c). The figure reveals two interesting and important properties of the system. First, the Open Hash is subject to very significant vulnerability in comparison to the Post Attack Operation Complexity metric. This is due to the fact that the delay is proportional to the second moment of chain length (and not to the first moment as in Figure (b)). Second, the Closed Hash is drastically more vulnerable (the y-axis scale is logarithmic, reaching values of 10,000, the hash is no longer stable for \( N \geq 237 \)) resulting from the compounded effects of the tendency of requests to address large clusters and of the mean queueing delay to depend on the second moment of the hash operation times.

We demonstrate the influence of the different parameters on the Vulnerability Using additional Figures. While increasing the number of buckets can reduce the Vulnerability in the Post Attack Operation Complexity metric (in the case of the Closed Hash), and in the Post Attack Waiting Time metric, the parameter has only negligible impact in the case of the In Attack Resource Consumption metric since there, the dominant factor is the magnitude of the attack, \( k \). In Figure 4 we study in more detail the vulnerability of the Hash using the In Attack Resource Consumption metric, as a function of the attack size \((k)\) and the number of existing elements in the table \((N)\), which influences directly the load before the attack. The figure demonstrates that when the table is lightly populated (upper figures) the Open and Closed Hash are quite similar to each other in vulnerability; the similarity stems from the fact that in the Closed Hash, with high likelihood a simple chain of the first \( k \) elements of the attack, will be formed.

\(^{17}\)It is common to consider them equivalent with respect to the space they require due to the pointers required by the Open Hash.
(and thus will be similar in performance to the Open Hash). The more common/realistic cases are depicted in the lower figures, representing medium to high Hash population (load). In these cases the vulnerability of the Closed-Hash becomes much larger due to the clustering effect.\footnote{One might question why at high load \((N = 700, k = 250)\) the Closed-Hash vulnerability metric starts decreasing and approaches that of the Open-Hash. The reason is that at this load, the Closed Hash is almost full, and thus is already very sensitive to the regular users, causing the ratio in Eq. 1 to decrease.}

Another implication is that in hostile environments one can no longer follow the simple approach of relying only on complexity based rules of operations in order to comply with the performance requirements of a system. For example, based on traditional performance one would double the size of the Closed Hash table (rehashing) when the table reaches 70–80\% load. However, in a hostile environment the rehashing must be done much earlier. For specific Hash state we can calculate the maximum magnitude of an attack so that the Hash remains stable after the attack has ended. Above this stability point, the status of the Hash is such that the arrival rate of regular user operations times their expected processing time is greater than one. Therefore the system cannot cope with the users requests, and hence the users suffer from a total denial of service and not just partial performance degradation. Using the state of the Hash in Figure 2\((c)\) \((M = 1000\) and \(k = N)\) the stability point is \(N = 237\). I.e., with less than 48\% of the buckets occupied the Hash is no longer stable. The attackers of the closed hash can destabilize the queueing system even if the expected cluster size is not too large, since the waiting delay is proportional to the second moment of the cluster size.

**IX. Concluding Remarks and Future Work**

The performance of today’s networks and computers is highly affected by DDoS attacks launched by malicious users. This work originated in the recognition that in order to properly evaluate the performance and qualities of these system one must bridge between the world of security and that of quality of service and add to the traditional performance measures, a metric that accounts for the resilience of the system to malicious users. We proposed such a Vulnerability Factor that measures the relative effect of a malicious user on the system. Using the Vulnerability measure we reveal some interesting understanding of the Hash system vulnerability: 1) the Closed Hash is much more vulnerable than the Open Hash to DDoS 2) after the attack has ended, regular users still suffer from performance degradation. 3) this performance degradation may reach the level of a total denial of service. We can calculate the exact magnitude of attack that can cause it. 4) application using Hash in the Internet, where there is a queue before the hash, has high vulnerability. In this case the waiting time the regular users suffer is proportional to the second moment, and hence the attack size has a duplicate effect on the Vulnerability.

This study has only just began and we believe that much more work is needed in order to evaluate many traditional systems and examine their vulnerability.
X. APPENDIX

Lemma 9: In Open Hash systems, the Insert Strategy is an Optimal Malicious Users Strategy under the Consumption of System Resources metric.

Proof: Let \( st = o_1, o_2, o_3, \ldots, o_l \) be the Optimal Malicious Users Strategy, a series of length \( l \) of Open Hash operations (i.e., insert, search or delete of a specific element value). Let \( n_i \) be the complexity (number of memory references) of the operation \( o_i \). Clearly, the operation that requires the most memory references is the insert operation of a new element (see [20]).

As a first stage we prove that \( st \), is a combination only of insert operations of new elements. Assume by contradiction, that there is an operation, \( o_r \), that the operation is not an insert operation of a new element. Let \( o_r \) be the first such operation in the series. Let \( B \) be the bucket on which the element of the operation \( o_r \) is hashed to.

Let \( st' = o_1', o_2', \ldots, o_l' \) be defined as follows: For \( i = r \), \( o_i' \) is an insert operation of a new element to bucket \( B \). All the other operations, remain the same i.e., \( o_i' = o_i \) for \( 1 \leq i \leq l \) where \( i \neq r \). Respectively, let \( n_i' \) be the complexity of the \( o_i' \) operation. Clearly for \( n_i' = n_i, 1 \leq i \leq r \) and \( n_r < n_r' \). Due to the operation \( o_r' \), the number of new elements in bucket \( B \) has increased, and hence \( n_i \leq n_i' \) for \( r < i \leq l \), (since the operation \( o_r' \) can only increase the memory references for other insert operations that are executed on bucket \( B \) after \( o_r' \)).

Contradiction: the new strategy \( st' \) has a higher consumption of system resources than the original \( st \).

Next we show that in the optimal strategy all the insert operations of new elements must be hashed to the same bucket (i.e. the optimal strategy is the Insert Strategy). For ease of presentation, in our analysis we ignore the elements in the hash that are inserted by regular users. Namely, we assume the hash was empty before the series of the operations \( st \).

The assumption does not change the final result, since the attacker has no knowledge about the elements regular users have inserted, and hence from his perspective, all the buckets are equal, and have the same load on average.

Again we will prove by contradiction: assume that the optimal strategy \( st = o_1, \ldots, o_l \) is a combination of insert operations of new elements to more than one bucket. Let us look at the states of the buckets after completing the execution of the series of operations \( st \). Let \( G \) be the bucket with the longest linked list. Let \( o_r \) be the first operation at \( st \) where the operation is to bucket \( S \) where \( S \neq G \). We create a new strategy \( st' = o_1', \ldots, o_l' \) where we change the \( r \)-th operation: \( o_r' \) is the insert operation of a new element to bucket \( G \) and all other operations remain the same i.e., \( o_i' = o_i \) for \( 1 \leq i \leq l \) where \( i \neq r \). Respectively, let \( n_i' \) be the complexity of the \( o_i' \) operation. Next, we show that the new strategy \( st' \) requires more memory references. Hence we have a contradiction to the optimality of the malicious strategy \( st \), and the lemma follows.

We show that \( \sum_{i=1}^{l} n_i' - \sum_{i=1}^{l} n_i = |G| - |S| + 1 \geq 1 \), where \( |G|, |S| \) is the number of elements in bucket \( G \) and \( S \), respectively, after the original strategy \( st \) was preformed.

From the definition of \( G \), \( |G| \geq |S| \). The change of the \( r \)-th operation influences the required memory references only on the \( r \) operation and the following operations after \( r \). Hence, for \( 1 \leq i \leq l, n_i' = n_i \). Assume that \( o_i \) inserts the \( i \)-th element to bucket \( S \), while \( o_i' \) inserts the \( g \)th element to bucket \( G \). Hence \( n_i' = n_i + g + 1 - s \). Moreover, the complexity for each of the next \( |G| - g \) operations at \( st \) that insert new element to bucket \( G \) is increased by one in the new strategy \( st' \) (i.e. for such \( o_i \) operation \( n_i' - n_i = 1 \)), while the complexity for each of the next \( |S| - s \) operations that insert new element to bucket \( S \) is decreased by one in the new strategy \( st' \) (i.e. for such \( o_i \) operation \( n_i' - n_i = -1 \)). Therefore, we receive that the change of consumption of system resources between strategy \( st' \) to \( st \) is equaled to \( |G| - g - (|S| - s) + g + 1 = |G| - |S| + 1 \). \( st \) is not optimal, contradiction.