On the Vulnerability of Hardware Hash Tables to Sophisticated Attacks

Udi Ben-Porat[1], Anat Bremler-Barr[2], Hanoch Levy[3], and Bernhard Plattner[1]

[1] Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland
{ehudb, plattner}@tik.ee.ethz.ch
[2] Computer Science Dept., Interdisciplinary Center, Herzliya, Israel
bremler@idc.ac.il
[3] Computer Science Dept., Tel-Aviv University, Tel-Aviv, Israel
hanoch@cs.tau.ac.il

Abstract. Peacock and Cuckoo hashing schemes are currently the most studied implementations for hardware network systems such as NIDS, Firewalls, etc.. In this work we evaluate their vulnerability to sophisticated complexity Denial of Service (DoS) attacks. We show that an attacker can use insertion of carefully selected keys to hit the Peacock and Cuckoo hashing schemes at their weakest points. For the Peacock Hashing, we show that after the attacker fills up only a fraction (typically $5\% - 10\%$) of the buckets, the table completely loses its ability to handle collisions, causing the discard rate of new keys to increase dramatically - between 10^2 and 10^4 times higher. For the Cuckoo Hashing, we show an attack that can impose on the system an excessive number of memory accesses and degrade its performance. We analyze the vulnerability of the system as a function of the critical parameters and provide simulations results as well.

Keywords: DDoS, Network Hardware, Hash , Peacock, Cuckoo.

1 Introduction

Modern high speed networks pose a challenge for routers, Firewalls, NIDS (Network Intrusion Detection System) or any other network devices that have to route, measure or monitor a network without slowing it down. Such network hardware elements are highly preferable targets for DDoS (Distributed Denial of Service) attacks since their failure can severely slow the network and, in the case of security systems, their failure can allow an attacker to conduct an attack on a critical system they meant to protect. Equipped with knowledge about how the system works, an attacker can perform a low-bandwidth sophisticated DDoS attack, targeting weak points in the system, rather than just flooding it (which takes more efforts and can be detected and countered more easily).

For example, Crosby and Wallach [2] demonstrated attacks on Open Hash table implementations in the Squid web proxy and in the Bro intrusion detection system. They showed that an attacker can design an attack that achieves worst

case complexity of $O(n)$ elementary operations per insert operation (instead of the average case complexity of $O(1)$), causing, for example, the Bro server to drop 71% of the traffic (without increasing the volume of the traffic). Smith et al. [1] describe a low bandwidth sophisticated attack on a NIDS system, in which the attack disables the NIDS of the network by exploiting the behavior of the rule matching mechanism and sending packets which require very long inspection times. In another example

Hash tables play an important role in the operations of the most important and time consuming tasks these systems have to perform. Using hashing techniques which allow constant operation complexity is therefore highly desirable. Multiple-choice Hash Tables (MHT), and in particular Peacock [3] and Cuckoo [4] Hashing, are easy to implement and are currently the most efficient and studied implementations for hardware network systems such as routers for IP lookup (for example [5], [6] and [7]), network monitoring and measurement (for example, [16]) and Network Intrusion Detection/Prevention Systems (NIDS/NIPS) [8]. For more information about hardware-tailored hash tables we recommend the recent survey by Kirsch et al. [9].

A Peacock hash table consists of a large main table (which typically holds 90% of the buckets) and a series of additional small sub-tables where collisions caused during insertions are resolved. Its structure is based on the observation that only a small fraction of the keys inserted into a hash table collide with existing keys (that is, hashed into an occupied bucket) and even a smaller fraction will collide again, etc. These backup tables are usually small enough for their summary (implemented by bloom filters) to be saved on fast on-chip memory which dramatically increases the overall operation performance in Peacock Hashing.

A Cuckoo Hashing is made of two or more sub-tables of the same size. Every key can be placed in one bucket into which it hashes in each sub-table. When a key k finds all its buckets are already occupied with keys, one of those keys is moved to one of its alternate locations to free the bucket for k. Cuckoo Hashing therefore allows higher utilization than achieved by other MHT schemes that do not allow moves while maintaining $O(1)$ amortized complexity of an Insert operation (although, unlike Peacock Hashing, the complexity of a single Insertion is not bounded by a constant).

In this work we expose the weak points of the Peacock and Cuckoo Hashing and the system parameters that affect them. To evaluate the vulnerability of Peacock and Cuckoo we refer to [10] which observed that an attack on a hash table data structure can damage the performance of the system in two ways: 1. *In-attack damage* - Insertions of keys that require excessive number of memory accesses; 2. *Post-attack damage* - Insertion of keys that are placed in the table in a way that causes future insertions of keys to take excessive memory accesses and/or reduce their probability to find an empty bucket. Using this classification, we propose a solution that makes Peacock hashing both resilient to in-attack damage and more efficient. We explain how such an attack can be countered easily; On the other hand we show that it is vulnerable to post-attack damage.

We propose a sophisticated attack that can dramatically increase the discard probability of a newly inserted key after the attack has ended. We show that after the attacker inserts keys into the table, it brings the table into an irreversible state in which the probability for a newly inserted key to be discarded can be 100 to $10,000$ times higher than after the same amount of keys are inserted by regular users. For Cuckoo hashing we explain why post-attack damage is irrelevant and analyze its in-attack vulnerability. We show that an attacker can slow the system by inserting keys that require 4 times more memory accesses than regular keys in a typical settings. The number of sub-tables and the moves limit (the maximal number of moves allowed during an insertion) define the utilization of a Peacock table. We show that increasing the number of sub-tables, not only increases the utilization of the table, but also decreases its vulnerability. In addition, we show that while increasing the moves limit increases the utilization of the table, it also increases its vulnerability.

In addition to mathematical analysis, we provide vulnerability simulation results of the both hashing schemes when following a use case in which a system designer designs a Cuckoo and Peacock hash tables which comply with the same requirements. In addition, we discuss the feasibility of the attack by evaluating the complexity of finding keys suitable for a sophisticated attack and show for both Peacock and Cuckoo Hashing that high number of sub-tables makes it harder for the attacker to find suitable keys.

The structure of the rest of the work is as follows: In Section 2 we explain the nature of sophisticated attacks against hash tables and the Vulnerability metric used in this work. Then, the main body of the work consists of sections 3 and 4 which are dedicated to the Peacock and Cuckoo Hashing respectively. Both sections have a similar structure. Each is divided into five parts covering the following topics: 1. The hashing algorithm; 2. Attack strategy; 3. Feasibility of the attack; 4. Vulnerability analysis and simulation results and 5. The resilience to in-attack (Peacock) or post-attack (Cuckoo) damage. Finally, Section 5 concludes the key results. In addition, a glossary of the key notations used throughout the work can be found in the Appendix.

2 Sophisticated Attacks on Multiple Hash Tables

In multiple hash table schemes, such as Peacock and Cuckoo Hashing, every key can be placed only in a small fraction of the buckets. While it allows performing Search and Delete operation with no more than a predefined constant number of memory references, it also poses a challenge: As the load in the table grows, both the *insertion complexity* (measured by the number of probed buckets) and the *discard probability* - the probability for an inserted key to not find a bucket to be stored in - increases. In order to keep these variables bounded by acceptable values, the utilization (maximal load) in the table is limited. Using knowledge about the table, an attacker can perform a *sophisticated* attack that degrades the system performance beyond its acceptable values (with which it was designed to comply) using just a small amount of keys and hence avoid reaching the maximal

load in the system. This reduces the amount of keys the attacker is requires to insert and helps the attack to avoid detection.

The vulnerability metric we use in this work has been proposed in [10] and is defined as the maximal performance degradation (damage) that malicious users can inflict on the system using a specific amount of resources (budget) normalized by the performance degradation attributed to regular users using the same amount of resources. Formally, according to [10], the *effectiveness* of an attack is defined by

$$E_{st}(budget = K) = \frac{\Delta Perf(M_{st}, K)}{\Delta Perf(R, K)}, \tag{1}$$

where $\Delta Perf(M_{st}, K)$ and $\Delta Perf(R, K)$ are the performance degradations caused by inserting additional K keys to the table (in the context of hash tables) by an attacker and regular users respectively and st is the attack strategy used by the attacker. Then, the vulnerability V of a system, is defined by the effectiveness of the strategy that causes the maximal damage:

$$V(budget = K) = max_{st}\{E_{st}(K)\}. \tag{2}$$

Therefore, when an attack strategy is not proved to be the optimal, its effectiveness is considered as a lower bound for the vulnerability of the system.

Note that in order to perform the sophisticated attacks analyzed in this work, the attacker is assumed to gain knowledge of the structure of the table (number of tables and their sizes but not how many keys are already stored in the table and where). In addition, the attacker is assumed to be able to compute or guess the hash values of keys. This knowledge can be achieved by reverse engineering of similar products acquired by the attacker, various guessing methods or exploiting the use of open source algorithms. For more information see [10].

3 Peacock Hashing

3.1 Insertion Algorithm

In Peacock Hashing [3] the buckets are divided into d sub-tables $\{T_i\}_{i=1}^{d}$ and d corresponding hash functions $\{h_i\}_{i=1}^{d}$. The size of the sub-tables follows a decreasing geometric sequence $M_{i+1} = M_i/r$ where r is the proportion between the tables[4]. The first sub-table T_1 is called the *main table*, while the rest are called the *backup tables*. T_1 is the largest table and it is where the insertion algorithm first tries to store a key. Every backup table handles the collisions in the sub-table that precedes it. The insertion algorithm probes the sub-tables $\{T_i\}_{i=1}^{d}$ one after the other until finding a table where the bucket into which the key hashes is free. In the rare case where a key cannot find its place in any of the tables - it is dropped. In addition, a summary of the keys stored in every backup table is maintained (implemented by bloom filters stored on the on-chip

[4] In [3] the authors recommended to use $r = 10$.

memory). The summary is used to avoid checking all the sub-tables when making sure an inserted key does not already exist in the table.

Remark 1 (Bucket Capacity). Increasing the number of locations in which a key can be placed can be done by increasing the number of sub-tables d. It can also be done by using buckets that can hold more than one key. Studies have shown that for both Peacock Hashing [3] and Cuckoo Hashing [**?**] it increases the table utilization dramatically (while maintaining good performance of the different hash operations). Unlike the number of sub-tables d, the number of keys in a bucket is not a major factor in the vulnerability of the system and the feasibility of the attack. Hence, in order to decrease the number of parameters, our model follows the original Peacock and Cuckoo algorithms where every bucket can hold only one key. It can be shown that while it increases the number of *overhead keys* (keys the attacker inserts before causing damage) it decreases by a greater factor the complexity of finding the keys for the attack[5].

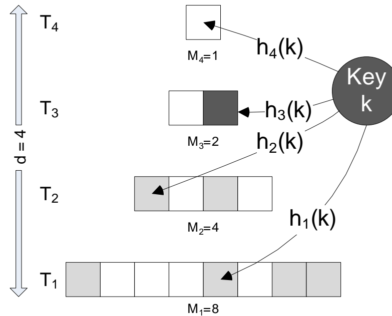


Fig. 1. An insertion of key k into a small Peacock hash table with $r = 2$ and $d = 4$. Gray buckets are occupied with existing keys and white buckets are free. The black bucket marks the bucket where k is finally placed.

After a long series of insertions and deletions (of keys) from the hash table, it becomes unbalanced. That is, the load in the smaller sub-tables is higher than in the larger tables and this increases the discard probability of a new insertion [3]. This state can be prevented by re-balancing the table after a key is deleted, as follows. After a key was deleted from bucket b in T_i, if there is a key k that is stored in $T_{j>i}$ such that $h_i(k) = b$, then k is moved back from T_j to the now-free bucket in T_i. The attack we propose and analyze below brings the hash table to an unbalanced state that cannot be resolved by re-balancing (unlike unbalance that occurs over time).

[5] Compared to a table that can hold the same total number of keys which is consisted of more buckets that have lower key capacity.

3.2 Post-Attack Damage: Attack on Peacock Hashing

As already explained, [3] showed that when the keys are concentrated in the backup tables (the table is unbalanced) the discard probability increases. The malicious user can artificially create an extreme case of this scenario by flooding the backup tables. A simple example of such an attack can be done by inserting K keys that all hash into the same bucket b at T_1. Every inserted key (except possibly for the first one) will collide with an existing key in b and then, according to the insertion algorithm, will be rehashed into a bucket in one of the backup tables. After the attack has ended the table is unbalanced, because the keys inserted by the malicious user are concentrated in the backup table while almost none are in the main table. This causes *Post-Attack* damage, measured by the increase in the discard probability.

Normally, a Peacock hash table maintains a low discard probability by limiting the maximal load in the table. The above sophisticated attack algorithm causes the discard probability to exceed the value in which the load-limit was meant to enforce.

by using only a small number of keys that do not increase the overall table load beyond its limit. Unlike a Peacock table which became unbalanced naturally, a re-balancing routine cannot bring items back into the main table after such an attack, because the bucket in which they all hash into in T_1 can contain only one of them but not all. So not only the table becomes unbalanced, there is also no way to re-balance it until the keys are flushed out from the table or the table is reconstructed with a new set of hash functions[6].

Denote the set of buckets in which a key is hashed into in all sub-tables as the *pool* of the key in the table. The insertion algorithm discards a new key only if all the buckets in its pool are already occupied. Every key is hashed into one bucket in each table, therefore, a bucket b in a table of size M_i, belongs to the pools of $1/M_i$ of the keys in the key space. In simple words, a key placed in a high table T_i "gets in the way" of more potential keys than if it was placed in a bucket in a lower table T_j $(j < i)$ (since $1/M_i > 1/M_j$). That is, if he could, a malicious user would always prefer to place a key in T_i rather than in T_{i-1} and increase the number of keys affected by factor of r $(r = M_{i-1}/M_i)$. Thus, if the malicious user can insert only a very limited number of keys, he would prefer to insert them as deep as he can.

We now give a formal description of the attack algorithm, generalizing the simple attack we described earlier. An attack in depth j is an attack in which the attacker inserts keys that not only hash into the same bucket in T_1, but also hash into the same buckets in $T_2, ..., T_j$. This will lead to the flooding of the most upper tables $T_{j+1}, ..., T_d$. That is, the simple attack we described above is an attack in depth 1.

Loss Probability of the Attack Keys:

During the flooding the upper backup tables they get more and more saturated. Therefore, the probability for a malicious key to be dropped increases

[6] Which if possible at all, will undoubtedly consume a huge amount of resources.

from one key to the next. At some point, the probability for the next malicious key to be dropped can be so high that it is may not be effective anymore to continue and insert additional keys in the same manner and it would create more damage if the next keys will be inserted into a lower table. During an attack in depth $j+1$, we examine the following two actions: A. The next key is inserted into Table T_{j+1} (as all keys before him); B. It is inserted into depth T_j. Assume that the tables $\{T_i\}_{i=j+1}^{d}$ are balanced and the load in each of them is β, while the load in $\{T_i\}_{i=1}^{j}$ is α. Let P_A and P_B be the the probabilities for a regular key to be dropped after action A and B respectively. In the following analysis we try and estimate at what point P_B becomes larger than P_B. Define P'_B be the value of P_B given that the inserted malicious key is hashed into a free bucket in T_{j-1}. Then, $P_B = (1-\alpha)P'_B + \alpha P_A$. Therefore, $P_B > P_A \iff P'_B > P_A$.

Lemma 1.

$$P'_A = \alpha^{j-1}(\alpha + 1/M_j)\beta^{d-j}. \tag{3}$$

Proof. The load in T_j after adding to it the malicious key is $\alpha + 1/M_j$. The probability for a key to be hashed into occupied buckets in all tables equals the product of their loads, hence we get Eq. 3.

Lemma 2.

$$P_B = \beta^{2(d-j)}\alpha^j + \beta^{d-2j-1}\sum_{i=j+1}^{d}\beta^i(1-\beta)\alpha^j(\beta + 1/M_i). \tag{4}$$

Proof. Following action B, the malicious key can be placed in a free bucket in $\{T_i\}_{i=j+1}^{d}$ or dropped. The probability it is dropped is β^{d-j}. In this case, the probability for a regular key to be dropped is $\alpha^j\beta^{d-j}$. the probability for a key to be placed in T_i is $\beta^{i-j}(1-\beta)$. In this case, the probability for a regular key to be dropped is $\alpha^j\beta^{d-j-1}(\beta + 1/M_i)$. Therefore, we get that $P_B = \beta^{d-j}\alpha^j\beta^{d-j} + \sum_{i=j+1}^{d}\beta^{i-j}(1-\beta)\alpha^j\beta^{d-j-1}(\beta + 1/M_i)$. Then we get $P_B = \beta^{2(d-j)}\alpha^j + \alpha^j\beta^{d-2j-1}(1-\beta)\sum_{i=j+1}^{d}(\beta^{i+1} + \beta^i/M_i)$.

First, observe that as M_j increases, the value of P_A decreases. This implies the influence of the size proportion between the sub tables (r), although it is not explicitly used in the above equations. For example, when $r=4$, since $M_i/r = M_{i+1}$ then $\sum_{i=j+1}^{d} M_i = M_j/4 + M_j/16 + ... \sim M_j/3$ and when $r=10$ then $\sum_{i=j+1}^{d} M_i \sim M_j/9$. As explained earlier, as β increases, so is the probability for the injected keys to be dropped and hence γ decreases (which decreases P_A). In addition, the smaller α is, the contribution to the drop rate of the k'/M_j gets more significant.

3.3 Feasibility of the Attack

As explained in Section 2, in order to perform the sophisticated attack the attacker is required to be able to predict the hash values of keys, but this alone

does not guarantee the feasibility of the attack. The feasibility of the attack depends on the ability of the attacker to find keys for the attack that hash into the same bucket(s) in the different tables. The effort of finding enough keys for the attack is measured by the number of keys that have to be probed before the malicious user finds a proper set of keys for the attack.

For an attack on a Peacock hash table with in depth j, the malicious user will have to find keys that all hash into the same place in $\{T_i\}_{i=1}^{j}$. A simple algorithm to find the keys is to randomly choose a key k_0, and then probe keys until finding K keys that all hash into the same buckets in $\{T_i\}_{i=1}^{j}$ as k_0. Note that the attacker can shorten his search by trying to match every key he probes not only with k_0, but also with other keys $k_2, .., k_i$, creating i pools of keys. Then when one of the pools contain enough keys for the attack, the malicious stops his search. The larger the number of pools is, the larger the memory required for the search. Regardless of the number of keys K the attacker searches or the manner he does that, the effort of finding them is a factor of the expected number of keys needed to be probed before finding the a key which hashes into a specific j buckets in the first j tables.

The probability that a probed key collides with the first key in all the first j tables is $\prod_{i=1}^{j} 1/M_i$. That is, the expected number of keys to be probed until the next key for the attack is found is $\prod_{i=1}^{j} M_i$. That is, the number of the expected keys to be probed increases as the attack gets deeper and also as the size of the first tables increases. Since $M_i = r^{d-i}$ (assuming $M_d = 1$) then $\prod_{i=1}^{j} M_i = r^{j(d-0.5(j+1))}$. Therefore, the larger r and d are, the harder will be for the attacker to find the keys for the attack. For example, lets look at a table with $r = 10$ and $d = 5$ (the total size of the table is $M = 11,111$). If the attacker plans an attack in depth $j = 1$ and flood the backup tables (consisted of $1,111$ buckets) with 1000 keys then he will need to probe $1000r^{j(d-0.5(j+1))} = 10^7$ keys for the attack, which is a feasible task. If the attacker wants to attack the top two tables (the attack depth is $j = 3$) that contain 11 buckets together, with 10 keys, he will need to probe about $10r^{j(d-0.5(j+1))} = 10^{10}$. Even if the attacker have the ability to probe so many keys in reasonable time, he might still not be able to perform it since the key space itself might not be that large. For example, if the keys are IPv4 IP addresses, then the key space is too small to find enough keys since $10^{10} > 2^{32}$. Therefore, the design of the table (setting the values of r and d keys) can limit the depth of the attack even for an attacker with unlimited computational capabilities.

The attacker can further decrease the complexity of finding the keys on the expense of the effectiveness of the attack by increasing the number of overhead keys required for the attack. Instead of choosing one target bucket in each of the first j table in which all the attack keys have to hash into, the attacker can choose a target set of buckets. Formally, the attacker choose j *target bucket sets* $\{S_i \subset T_i\}_{i=1}^{j}$ (each for every table) and adds to the attack key set every key k that such that $\wedge_{i=1}^{j} h_i(k) \in S_i$.

Theorem 1. *When using target sets of size* $\{S_i \subset T_i\}_{i=1}^{j}$*, the expected number of keys the attacker has to probe before finding a suitable key for the attack is*

$$C_{key}^{peacock} = \prod_{i=1}^{j}(M_i/|S_i|) \tag{5}$$

and the number of overhead keys is

$$K_{o/h}^{peacock} = (1-\alpha)\sum_{i=1}^{j}|S_i|. \tag{6}$$

Proof. The probability for a random probed key to hash into a bucket in T_i that belongs to S_i is $|S_i|/M_i$ and the probability to hash into all the sets in $\prod_{i=1}^{j}(M_i/|S_i|)$. Then the expected number of keys to be probed until finding a key that will be added to the attack keys set is $C_{key}^{peacock} = 1/\prod_{i=1}^{j}(M_i/|S_i|$. For the number of overhead keys, up to $\sum_{i=1}^{j}|S_i|$ of the attack keys can find a place in the first j tables before reaching the targeted backup tables. If the existing load prior the attack is α, than only $1-\alpha$ of the buckets in the targeted sets are expected to be free when the attack start, hence we get $K_{o/h}^{peacock} = (1-\alpha)\sum_{i=1}^{j}|S_i|$. □

For example, consider again a table where $r = 10$ and $d = 5$ where the attacker prepares an attack in depth $j = 3$. If the attacker use target sets in size $S_i = 1\%M_i$, then the number of expected keys to be probed before he finds the next key for the attack is reduced from $\prod_{i=1}^{3} M_i = 10^9$ to $\prod_{i=1}^{3}(M_i/S_i) = 10^6$. Nevertheless, this approach will cause more malicious keys to occupy buckets in the lower j tables rather than the targeted backup tables, therefore the malicious user will have to use more keys for the attack. During an attack in depth j, some of the keys are stored in the lower j tables and do not reach the upper targeted table. These keys cause less damage than the rest of the keys, therefore we refer to them as *overhead keys* as opposed to the rest of the keys which are referred to in this work as *effective keys*.

In the original attack algorithm (where $\{|S_i| = 1\}_{i=1}^{j}$) the expected number of overhead keys is $(1-\alpha)j$ (where α is the load in every table, assuming the tables are balanced before the attack). Using target sets, the number of overhead keys is $(1-\alpha)\sum_{i=1}^{j}|S_i|$. In the previous example, using target sets such that $|S_i| = 1\%M_i$, the amount of overhead keys is multiplied with $(\sum_{i=1}^{j}|S_i|)/j = 370$. That is, using target sets the attacker will have to search for 370 times more overhead keys to "pad the way" for the rest of the keys to reach the upper tables. Since using target sets allows the malicious find keys $10^9/10^6 = 1000$ times faster, it will still improve the feasibility of finding the keys for the attack (even that now it will have to find more keys).

To conclude the feasibility results, as expected, they show that since every key has to hash to the target bucket(s) in each of the first j tables, the results show that the deeper the attack is (larger j), the harder it is for the attacker

to find keys for the attack (larger $C_{key}^{peacock}$). The larger the tables-proportion is (large r), the bigger is the percentage of buckets that the first j tables hold. For example, when $d \leq r$, it can be shown that the main table holds at least $r/(r+1)$ of the buckets in the table. For $d = 5$, the main table holds (M_1) 95%, 90% and 80% of the total buckets in the table (M) for $r = 20$, $r = 10$ and $r = 5$ respectively. Therefore, the key search complexity $C_{key}^{peacock}$ increases with r. Note that on the other hand, when r is large, the amount of buckets in the targeted backup tables is lower, and the attacker will need a smaller attack key set. Therefore, while a large r increases the complexity to find one attack key, we cannot conclude whether it increases the total complexity of finding the attack key set based on r alone. The results also show that the attacker can, on the expense of the effectiveness of the attack, ease the complexity of finding the attack-keys by decreasing the depth of the attack and/or by decreasing the accuracy of the attack-keys by using target-sets.

3.4 Post-Attack Damage: Vulnerability of Peacock Hashing

The Vulnerability factor (described in Section 2) measures the proportion between the performance degradation caused by a malicious user and a regular one. Therefore, the first step in evaluating the vulnerability is deciding what is the performance metric in which its vulnerability is measured. The sophisticated attack described in this work aims to increase the discard probability of the keys inserted after the attack has ended. Hence, in this section we focus on analyzing the discard probability. Note that one can also measure the vulnerability of other performance metrics influenced by the attack such as the average waiting time or queue length for the pending hash operations and so on. When considering a specific system, a system designer can measure the vulnerability of the end performance of the designed system, such as the forwarding speed in routers or the percentage of packets (or flows) that go unnoticed when designing an intrusion detection system.

The probability for an inserted key to collide with existing keys in all d tables and hence be dropped is $Q = \prod_{i=1}^{d} alpha_i$ where α_i is the load in T_i. If the key was stored in T_l, the load in T_l increases to $\alpha_l + 1/M_l$ and the probability that the next key to be dropped is $Q' = Q \cdot (1 + 1/(\alpha_l M_l))$. Therefore, the smaller T_l is (small M_l), the increase in the discard probability is higher. For example, assume a Peacock hash table with proportion $r = 10$ ($M_i = r^{d-i}$)and $d = 5$ tables where all the tables have the same load $\alpha = 0.1$. If a regular user inserted a key that found a place in T_2, then the discard probability after the insertion of the key is $Q'/Q = (1 + 1/(0.1 * 10^3) = 1.01$. If it was a malicious user, managing to insert a key to T_4, the discard probability would increases dramatically by $Q'/Q = 2$. This is the reason how a malicious user inserting keys targeted at the higher tables increases the discard probability.

The following analysis takes into account the different probability for a key to be dropped during insertions by an attacker and a regular user.

In the following lemmas we measure the vulnerability of the probability for a new inserted key to be discarded. Let DP_I, DP_R and DP_A ('I' - Initial, 'R' -

Regular, 'A' - Attacker) be the expected discard probabilities of a new inserted keys at the following states: 1. DP_I - when the load in the table is α, before any additional key is inserted; 2. DP_R - after K regular (random) additional keys were inserted; 3. DP_A - after K additional keys were inserted by a sophisticated attacker.

Even when assuming that prior the attack the table was balanced, the exact evaluation of these values is very complex. This is mainly because the fate of an inserted key (during an attack and in general) depends on how many of the previously inserted keys were dropped and in which sub-tables the rest of them were stored. We avoid making an exact evaluation since not only very hard to compute for large numbers, it also hides the main factors behind the vulnerability. Therefore, in order to simplify the analysis, we assume that after the insertion of every key, the loads in all sub table are equal, that is, the table is always balanced. Note that it is true in the long run but not after every insertion as we assume here. In mathematical terms, if X and Y are the loss probabilities of the i-th and the $i+1$-th insertions, then we use the equality $E[XY] = E[X]E[Y]$ although X and Y are not completely independent, especially in a small table (which is not common). Hence, we refer to the results here as estimations and not as exact evaluations. Later we compare our estimations with simulation results and show that... [**COMPLETE after the graphs for the analysis are ready**].

Lemma 3. *After inserting K regular random keys (into a balanced peacock table with load α) the probability for a new inserted key to be dropped is estimated by*

$$DP_R(K) = (\alpha + \sum_{s=1}^{K} p_s/M)^d, \tag{7}$$

where $p_1 = 1 - \alpha^d$ and $p_i = 1 - (\alpha + \sum_{s=1}^{i-1} p_s/M)^d$.

Proof. Let p_i be the probability that the first inserted key is not dropped, then since the tables are balanced, $p_1 = 1-\alpha^d$. In addition, after the insertion of $i-1$ keys, the expected number of keys which were not dropped can be estimated by $\sum_{s=1}^{i-1} p_s$, therefore the new load in the peacock table is $\alpha+\sum_{s=1}^{i-1} p_s/M$ (M is the total number of buckets in the table). Therefore, since the tables are balanced, $p_i = 1-(\alpha+\sum_{s=1}^{i-1} p_s/M)^d$. In a similar way, the estimated probability of a new inserted key to be dropped after inserting K random keys is as appears in Eq. 7. □

Lemma 4. *After an attacker performs an attack in depth j using K keys (in a balanced peacock table with load α) the probability for a new inserted key to be dropped is estimated by*

$$DP_A(K) = [\prod_{l=1}^{j}(\alpha + (1-\alpha)/M_l)](\alpha + \sum_{s=1}^{K'} p'_s/M')^{d'}, \tag{8}$$

where $K' = K - \lfloor(1-\alpha)j\rfloor$, $M' = \sum_{l=j+1}^{d} M_l$, $d' = d - j$, $p'_1 = 1 - \alpha^{d'}$ *and* $p'_i = 1 - (\alpha + (\sum_{s=1}^{i-1} p'_s)/M')^{d'}$.

Proof. Recall that all the inserted keys are hashed into the same j buckets in the first j hash tables. Since the initial load in the tables is α, then only $(1-\alpha)j$ of them are expected to be free before the attack begins. Therefore, it is expected that the first $(1-\alpha)j$ keys will be stored in the lower j tables, while the rest of the keys are expected to find all the first j buckets full and be inserted (randomly) into the upper $d-j$ tables. Now we want to know how many of the remaining $K' = K - \lfloor(1-\alpha)j\rfloor$ keys are expected to not be dropped when they reach the upper $d' = d - j$ tables where the load is α. This is similar to problem that was already solved in the proof for Lemma 3.4 - how many keys are expected to not be dropped when inserting K' random keys into a table of size d'? In a similar way, the result is $\sum_{s=1}^{K'} p'_s$ where $p'_1 = 1 - \alpha^{d'}$, $p'_i = 1 - (\alpha + \sum_{s=1}^{i-1} p'_s/M')^{d'}$ and $M' = \sum_{l=j+1}^{d} M_l$. Note that the load in the different first j tables is not expected to grow in the same way due to the nature of the attack. While the expected number of keys $(1-\alpha)$ that is be added to each of them is equal, the size of the tables is not. Therefore, the load of each table T_l $(j \geq l)$ is estimated by $\alpha + (1-\alpha)/M_l$. Since the rest of the keys are inserted randomly into the rest of the table, the ones which are not dropped are assume to increase the load in the upper tables in a balanced manner. That is, the load in each table T_l $(d \geq l > j)$ is estimated by $\alpha + \sum_{s=1}^{K'} p'_s/M'$. Therefore, the estimated probability for a random key to be dropped after the attack is as appears in Eq. 8. □

Theorem 2. *The vulnerability of the discard probability is estimated by*

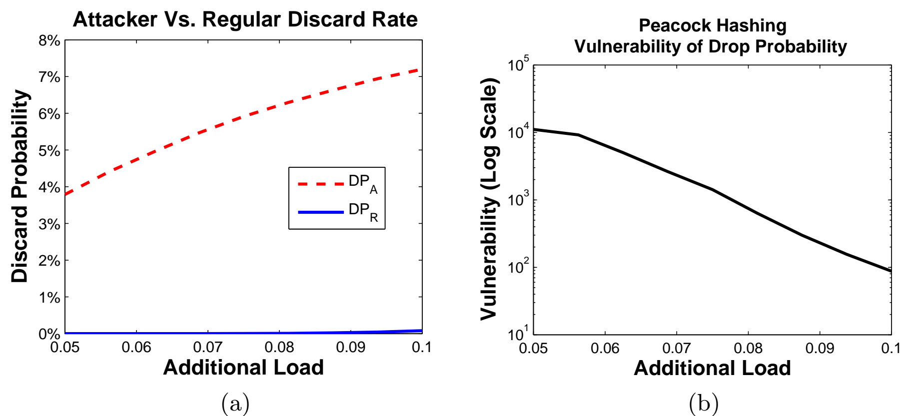
$$V_{DP}(K) = \frac{DP_A(K) - DP_I}{DP_R(K) - DP_I}, \quad (9)$$

where $DP_R(K)$ *and* $DP_A(K)$ *are given in Eq. 7 and 8 respectively and* $DP_I = \alpha^d$.

Proof. According to its definition, the vulnerability metric is the proportion between the degradation caused by an attacker to the degradation caused by regular users. The increase in the discard probability is $DP_A(K) - DP_I$ and $DP_R(K) - DP_I$ for the attacker and a regular user respectively. The probability for an inserted key to be dropped when inserted into a balanced table with load α is the probability that all the buckets it hashes into in all d tables are full, hence $DP_I = \alpha^d$. The values of $DP_R(K)$ and $DP_A(K)$ are proved in Lemmas and . □

The simulations we conducted follow a scenario in which the system designer examines the option of using Peacock Hashing for hardware that can allocate memory equal to approximately 10^5 buckets for the table. Building a table consisted of $d = 5$ sub-tables with sub-tables proportion of $r = 10$ results in a table of size $M = 11,111$ buckets. As mentioned before, in Peacock Hashing (as well in Cuckoo Hashing and other modern hashing schemes) the probability for a key to

be dropped during an insertion increases with the load in the table. Therefore, the maximal load in the table is decided by the *Acceptable Loss Fraction*, that is, the maximal percentage of inserted keys that the system can afford to lose. It is important to note that *Discard Probability* (Figure 2(a)) that is used to measure the vulnerability and *Loss Fraction* that is used to set the maximal load are two different metrics. *Discard Probability* measures the probability to drop an inserted regular key *after* additional keys were inserted by malicious or regular users while *Loss Fraction* measures the percentage of the inserted (regular) keys that are dropped during an insertion and is used to set the maximal load of the table. In this example, we assume that the desired maximal *Loss Fraction* allowed is 1%. Our simulations showed that such table a Peacock table with $d = 5$, $r = 10$ and $M = 11,111$ is suitable for the insertion of up to $0.3M = 3,333$ keys (the table utilization is 30%) before exceeding loss fraction of %1.



(a) (b)

Fig. 2. Figure (a): The discard probability after insertions by regular (DP_R) and malicious (DP_A) users as a function of the load they add (K/M). Figure (b): The vulnerability (V) of a table with proportion $r = 10$ and $d = 4$ with an existing load of 10% as a function of the additional load.

In the simulations, the attack was conducted on a table with an existing load $\alpha = 0.1$ and the attack by the malicious user is in depth $j = 1$. We can see that when the additional load is 5%, the discard rate is not changed under regular attack and practically remains zero, while the sophisticated attack causes the discard rate to increase to 4%, a discard probability that can be achieved by a regular attack with load which is 3.5 times larger (17.5%).

Recall that an attack in depth $j = 1$ on target the upper 4 tables that hold together only $1,111/11,111 = 9.9\%$ of the buckets that some of them are already occupied prior the attack. Therefore, an attacker would avoid inserting more than 10% of additional load (where the x-axis ends). Note that since the maximal load in the table is 0.3, the system is expected to allow an additional

up to 0.2, hence the attack does not cause the load in the table to exceed its limit.

We can see a significant difference in the discard probability after a malicious attack and after a regular attack. This difference is expressed by the extremely high vulnerability values, depicted in Figure 2(b), where the results show that the discard probability after the attack has ended caused by the attacker is between 100 and $10,000$ times larger than the discard probability after the attack has ended. This drives the performance of the hash table far beyond the performance in which it is assumed to be operating. This result emphasize the fundamental vulnerability of Peacock Hashing and its variants which dedicate specific range of buckets (the upper sub-tables) for resolving collisions.

3.5 Resilience to In-Attack Damage (and Improving Performance Using Bitmaps)

As already mentioned in Section 1, in this work we focus on analyzing an attack which causes *post-attack* damage. We now show how Peacock hashing can be made both more efficient and resilient to attacks aiming at *in-attack* damage by inserting keys that require excessive number of memory accesses during the attack. Our suggestion is to keep a bitmap summary of the occupied buckets for every *backup* table. Since the total size of the backup tables is small (about 10% of M when $r = 10$), these bitmaps are compact enough to be stored in the fast on-chip memory. Hence, the complexity of accessing a key is negligible. Then, when handling an insertion of a new key which cannot find its place in the main table, its final bucket (in a backup table) can be found directly by checking its hash values against the fast bitmap summaries. This way, no insertion has to probe more than two buckets (one in the main table, and one in the final destination). Note that it will not cause a mistaken insertion of a key that already exists, since (as mentioned earlier) in additional to the bitmap summary, there are also on-chip *key* summaries (commonly implemented in bloom filters [3]) which are used to make sure the key does not already exist in the sub-tables (before checking the bitmap summary to locate a free bucket for the key).

4 Cuckoo Hashing

4.1 Algorithm Description

According to the original definition by Pagh and Rodler [4] a Cuckoo hash table is made of two sub-tables, equal in size. Every key k hashes into one bucket in each of them using two hash functions $h_1(k)$, $h_2(k)$. If during an insertion, both of the possible buckets are occupied, the key is placed in one of them instead of an exiting element which is then moved into its alternative location in the same manner. Therefore, every insertion of a new key is consisted of a series of one or more *moves*. The complexity of a new insertion is measured by the number of moves it triggers. The expected complexity of an insertion is proved

to be bounded by $O(1)$ as long the maximal capacity of the table is not reached. Figure 3 depicts an example of an insertion of k_2 into a Cuckoo hash table with 4 sub-tables. When k_2 is inserted, the buckets in which it hashes into (dark left arrows) are all occupied. Then, the insertion algorithm chooses to eject k_1 and place k_2 in its place. k_1 is then relocated to an alternative free bucket in which it hashes into in T_3. Note that as already explained in Section 3.1, we discuss the model in which a bucket can hold only one key.

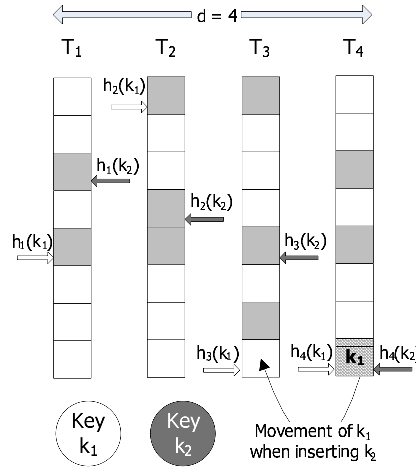


Fig. 3. A move of key k_1 during the insertion of k_2 into a Cuckoo hash table with 4 sub-tables. The white (right) and the dark (left) arrows mark the hash values of k_1 and k_2 in the different sub-tables respectively.

For the original Cuckoo hash table consisted of two sub-tables, Pagh and Rodler [4] showed that the complexity of an insertion is $O(1 + 1/\epsilon)$ where $M = 2N(1 + \epsilon)$, M is the total number of buckets (in both tables) and N is the maximal number of keys the table is meant to hold.

In [13] Fotakis, Pagh et al. generalized Cuckoo Hashing to *d-ary Cuckoo Hashing* where $d \geq 2$ sub-tables are used. They showed how N keys can be stored in $M = (1 - \epsilon)N$ buckets for any constant $\epsilon > 0$. They showed that Search and Delete operations take[7] $O(ln(1/\epsilon))$ and proved the generalized Cuckoo Hashing has a constant amortized insertion time. Following [13], Frieze and Mitzenmacher [14] suggested a more efficient insertion method with a polylogarithmic upper bound. In latest studies (such as [14], [13] and [15]) different insertion algorithms have been proposed. They mainly differ in the way they choose the key that will be relocated (in order to free a bucket for an inserted key that all his possible d locations are occupied). Note that the attack efficiency is independent of the way the key (to be moved) is chosen, hence we do not bring here how the variations

[7] Their experiments showed that 4 probes suffice for $\epsilon \approx 0.03$.

on the insertion algorithm since for our work one can assume the moved key is chosen randomly. In addition, since the exact insertion complexity of the various insertion algorithms is not fully analyzed, we use the fact that it is proved to take an amortized $O(1)$ time when we estimate the vulnerability. In addition, we use simulations give precise results for selected examples.

4.2 In-Attack Damage: Attack on Cuckoo Hashing

The basic idea behind the attack is to insert keys that all hash into the same small number of buckets such that $K > B$. Except for the first $|B|$ keys, every attack key causes an insertion loop in which every move triggers another. Formally, the attacker inserts K keys such for every key k, $\{h_i(k)\}_{i=1}^{d} \subset B$ where B is a group of buckets such that $K > |B|$. Such attack creates insertion loops and cause the insertion algorithm to require excessive number of memory accesses. Note that the size of B can be is as low as d (when B contains exactly one bucket from each sub-table) but large buckets set B allows the attacker to find keys for the attack more easily. Note that a general algorithm that will detect every possible loop can be proved impractical, especially in hardware. Therefore, the most popular approach to address this issue is to limit the number of moves to a predefined fixed value W since their vast majority is very low[8].

4.3 Feasibility of the Attack

Although the hash functions are assumed known to the attacker, it might be infeasible for him to find the right keys if it takes too long. In this section we evaluate the number of keys the malicious has to probe. In a similar way to attack on Peacock hash table, we describe two search algorithm, one is simple and leads to an attack key set with minimal overhead keys while the other is faster but may require a larger number of overhead keys.

In the most simple key-selection algorithm the attacker targets d buckets, one from every sub-table $B = \{b_i \in T_i\}_{i=1}^{d}$ ($|B| = d$). Then he starts probing random keys, and adds to the attack key set every key k that hashes into these target buckets $\wedge_{i=1}^{d} h_i(k) = b_i$. The attacker stops after finding $K \geq d$ attack keys. The probability for a random key to be hashes into specific bucket in each table is $(d/M)^d$. In addition, during the attack itself, inserting the first d keys, is not expected to cause an insertion loop (unless by chance there already exists in the table a key that also hashes exactly into these d buckets). Therefore, we count them as overhead keys. The remaining $K' = K - d$ keys are expected to be effective and require W moves each. Therefore, in order to find K' effective keys, the attacker is expected to probe $(K' + d)(M/d)^d$ keys. We can see that the complexity of finding the attack key-set increases exponentially with d.

In order to reduce the complexity, the attacker can increase the number of targeted buckets. In this approach, which is a generalization of the previous key selection algorithm, instead of choosing one bucket in each sub-table the attacker

[8] W is supposed to be $alog(n)$ where a is appropriately chosen constant [14].

target a group of g buckets in every sub-table. Formally, the targeted buckets set is $B = \cup_{i=1}^{d} G_i$ where $G_i \subset T_i$ and $|G_i| = g$ ($|B| = gd$). Note that the group sizes do not have to be equal, but it can be proved that it is less efficient. Then, when the attacker probes for keys, a key k is added to the attack key set if $\wedge_{i=1}^{d} h_i(k) \in G_i$. The attacker stop searching for keys only after there are $K > gd$ keys in the attack pool (in order to trigger looped insertions, the attacker has to insert more keys than what the buckets can hold).

Theorem 3. *When using target groups of size g in every sub-table, the expected number of keys the attacker has to probe before finding a suitable key for the attack is*

$$C_{key}^{cuckoo} = (M/gd)^d \tag{10}$$

and the number of overhead keys is

$$K_{o/h}^{cuckoo} = gd. \tag{11}$$

Proof. The probability for a random key to hash into the target bucket group of every table is $(gd/M)^d$, therefore the expected number of keys to be probed before finding one key for the attack is $C_{key} = (M/gd)^d$. The number of overhead keys cannot exceed gd keys since the total number of buckets in all sub-tables is gd which is not enough to store $gd + 1$ keys. Therefore, no more than gd keys can be inserted without causing an insertion loop. □

The number of overhead keys cannot exceed gd keys (and cannot be lower than d). Therefore, when choosing the value of g, the attacker has to take into consideration that it is expected to require the probing of $K_{o/h} = (gd)(M/gd)^d$ to find the overhead keys before moving to look for the first effective key.

Note that due to the different structure of Peacock and Cuckoo hash tables and the different damage the attacks on them cause (In-attack and Post-attack damage) the attacks and their feasibility cannot be compared directly. Nevertheless, we can still observe that while in the attack on Cuckoo Hashing every key has to hash into the target bucket(s) in all d sub-tables, in Peacock Hashing it has to hash only to the target bucket(s) in each of the first j tables where j can be low as 1. So while the exponent for $C_{key}^{cuckoo} = (M/gd)^d$ is fixed and decided by the system designer, the exponent j in $C_{key}^{peacock} = \prod_{i=1}^{j}(M_i/|S_i|)$ (Equation 5)is smaller than d and decided by the attacker. We believe that in most cases the exponent is the most important factor in the complexity of finding attack keys and shadows other factors when attacking large tables, we conclude that in general, it is harder to find attack keys for an attack on Cuckoo Hashing. Nevertheless, as the following remark explains, that unlike the attack on Peacock Hashing, the attacker can cause a durable attack against Cuckoo Hashing by re-using the attack key.

Remark 2 (Size of the Attack Set and the Length of the Attack). Another approach that can be used by the attacker to reduce the complexity of preparing the attack is to use the attack keys more than once. After inserting all the K

attack keys set, since they are all hash into the same dg buckets, only dg of them are stored in the table[9]. When inserting a key that already exists in the table, the insertion algorithm immediately detects that it already exists and not try to insert it. Therefore, when inserting the K keys again, dg of them do not cause insertion loops while the rest $K - dg$ keys will do. Hence, using this method, the attacker can theoretically insert the same K keys over and over again and conduct a durable, if not infinite, attack (as long as it is not detected) even with a small attack key set. The larger K is, the longer it takes for the attacker to find the keys (as we already analyzed) but also the smaller is the fraction of the overhead keys dg/K in the attack.

4.4 In-Attack Damage: Vulnerability of Cuckoo Hashing

Following the vulnerability described in Section 2, we now measure the vulnerability of the insertion complexity of a new key into the table (measured by the number of moves it triggers). Let IC_R and IC_A ('R' - Regular, 'A' - Attacker) be the expected total complexity during the insertion of K keys by the attacker and regular users respectively. Note that regardless of the strategy of the attacker, the vulnerability cannot exceeds W ($V_{IC(K)} = \frac{IC_A(K)}{IC_R(K)} \leq W$) since $W \geq IC_A(K), IC_R(K) \geq 1$.

Theorem 4. *The vulnerability of the insertion complexity is given by*

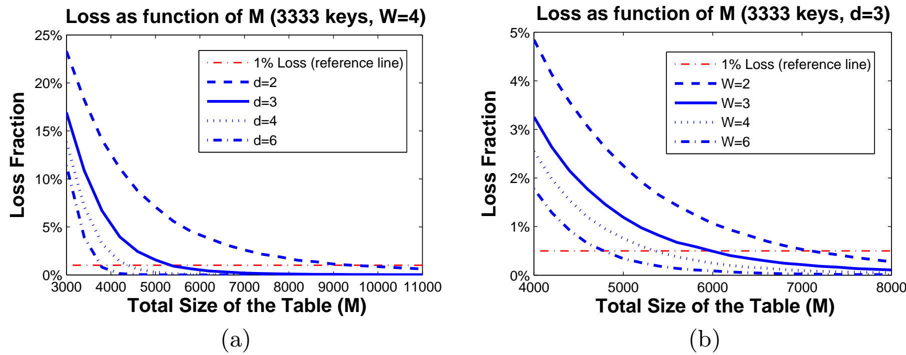
$$V_{IC(K)} = \frac{IC_A(K)}{IC_R(K)} = |B|/K + (1 - |B|/K)\frac{W}{c}, \tag{12}$$

where $c \geq 1$ is the average insertion complexity of a regular key and B is the group of buckets in which all the attack keys hash into.

Proof. Since the insertion time of a random regular key has an amortized $O(1)$ complexity we can conclude that $IC_R(K) = Kc$ where $c \geq 1$ is a constant (very close to 1 in practice) denoting the insertion complexity of a random key. The first $|B|$ keys inserted by the attacker might not cause insertion loops since they all can find an empty bucket. Therefore the complexity of their insertion is $|B|c$. Each of the remaining $K - |B|$ keys inserted by the attacker triggers an insertion loop that is terminated only after W moves. Then $IC_A(K) = |B|c + (K - |B|)W$. To conclude, the results for $IC_R(K)$ and $IC_A(K)$ lead to the result in Eq. 12. □

In the same way as we did for Peacock Hashing, we aim to construct (and then attack) hash table that can hold up to 3333 keys with an *Acceptable Loss Fraction* of 1%. As a system designer would do, we ran simulations with various combinations of M, d and W values. Figure 4 depict loss fraction as a function of those values. Note that the efficiency of Cuckoo Hashing is not the topic of this work and the results brought here are used to compare the role of the different system parameters in its efficiency and its vulnerability.

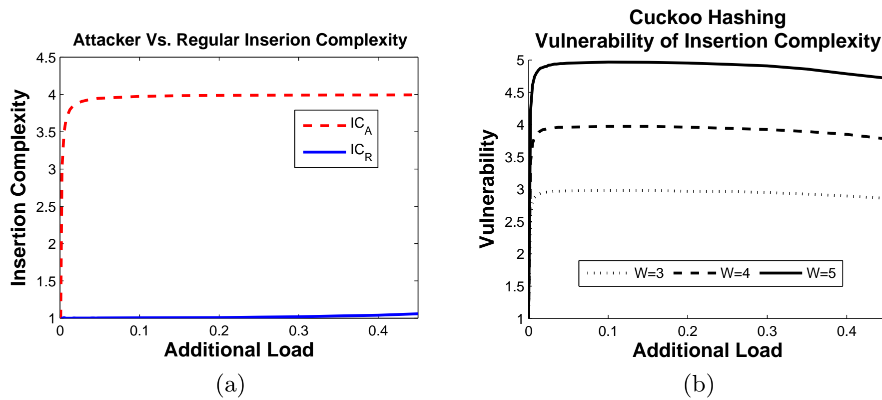
[9] It is unpractical assumption to say the attacker knows which keys were dropped and which not.



(a) (b)

Fig. 4. write caption

As expected, we can see in plots 4(a) and 4(b) that higher values of d and W allow smaller tables to be accommodate up to 3333 keys with the required loss fraction. However, large d and W means a compromise in the performance of the table - large d increases the complexity of Search and Delete operations in the table and large W means increased average insertion complexity which decreases the throughput of the system. When evaluating the vulnerability, we used the values $d = 3$, $W = 4$ and $M = 6000$ (M/d is the size of a sub-table) which keep the fraction rate below 1%. Figures 4(a) and 4(b) depict the simulation results on a table with (arbitrarily chosen) existing load of $\alpha = 0.1$. The x-axis in both figures corresponds to the additional load of keys that was added to the table. The x-axis ends at 0.45 since the maximal allowed load in the table is $3333/6000 = 0.55\%$ (and it already contains 0.1).



(a) (b)

Fig. 5. Plot (a): The average insertion complexity of keys inserted by an attacker $IC_A(K)/K$ and a regular user $IC_R(K)/K$ in a system with $W = 4$. Plot (b): The vulnerability ($V = IC_A(K)/IC_R(K)$) as a function of the additional load (K/M). In both plots the existing load is $\alpha = 0.1$.

According Theorem 4 the vulnerability is mainly defined by W/c (since $|B|/K$ is expected to be very small, especially when $|B| = d$). Note that using $c = 1$ gives an upper bound on the estimated vulnerability which is then approximated by W. As we can see in Figure 5(a), the lower curve which depicts $IC_R(K)/K$ (the average complexity of regular insertion) - c is indeed very close to 1 in the simulation results. Therefore, as also shown in Figure 5(b), setting the moves limit W practically decides the vulnerability of the system. In addition, one can observe that, unlike Peacock Hashing, in Cuckoo Hashing the size of the table M has no role in the vulnerability (Theorem 4).

The following table summarize the results so far. The arrows marks how different properties of a Cuckoo hash table are affected when increasing W, d and M. $\Uparrow$, $\Downarrow$ and $\Leftrightarrow$ mark that a property is increased, decreased or do not chance (respectively).

	High W	High d	High M
Table Utilization	$\Uparrow$	$\Uparrow$	$\Downarrow$
Vulnerability	$\Uparrow$	$\Downarrow$	$\Leftrightarrow$
Attack Feasibility	$\Leftrightarrow$	$\Downarrow$	$\Downarrow$

As summarized in the table above, from the efficiency point of view, the system designer had to choose between increasing W and d in order to increase the table utilization. As explained above, increasing d and W decrease the performance of the system. Our work, shows through the above analysis and simulation results the security implications of setting W and d. The results show that W is a key factor in the vulnerability of Cuckoo Hashing. Therefore, from the vulnerability point of view, it is preferred to increase the number of sub-tables d in order to keep the moves limit W as low as possible. Increasing d not only decreases the vulnerability but also decrease the feasibility by forcing the attacker to invest more efforts in finding a suitable attack keys set.

4.5 Resilience to Post-Attack Damage

There is no general way to cause post attack damage (as we did in Peacock) since there is no specific layout of elements in the table that Cuckoo is vulnerable to more than others since it is unpractical to assume the attacker knows where new keys will hashed to after the attack has ended.

5 Summary

In this work we exposed the weak points of the Peacock and Cuckoo Hashing. we showed that Peacock is resilient against in-attack damage. Nevertheless, we showed that it is vulnerable to an attack that increases the discard probability of a newly inserted key after the attack has ended to be 100 to $10,000$ times higher than after the same amount of keys are inserted by regular users. For Cuckoo hashing we showed that an attacker can slow the system by inserting keys that require 4 times more memory accesses than regular keys in a typical

settings. We also provided simulation results for a use case in which a system designer plans to design a Cuckoo and Peacock hash tables which comply with the same requirements.

6 Appendix

Variables Glossary	
T_i	Sub-table # i
d	Number of sub-tables
M_i	Number of buckets in T_i ($M_i = \|T_i\|$)
M	Total number of buckets ($M = \sum_{i=1}^{d} M_i$)
α	The existing load (keys/buckets) in the table (prior an attack)
K	Number of additional keys inserted by a malicious/regular user
r	$r = M_i/M_{i+1}$ (in Peacock Hashing)
W	The maximal number of sub-insertions allowed (in Cuckoo Hashing)

References

1. Smith, R., Estan, C., Jha, S.: Backtracking Algorithmic Complexity Attacks Against a NIDS. In: Proceedings of ACSAC Annual Computer Security Applications Conference (2006)
2. Crosby, S., Wallach, D.: Denial of Service via Algorithmic Complexity Attacks. In: Proceedings of USENIX Security Symposium (2003)
3. Kumar, S., Turner, J., Crowley, P.: Peacock Hash: Fast and Updatable Hashing for High Performance Packet Processing Algorithms. In: Proceedings of IEEE INFOCOM (2008)
4. Pagh, R., Rodler, F.: Cuckoo Hashing. Journal of Algorithms (2001)
5. Mitzenmacher, M., Broder, A.: Using Multiple Hash Functions to Improve IP Lookups. In: Proceedings of IEEE INFOCOM (2000)
6. Song, H., Dharmapurikar, S., Turner, J.,Lockwood, J.: Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In: Proceedings of ACM SIGCOMM (2005).
7. Waldvogel, M.,Varghese, G., Turner, J., Plattner, B.: Scalable High Speed IP Routing Lookups. In: Proceedings of ACM SIGCOMM (1997)
8. Thinh, T., Kittitornkun, S.: Massively Parallel Cuckoo Pattern Matching Applied for NIDS/NIPS. In: Proceedings of IEEE DELTA (2010)
9. Kirsch, A., Mitzenmacher, M., Varghese, G.: Hash-Based Techniques for High-Speed Packet Processing. Algorithms for Next Generation Networks, Springer (2010)
10. Ben-Porat, U., Bremler-Barr, A., Levy, H.: Evaluating the Vulnerability of Network Mechanisms to Sophisticated DDoS Attacks. In: Proceedings of IEEE INFOCOM (2008)
11. Ben-Porat, U., Bremler-Barr, A., Levy, H., Plattner, B.: On the Vulnerability of Hardware Hash Tables to Sophisticated Attacks. Technical Report. http://www.faculty.idc.ac.il/bremler/ (2011)
12. Kirsch, A.,Mitzenmacher, M., Wieder, U.: More Robust Hashing: Cuckoo Hashing with a Stash. In: Proceedings of European Symposium on Algorithms (ESA) (2008)

13. Fotakis, D., Pagh, R., Sanders, P., Spirakis, P.: Space Efficient Hash Tables With Worst Case Constant Access Time. In: Proceedings of STACS (2003)
14. Frieze, A., Melsted, P., Mitzenmacher, M.: An Analysis of Random-Walk Cuckoo Hashing. In: Proceedings of APPROX/RANDOM (2009)
15. Kirsch, A., Mitzenmacher, M.: The Power of One Move: Hashing Schemes for Hardware. IEEE/ACM Transactions on Networking, 18(6), 1752–1765 (2010)
16. Estan, C., Keys, K., Moore, D., Varghese, G.: Building a Better NetFlow. In: Proceedings of ACM SIGCOMM (2004)