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 hash 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 ($100 - 1,800$ 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.

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 are 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).

R. Bestak et al. (Eds.): NETWORKING 2012, Part I, LNCS 7289, pp. 135–148, 2012.

In another example, 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.

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 (to which it hashes) in each sub-table. When a key k finds all its buckets occupied, one of the keys residing in those buckets is then moved to one of its alternate locations to free the bucket for k. Cuckoo Hashing, therefore, allows achieving a higher table utilization than that achieved by alternative MHT schemes that do not allow moves, while maintaining $O(1)$ amortized complexity of an Insert operation (although 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 show that Peacock is resilient against in-attack damage and 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 discard probability for a newly inserted key can be 100 to 1,800 times higher than the discard probability 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. We further analyze the vulnerability with respect to two key design parameters – the number of sub-tables and the number of moves allowed per insertion; we show that while the utilization in the table increases with either of these parameters, the vulnerability decreases with the former while increases with the latter. In addition to mathematical analysis, we also provide simulation results for a use case in which a system designer plans to design 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 at 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 an available 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. Therefore, a simple flooding attack where the attacker simply inserts a large number of (random) keys cannot degrade the system performance beyond its acceptable limits. Note that a flooding attack can be handled by forwarding the keys to another table/device or by blocking the attacker since it can then be detected. 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 number of keys and hence avoid reaching the maximal load in the system.

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 malicious and regular users, respectively, where 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 due to 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 sizes of the sub-tables follow a decreasing geometric sequence $M_{i+1} = M_i/r$ where r is the proportion between the table sizes[1]. 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 to 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 memory). It is used to avoid checking all the sub-tables when making sure an inserted key does not already exist in the table. Note that due to lack of space,

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

in this work we follow the original Peacock and Cuckoo models and exclude the case in which a bucket can hold more than one key; this case is discussed in a technical report [11].

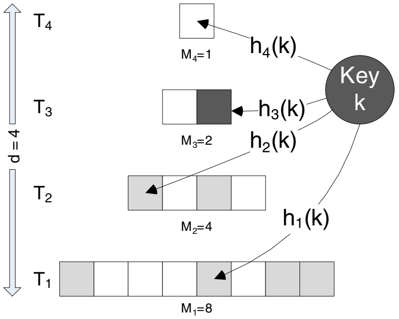


Fig. 1. An insertion of key k into a small Peacock hash table with $r = 2$ and $d = 4$. Gray buckets are occupied with existing keys and white buckets are free. The black bucket marks the bucket where k is finally placed.

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 bigger 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 solved by re-balancing (unlike a natural unbalanced state that occurs over time).

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, Peacock hash table maintains a desired low discard probability by limiting the maximal load in the table. Nevertheless, an attacker using the above sophisticated attack algorithm can cause the discard probability to exceed its desired value by inserting only a small number of keys that do not cause the table to reach its maximal load. 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 to which they all hash 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[2].

Denote the set of buckets to which a key is hashed 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 $1/M_i > 1/M_j$). Therefore, the basic idea behind the attack is to insert keys that will be placed in the upper tables. Following is the formal description of the attack algorithm, generalizing the simple attack described earlier. An attack in depth j is an attack to 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. Due to lack of space, the complexity of finding the keys is excluded from this work (and can be found in [11]). Note just that for large tables (M) and for deeper attacks (larger j) - it is harder for the attacker to find the keys for the attack.

3.3 Post-Attack Damage: Vulnerability of Peacock Hashing

We use the Vulnerability factor (described in Section 2) to measure the proportion between the increase in the discard probability caused by additional keys inserted by an attacker and regular users. Let DP_I, DP_R and DP_A ('I' - Initial, 'R' - Regular, 'A' - Attacker) be the expected discard probabilities of newly 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. Note that due to their length, we exclude from this work the proofs and the full discussion of the following claims and they can be found in our technical report [11].

Lemma 1. *The drop probability after regular key insertion is approximated by*

$$DP_R(K) \sim (\alpha + R^{IN}/M)^d, \tag{3}$$

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

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

Lemma 2. *The drop probability after a sophisticated attack is approximated by*

$$DP_A(K) \sim [\prod_{l=1}^{j}(\alpha + (1-\alpha)/M_l)](\alpha + A^{IN}/M')^{d'}, \tag{4}$$

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

Theorem 1. *The vulnerability of the discard probability is given by*

$$V_{DP}(K) = \frac{DP_A(K) - DP_I}{DP_R(K) - DP_I}, \tag{5}$$

where $DP_I = \alpha^d$*.*

R^{IN} (Eq. 3) and A^{IN} (Eq. 4) can be roughly described as the number of keys that remain in table after the K keys were inserted by regular users and an attacker, respectively. Practically, the values of R^{IN} and A^{IN} are very close to each other and to K. The major factor that makes the value of $DP_A(K)$ (Eq. 4) significantly higher than $DP_R(K)$ (Eq. 3) is that R^{IN} is divided by M while A^{IN} is divided only by M'. This is because keys inserted by attacker are spread among the M' of the attacked backup tables while keys inserted by regular users are spread among all the M buckets in the table. Since $M_i = r^{d-i}$, $M' = \sum_{i=j+1}^{d} M_i$ is only a small fraction of $M = \sum_{i=1}^{d} M_i$ and this is the major factor behind the differences between $DP_A(K)$ and $DP_R(K)$ as seen in Figure 2(a).

Evaluation of the Vulnerability and the Analysis. To evaluate the vulnerability of the system as a function of the system parameters and to evaluate the quality of the approximations (lemmas 1 and 2) we next conduct a set of simulations. The simulations we conducted follow a scenario in which the system designer examines the option of using Peacock Hashing for hardware that can support approximately 10^5 buckets in 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 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.

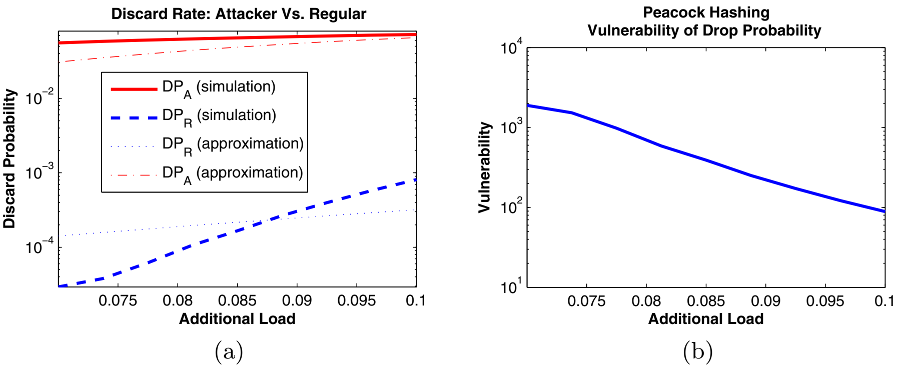


Fig. 2. Figure (a): The discard probability after insertions by regular (DP_R) and malicious (DP_A) users as a function of the load they add (K/M). Figure (b): The vulnerability simulation results (V) of a table with proportion $r = 10$ and $d = 4$ with an existing load of 10% as a function of the additional 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 $j = 1$. 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 which are already occupied prior the attack. Therefore, an attacker would insert no more than 10% of additional load. Therefore, the range of the x-axis in Figures 2(a) and (b) was chosen accordingly. Note that the additional 0.1 load together with the existing load $\alpha = 0.1$ does not exceed the maximal load in the table which is 0.3. We can see in Figure 2(a) that when the additional load is 7%, the discard probability remains very low (below 0.1%), while the sophisticated attack causes the discard probability to increase to values between 5.5% (when the additional load is 0.07) and 7.2% (when the additional load is 0.1), a discard probability that is achieved by regular insertions only with load which is 3.5 times larger. In addition we can see that, as expected, the approximation curves in Figure 2(a) reflect the behavior of DP_R and DP_A although do not imitate them precisely (see [11] for more details).

We can see a significant difference in the discard probability after a malicious attack and after regular insertions. 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 1,800 times larger than the discard probability after the insertion of the same load of keys by regular users. This drives the discard probability of the hash table far beyond the discard probability in which it is assumed to be operating. This result emphasizes the fundamental vulnerability of Hashing schemes, such as the Peacock Hashing, that dedicate specific range of buckets (the upper sub-tables) for collision resolution.

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

As already mentioned in Section 1, we analyze an attack focused at creating *post-attack* damage. We avoided analyzing attacks aiming at *in-attack* damage by inserting keys that require excessive number of memory accesses during the attack, since we believe such attacks can be countered easily. 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 summary. 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 addition 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, causing the key placed in the occupied bucket to *move* to its alternative bucket in the same manner. Therefore, every insertion of a new key consists 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 into which it hashes (dark left arrows) are all occupied. Then, the insertion algorithm chooses to eject k_1 (from T_4) and place k_2 in its place. k_1 is then relocated to an alternative free bucket to which it hashes 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.

For the original Cuckoo hash table consisting 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

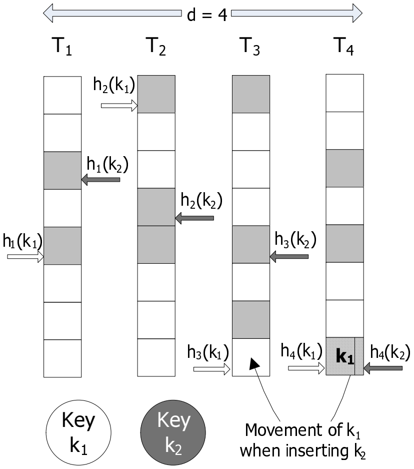


Fig. 3. A move of key k_1 during the insertion of k_2 into a Cuckoo hash table with 4 sub-tables. The white (right) and the dark (left) arrows mark the hash values of k_1 and k_2 in the different sub-tables, respectively.

for any constant $\epsilon > 0$. They showed that Search and Delete operations take[3] $O(ln(1/\epsilon))$ 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 later 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 discuss here the variations of the insertion algorithm and 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 approximate the vulnerability. In addition, we use simulations to give precise results for selected examples.

4.2 In-Attack Damage: Attack on Cuckoo Hashing

The basic idea behind the attack is to insert K keys that all hash into the same small set of buckets B, 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 set 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

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

from each sub-table) but a large bucket 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[4].

In our technical report [11] we describe the algorithm used by the attacker to find the keys for the attack and its complexity is analyzed. Our main result on this subject is that it is harder to find Cuckoo attack keys than to find the Peacock attack keys[5]. However, on the other hand, in contrast to the Peacock attack, the attacker can use the same keys over and over again and sustain his attack until it is mitigated, if at all.

4.3 In-Attack Damage: Vulnerability of Cuckoo Hashing

Using the vulnerability measure described in Section 2, we now evaluate the system vulnerability due to new-key insertion complexity (measured by the number of moves it triggers). Let IC_R and IC_A ('R' - Regular, 'A' - Attacker) denote the overall number of operations performed during the insertion of K keys, by the attacker and by 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 trivially $IC_R(K) >= K$ and since the system enforces $IC_A(K) <= KW$.

Theorem 2. *The system vulnerability due to insertion complexity is*

$$V_{IC(K)} = \frac{IC_A(K)}{IC_R(K)} = |B|/K + (1 - |B|/K)\frac{W}{c}, \tag{6}$$

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

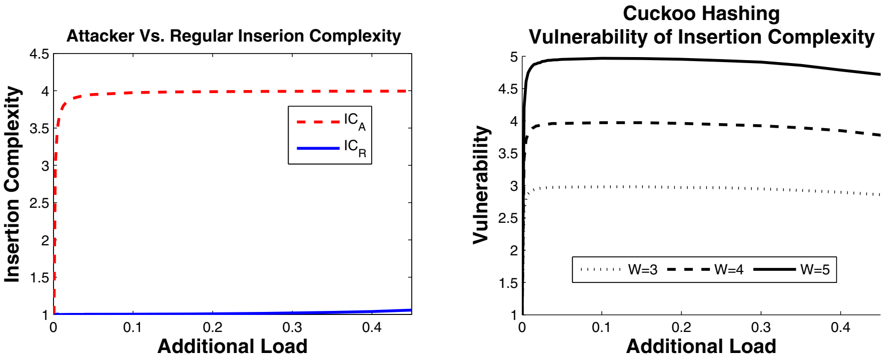
Proof. Since the insertion time of a random regular key has an amortized $O(1)$ complexity we can conclude that $IC_R(K) = Kc$ where $c \geq 1$ is a constant (very close to 1 in practice) denoting the insertion complexity of a random key. The first $|B|$ keys inserted by the attacker might not cause insertion loops since they all can possibly 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 values of $IC_R(K)$ and $IC_A(K)$ lead to the result in Eq. 6. □

Evaluation of the Vulnerability and the Analysis. 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

[4] W suppose to be set to $alog(n)$ where n is the number of keys the table can hold and a is appropriately chosen constant [14].

[5] Since keys have to hash to the same buckets in all sub-tables.

system designer would do, we used simulations to measure the loss fraction with different values of d, W and M and decided to use a system with $d = 3$, $W = 4$ and $M = 6000$. Note that higher values of d and W allow higher utilization of the table[6]. That is, smaller value of M is required in order to accommodate up to 3333 keys with the required loss fraction.



(a) (b)

Fig. 4. 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 $\alpha = 0.1$.

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).

According Theorem 2 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 4(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 4(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 2).

The following table summarize the results so far, including those from the attack feasibility analysis which is excluded from this paper and can be found in a technical report [11]. 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).

[6] However, high values of d and W also increase the complexity of the different operations in the table.

	High W	High d	High M
Table Utilization	⇑	⇑	⇓
Vulnerability	⇑	⇓	⇔
Attack Feasibility	⇔	⇓	⇓

As explained above, from the efficiency point of view the system designer has to choose between higher complexity of the hash operations (high values of d and W) and lower utilization (large M). 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 preferable 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 effort in finding a suitable attack keys set (see [11]).

4.4 Resilience to Post-Attack Damage

There is no general way to cause post attack damage (In contrast to Peacock Hashing, see Section 3.2) since there is no specific layout of elements in the table that Cuckoo is vulnerable to more than other; This is true since it is impractical 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 highly vulnerable to an attack that aims at driving the system to high post-attack discard rates; such an attack can increase these rates by a factor of $100 - 1,800$ in comparison to normal behavior. 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.

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 (2011), `http://www.faculty.idc.ac.il/bremler/`
12. Kirsch, A., Mitzenmacher, M., Wieder, U.: More Robust Hashing: Cuckoo Hashing with a Stash. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 611–622. Springer, Heidelberg (2008)
13. Fotakis, D., Pagh, R., Sanders, P., Spirakis, P.: Space Efficient Hash Tables with Worst Case Constant Access Time. In: Alt, H., Habib, M. (eds.) STACS 2003. LNCS, vol. 2607, pp. 271–282. Springer, Heidelberg (2003)
14. Frieze, A., Melsted, P., Mitzenmacher, M.: An Analysis of Random-Walk Cuckoo Hashing. In: Dinur, I., Jansen, K., Naor, J., Rolim, J. (eds.) APPROX and RANDOM 2009. LNCS, vol. 5687, pp. 490–503. Springer, Heidelberg (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)

A Appendix

Glossary of Notations	
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)
j	Attack depth (in Peacock Hashing)
W	The maximal number of moves allowed (in Cuckoo Hashing)