Reviriego, Pedro; Ting, Daniel (2022). Breaking Cuckoo Hash: Black Box Attacks. *IEEE Transactions on Dependable and Secure Computing*, 19(4), pp.: 2421-2427.

DOI: https://doi.org/10.1109/TDSC.2021.3058336

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3058336, IEEE Transactions on Dependable and Secure Computing

Breaking Cuckoo Hash: Black Box Attacks

Pedro Reviriego[1] and Daniel Ting[2]

Abstract—Introduced less than twenty years ago, cuckoo hashing has a number of attractive features like a constant worst case number of memory accesses for queries and close to full memory utilization. Cuckoo hashing has been widely adopted to perform exact matching of an incoming key with a set of stored (key, value) pairs in both software and hardware implementations. This widespread adoption makes it important to consider the security of cuckoo hashing. Most hash based data structures can be attacked by generating collisions that reduce their performance. In fact, for cuckoo hashing collisions could lead to insertion failures which in some systems would lead to a system failure. For example, if cuckoo hashing is used to perform Ethernet lookup and a given MAC address cannot be added to the cuckoo hash, the switch would not be able to correctly forward frames to that address. Previous works have shown that this can be done when the attacker knows the hash functions used in the implementation. However, in many cases the attacker would not have that information and would only have access to the cuckoo hash operations to perform insertions, removals or queries.

This paper considers the security of a cuckoo hash to an attacker that has only a black box access to it. The analysis shows that by carefully performing user operations on the cuckoo hash, the attacker can force insertion failures with a small set of elements. The proposed attack has been implemented and tested for different configurations to demonstrate its feasibility. The fact that cuckoo hash can be broken with only access to its user functions should be taken into account when implementing it in critical systems. The paper also discusses potential approaches to mitigate this vulnerability.

Index Terms—Cuckoo hash, key value store, security, vulnerability.

1 Introduction

Many computing and networking applications need to store a set of (key, value) pairs and check incoming keys against those stored in order to retrieve or update the value associated with a key [1], [2]. Cuckoo hashing [3] is a widely used algorithm to implement key value store that has a low and constant worst case number of memory accesses for queries and that achieves close to full memory utilization in practical configurations [3], [4]. Instead, insertions are more complex and can take a large number of memory accesses when occupancy is high. These features make cuckoo hashing attractive for workloads on which queries or updates dominate over insertions. This is, for example, the case in most networking applications [5]. Another advantage of cuckoo hashing is that it is amenable to hardware implementation, and in fact, it is widely used on both FPGAs [2] and ASICs [6]. Software implementations are also available for example for networking applications [5], [7]. For a software implementation, the cuckoo algorithm can be optimized to try to place the most frequently used (key, value) pairs on the hash tables that are accessed first, thus reducing the average number of memory accesses needed per query/update operation [8].

Security is one of the main requirements for key-value stores [9]. The use of data structures in general, and of probabilistic or hashing based algorithms in particular, poses some security risks when an attacker tries to disrupt the system [10]. For example, attacks on Bloom filters where the adversarial tries to degrade the false positive rate or to create false positives for specific elements have been widely studied [11], [12], [13]. The security of HyperLogLog, a commonly used data structure for cardinality estimation has also been recently considered in [14]. In the case of hash tables, it has been shown that if the attacker has access to the hash functions used, he can disrupt their functionality in different ways [15] and techniques to try to identify the hash function used have been presented [16]. In particular, for cuckoo hashing or cuckoo filters an attacker can create a small set of elements which creates an insertion failure when added to the tables [15], [17]. This failure can have different effects depending on the cuckoo hash implementation and on the application for which it is used. For example, in a software implementation the tables may be resized to accommodate more elements or rebuilt using other hash functions. This requires a non negligible amount of time during which the functionality is degraded. In many hardware implementations, the table sizes are fixed and rebuilding the tables cannot be easily done. As for the application, when cuckoo hashing is used in high performance switches [6], an insertion failure can create a functional failure where the switch can no longer ensure a correct processing of all packets. For example, when cuckoo hashing is used to retrieve the output port of a frame based on its destination address [7], the switch would not be able to find a matching entry for it, if that address cannot be inserted into the cuckoo hash. In other cases, an insertion failure may just lead to a performance degradation.

In many cases, the attacker will not have access to the hash functions used in the cuckoo hash implementation and thus cannot force an insertion failure using the attacks described in existing works. For the cuckoo filter, recent work gives an insertion attack which can be done even when the hash functions are unknown by the attacker by using false

This work was supported by the ACHILLES project (PID2019-104207RB-I00) and the Go2Edge network (RED2018-102585-T) funded by the Spanish Ministry of Science and Innovation and by the Madrid Community project TAPIR-CM (P2018/TCS-4496).

[1]*Pedro Reviriego is with Universidad Carlos III de Madrid, Leganés 28911, Madrid, Spain. email:* revirieg@it.uc3m.es

[2]*Daniel Ting is with Tableau Software, Seattle, Washington, USA. email:* dting@tableau.com

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3058336, IEEE Transactions on Dependable and Secure Computing

positives to detect colliding elements [17]. However, this attack cannot be applied to cuckoo hashing as it produces no false positives.

This paper considers the case where an attacker that can only use the cuckoo hash as a black box and perform insertion, removal and query operations. This is a much weaker assumption than knowledge of the hash functions used for implementation. In fact, any user of the cuckoo hashing would have access to those operations. For this more general adversarial model, it is shown that insertion failures can also be created with a complexity that is similar to that of existing attacks that assume the knowledge of the hash functions [15]. This is worrying as cuckoo hashes are widely used in many modern systems. Exposing this vulnerability of cuckoo hashing motivates the need for having security as a design requirement in cuckoo hashing. The paper also discusses potential schemes to mitigate or avoid the attack and suggests some directions for future work.

The rest of the paper is organized as follows. Section 2 covers the preliminaries on both cuckoo hashing and attacks on it. The proposed attack is presented and analyzed in Section 3. Section 4 presents the evaluation of the proposed attack in a cuckoo hash implementation and Section 5 provides some discussion on other types of attacks and techniques that can be use to mitigate or avoid them. Finally, the paper ends with the conclusion and some ideas for future work on Section 6.

2 Preliminaries

2.1 Cuckoo hash

In cuckoo hashing, each element x is mapped to d positions in a table using a set of hash functions $h_i(x)$ [3]. Each position on the table is a bucket that has c cells (each capable of storing an element). This is illustrated in Figure 1. The table has M buckets and thus can store a maximum of $S = M \cdot c$ elements. The operations supported are: query, update, insertion and removal.

To query for an element y, the hash functions $h_i(y)$ are computed and the corresponding buckets are accessed. If any of the stored elements is equal to y, there is a match and the value stored in that cell is returned. The process for updates and removals is similar but once the matching element is found, its value is updated or the element is removed. The most complex operation is insertion. To insert an element y, the hash functions $h_i(y)$ are computed first. The corresponding buckets are then checked to see if there are empty cells, and if so, y is stored in one of them. However, if all those buckets are full, y kicks out an element z stored in one of those buckets, and an attempt to re-insert z in its remaining buckets is made. Depending on the cuckoo table's occupancy, many such movements may be needed to find an empty cell. In many implementations, a limit t_{limit} is set for the number of movements and when it is reached, an insertion failure occurs.

Another option for cuckoo hashing is to have d tables with the corresponding hash function $h_i(x)$ as shown in Figure 2. In this case, up to $S = M \cdot d \cdot c$ elements can be stored in the tables. This multiple table configuration is well suited for hardware implementation. By mapping each table to a different memory, queries can be completed in a single memory access cycle [6]. Both alternatives, single

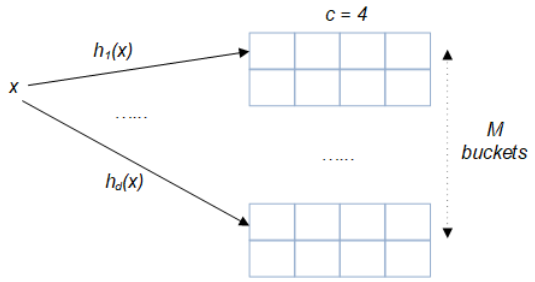

Fig. 1: Cuckoo hash with a single table

and multiple tables, achieve a similar performance in terms occupancy when the tables are large.

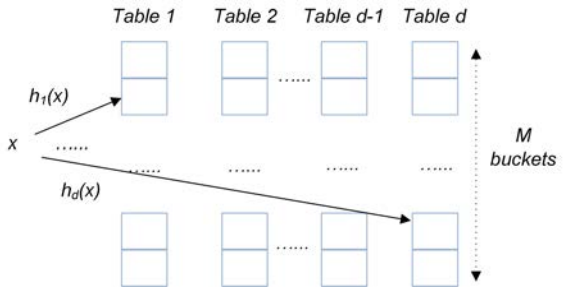

Fig. 2: Cuckoo hash with multiple tables

The occupancy that can be achieved by cuckoo hashing before an insertion failure occurs depends on the number of tables d and the number of cells per bucket c [4]. Two widely used configurations are two tables with buckets of four cells (d = 2, c = 4) and four tables with single cell buckets (d = 4, c = 1), both achieve close to full occupancy. The first one reduces the number of memory accesses in software implementations [5] while the second minimizes the amount of data accessed per query and is useful in hardware implementations [6].

2.2 Attacks on cuckoo hash

Cuckoo hash tables can be attacked by forcing an insertion failure. For hardware implementations which cannot be resized, this puts the hash table into an incorrect state that can disrupt the operation of a system and lead to a failure. For resizable software implementations, this can make the cuckoo hash tables larger than necessary. The security of cuckoo hashing has been studied in [15] where it was assumed that the attacker has access to the implementation details. In particular, the attacker knows the hash functions used $h_i(x)$. Then, the attacker can easily build an attack set $A = \{a_j\}_j$ by selecting the elements a_n to have the same hash value in all d tables, that is $h_i(a_n) = h_i(a_m)$ for $i = 1, 2, ..., d$ and all pairs of elements a_m, a_n in the set. If the attack set A is large enough, namely if $|A| \geq c \cdot d + 1$, then inserting all elements in A would lead to an insertion failure. This is because they map to the same d buckets in the tables with each bucket containing c cells. Since at most $c \cdot d$ elements can be placed in those buckets, trying to place more elements necessarily leads to an insertion failure.

To construct the set, elements can be generated randomly and their hash values checked. The expected number of elements that need to be tested can be estimated as:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3058336, IEEE Transactions on Dependable and Secure Computing

$$T = (c \cdot d + 1) \cdot (M)^d, \quad (1)$$

where M is the number of buckets in each table, c the number of cells per bucket and d the number of tables. This clearly shows that implementations that use more tables are harder to attack. Therefore, considering the two commonly used configurations: d = 2, c = 4 and d = 4, c = 1, the later would be more robust.

3 Proposed Attack

In this section, the proposed attack is presented. First, the adversarial model considered is discussed. Then the attack is described and analyzed.

3.1 Adversarial model

In many scenarios, the attacker may not have access to the cuckoo hash implementation details. However, in most cases the attacker will be able to perform operations to insert, remove or query for elements. That is, the attacker can use the cuckoo hash as a black box and perform the operations that a user or client would perform.

We consider the case where the cuckoo hash has fixed size and an insertion failure is a soft failure, so that users are alerted to insertion failures but are allowed to continue using the cuckoo hash even after encountering an insertion failure. We can assume the fixed size S is known, since we also show that it can be inferred otherwise. Finally, it is assumed that there are no restrictions on the number of operations that can be done on the cuckoo hash. We show that even without knowing the hash functions, an adversary can still generate a small attack set that induces an insertion failure with high probability.

3.2 Description

Like [15], our proposed attack forces an insertion failure by generating elements $A = \{a_1, \ldots, a_{cd+1}\}$ that are mapped to the same d buckets spanning the d tables. However, with only the ability to insert, remove, and query items, an attacker has no easy way to check that items belong to the same bucket, and it is unclear how such an attack set can be generated. Our attack does so by isolating a set of d buckets where every successful insertion maps to the same d buckets. The ability to do so is based on the following observation and hypothesis.

- Observation: when the cuckoo hash tables are all full except for one cell, elements that map to that cell are successfully inserted on that cell.
- Hypothesis: when the cuckoo hash tables are all full except for one cell, it is likely that all new elements that do not map to that cell will fail insertion.

The observation is obvious as the insertion for an element that maps to the last empty cell can always place the element on that cell. The hypothesis is far from being obvious, but let us assume for now that it is true. We will later prove it is true with high probability for the basic cuckoo hash with parameters $d = 2$ and $c = 1$. Suppose a cuckoo hash contains only one unoccupied cell as shown in Figure 3. Then, from the observation and the hypothesis, it follows that an element x is successfully inserted if and only if it maps to that last empty cell. Consequently, if we try insertions of elements until one such element x is found and after the successful insertion we remove it and repeat the same procedure until another such element y is found, then both x and y would map to the bucket where the last empty cell is. More formally, this means that we can generate elements x and y with hash values $h_i(x) = h_i(y)$ for some table i, and we can do so using only user operations and without knowing the cuckoo hash implementation's hash function h_i. We can extend this to finding elements that map to the same d buckets in d tables by creating multiple instances of cuckoo hashes where the last empty cell is on different tables.

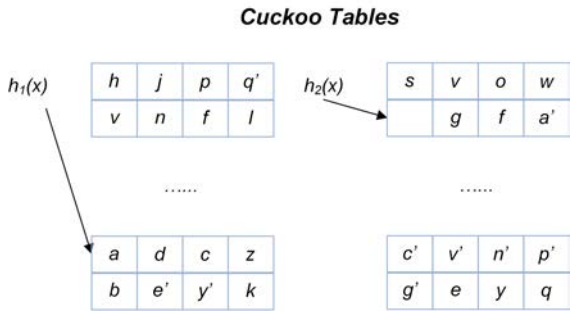

Fig. 3: Cuckoo hash with d = 2, c = 4 filled except for one cell with element x mapping to the bucket of the empty cell on the second table

Before presenting the attack procedure in more detail, we provide the intuition behind the hypothesis and prove it in the case of a basic cuckoo hash where $d = 2, c = 1$. An insertion into a cuckoo hash generates a chain of cuckoo kickout and insertion operations that terminates when an open cell is found. These potential kickout and insertions link together two buckets and can be represented using a graph. If an open cell is the end point of a long chain, then inserting any element that appears along the chain may fill in that open cell. In the basic cuckoo hash with $d = 2, c = 1$, it always fills in the open cell. Since each inserted item randomly chooses d buckets uniformly, long chains with more buckets are more likely to be selected, and the open cell at the end of a long chain is likely to be filled early. A cell that is part of a chain of length 0, in other words, a cell where no other elements map to its bucket (except those already in the bucket when $c > 1$), is likely to be the one of the last cells filled. The hypothesis holds precisely when such a cell is the last open cell.

We now prove that not only is this a likely outcome but it is the outcome with probability approaching 1 when $d = 2, c = 1$ and the table size M is large. Here, we use the single table version of the cuckoo hash which chooses d buckets from a single large table of size $d \cdot M$ and base our analysis on the cuckoo graph.

The cuckoo graph is the graph on $d \cdot M$ buckets where there is an edge drawn between two buckets whenever an item is hashed to those two buckets. As noted by [18], an item y can be successfully inserted if and only if the cuckoo graph containing all items up to and including y, the component containing y has at most 1 cycle. Since each edge corresponds to an item, if a connected component contains v buckets and at least $v + 1$ edges, then all buckets in the component contain an item.

Since all edges have equal probability of being chosen, given the number of edges $|E|$, the cuckoo graph is a form of Erdös-Rényi random graph $\Gamma_{2M,|E|}$ with a fixed number of edges [20]. As the number of edges in an Erdös-Rényi graph increases to $|E| \approx M \log 2M$, the graph becomes one large connected component plus a few isolated vertices. Since for large M, the large connected component has $M \log 2M > 2(M-1)$ edges, it follows that the corresponding cuckoo hash table consists of filled buckets plus a few buckets that have no items mapped to it. In particular, the last empty bucket will have no items mapped to it.

Theorem 1. *Consider a sequence of cuckoo hash tables with $2M \to \infty$ entries, $d = 2$ choices of bucket per item, and $c = 1$ cells for bucket. Let T_{2M} denote the number of items inserted to fill up completely the table of size $2M$. Then, for $\epsilon > 0$,*

$$|T_{2M} - M \log 2M| = o_p(\log^{2+\epsilon} M) \tag{2}$$

or equivalently, $P(|T_{2M} - M \log 2M| < \log^{2+\epsilon} M) \to 0$ as $M \to \infty$. Furthermore, let D_{2M} denote the number of items hashed to the last empty entry when only $T_{2M} - 1$ items are inserted. Then,

$$P(D_{2M} = 0) \to 1 \quad \text{as } M \to \infty. \tag{3}$$

Proof. Here o_p is the probabilistic analog to little-o notation. In other words, $X_n = Y_n + o_p(Z_n)$ means that $(X_n - Y_n)/Z_n \xrightarrow{p} 1$ converges in probability. The proof is a formalization of the argument given above that addresses a few technical issues: the possible duplication of edges that does not arise in Erdös-Rényi graphs and handling the problem where the number of added edges is a random process with a stopping condition. Equation 2 is almost an immediate consequence of theorem 4 in [20]. The primary difference between items inserted and edges in the cuckoo graph is that some items may result in duplicate edges. By showing the number of duplicate edges is not too large, it follows that they can be ignored. The expected number of duplicate edges from v items is $\frac{\binom{v}{2}}{\binom{2M}{2}} = \frac{v(v-1)}{4M(M-1)}$ from the usual birthday paradox calculation. Thus, the number of duplicate edges from $M \log 2M + o(\log^{2+\epsilon} M)$ items is $\log^2 2M + o(\log^2 2M)$. The result on T_{2M} follows from theorem 4 in [20] and Markov's inequality.

To handle the problem of there being some random number of edges until there is only one empty bucket, we first add a sufficiently large constant number of edges so that existing results imply that the graph becomes a giant connected component plus isolated buckets with high probability. By continuing to add edges, we find that the last empty bucket is isolated with high probability.

By theorem 2 in [20], we may choose a sequence $c_{2M} \to -\infty$ and $|c_{2M}| < 1/2 \log\log M$ such that the cuckoo graph with $M \log 2M + c_{2M} \log 2M$ edges consists of a giant connected component and K_{2M} isolated buckets with probability converging to 1 as $M \to \infty$. Furthermore, the number of isolated buckets $K_{2M} \overset{D}{\rightsquigarrow} Poisson(\lambda)$ where $\lambda = e^{-2c_{2M}} < \log M$. The probability that any newly inserted item will connect two isolated buckets is thus less than $\log^2 M/M^2$. Probability the cuckoo graph will be fully connected after $M \log^{1+\epsilon} 2M$ converges to 1 almost surely, a union bound gives that the probability two isolated buckets will ever be connected is $< \log^{3+\epsilon} /M \to 0$, and $P(D_{2M} = 0) \to 1$. □

Therefore, the hypothesis is true with high probability for $d = 2$, $c = 1$. Unfortunately, the general case seems challenging to prove and is left for future work. For other parameter settings, we check by simulation that the hypothesis remains true with high probability in Section 4.

3.3 Attack

Given the validity of the hypothesis, we exploit it to generate elements that collide in one of the tables. Let us now describe the proposed attack using the case where d = 2, c = 4 as an example.

Figure 4 shows the process used for the attack. The first step is to create two cuckoo hash instances using the same hash functions which we denote by a) and b). These are filled with different sets of random elements, stopping when one empty cell remains or, equivalently, when $S-1$ elements have been successfully inserted. Examples of instances a) and b) are shown in Figure 5. In the Top one, the empty cell on instance a) is on the first table and the empty cell on instance b) on the second table. For $d = 2$, these empty cells appear on separate tables with probability $1/2$. If the hypothesis is valid on both instances, a successfully inserted element maps to precisely these two buckets with empty cells. Although the attacker does not know which two buckets these are, by successively inserting into these buckets and removing them immediately, one generates a set of items that map to the same two buckets, quickly leading to an insertion failure. We also generate a third cuckoo hash instance c) using the same hash functions and use it to test the validity of an attack set by ensuring the items that are generated using a) and b) generate an insertion failure. In more detail, a random element t is generated and insertion is tried on the a) instance; if the insertion fails, a new element is generated and the process starts again. If insertion is successful, the element t is removed and insertion is tried on instance b). Again, if the insertion fails, a new element is generated and the process starts again. If the element can be inserted, then it is removed and inserted on a third cuckoo hash instance c). If insertion fails, then the elements stored in instance c) plus t can be used for an attack.

The procedure tries to ensure that the elements inserted on instance c) map to the same buckets on the two tables as discussed before. In fact, this would be the case when the hypothesis holds and the empty cells on a) and b) are on different tables. In that scenario, for the example considered, d = 2, c = 4, once the number of elements on instance c) reaches $d \cdot c = 8$, the next insertion will fail and thus the construction of the attack set is complete.

For the multiple table implementation considered, an attempt to build an attack set may fail. This occurs if the empty cell falls on the same table on instances a) and b) as for example, in the Bottom of Figure 5. In that case, it is not possible for an element to be successfully inserted on instances a) and b) as it would have to map to two different buckets on the first table. Assuming that the empty cell is randomly placed on each table, the probability of this happening would be approximately 50%. Instead, for a single table implementation (see Figure 1), this cannot happen as all buckets are mapped to just one table. Therefore, for

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3058336, IEEE Transactions on Dependable and Secure Computing

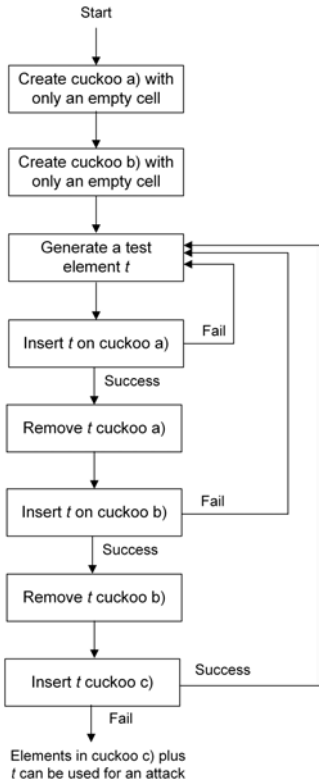

Fig. 4: Flow diagram of the proposed attack for $d = 2$, $c = 4$

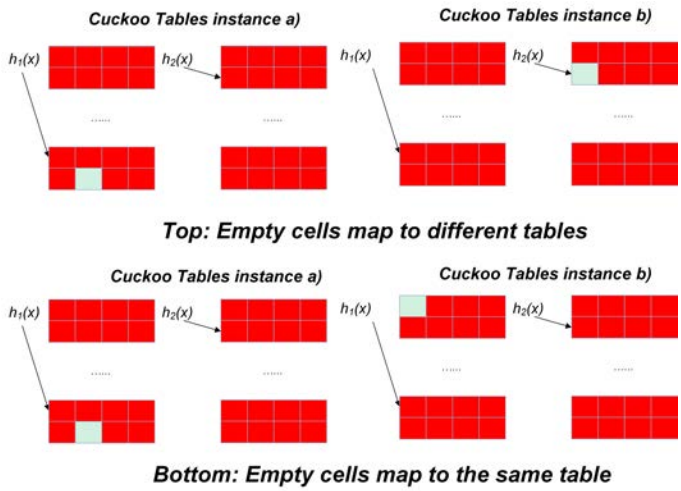

Fig. 5: Examples of instances a) and b) for $d = 2$, $c = 4$. The empty cells fall on different tables (Top). The empty cells fall on the same table (Bottom)

a single table implementation, if the hypothesis holds, the procedure described will always find a set of elements for the attack if a sufficient large number of elements is tested. This means that the multiple table implementation is the worst case for an attacker and thus is the one that will be considered in the rest of the paper unless otherwise noted.

The proposed attack procedure can be generalized for a d table implementation by using d cuckoo hash instances that are almost full and testing elements for insertion. The general algorithm is shown in Algorithm 1. It can be seen that the basic procedure is the same as in the example discussed in the previous subsection. The main difference is that now, d cuckoo instances are used.

3.4 Cuckoo hash table size inference

If the capacity of the cuckoo hash tables S is unknown, we must first infer the size. By inserting items until one is confident filter is full and the failure probability equals 1, one can estimate the size by the number of successful insertions I. A simple procedure inserts until the number of consecutive failed insertions sufficiently outnumber the number of successful insertions: $F > c \cdot I$ for some constant $c \gg 1$.

One can take $c = -\log \rho$ where $\rho \in (0, 1)$ controls the probability that the filter is mistakenly considered full, and I is used as the estimated size of the filter. To see this, note that the probability of a successful insertion is at least J/M when there are J buckets with empty entries. Thus, when the filter is not full, the probability of observing F failed insertions in a row is $< (1 - J/M)^F \approx \exp(-J\,F/M) < \exp(-J\,F/S)$. This gives there is less than a probability ρ chance that $\hat{F}_0 = \frac{S}{J} \log \rho$ consecutive insertion failures are observed given J, S. Approximating the number of non-filled buckets J with the number of empty entries $S - I$ and noting that $I + 1 \geq S/(S - I)$ gives $\hat{F} := I \log \rho \approx (I + 1) \log \rho \geq \frac{S}{S-I} \log \rho \approx \hat{F}_0$. Thus, ρ is an approximate upper bound on the probability that the filter is full when $\hat{F}$ consecutive insertion failures are observed.

3.5 Analysis

The two primary concerns for an attacker running the algorithm are the cost of an attack and the probability it is successful. We first analyze the cost for an attack on a cuckoo hashing instance using d tables with M buckets each and c entries per bucket. We then study the probability that an attack fails for the multiple table implementation due to any of the d cuckoo instances having the last empty cell on the same table. For this case, the probability that all empty cells on all instances map to different tables is computed.

Algorithm 1 Procedure to generate an attack set for an implementation with d tables

```
construct d instances of the cuckoo and fill them except for one cell.
construct a test instance of the cuckoo that is empty.
success ← false
AttackSet ← {}
while success == false and not time out do
    generate a test element t
    isCandidate ← true
    for i = 1 to d do
        if insert(t) on cuckoo instance i == true then
            remove(t) on cuckoo instance i
        else
            isCandidate ← false
            break for
        end if
    end for
    if isCandidate then
        AttackSet ← AttackSet ∪{t}
        if insert(t) on cuckoo test instance == false then
            set success = true
        end if
    end if
end while
return AttackSet
```

There are two main parts in the algorithm, the first one is to build the d cuckoo hash instances that are almost full

and the second is to test elements by inserting them on those instances. The complexity of the construction of the cuckoo hash instances is mostly due to the number of elements that have to be inserted to fill all the cells except one. This can be estimated by the number of elements needed to fill a single hash table with M buckets of c cells. This is a coupon collector problem for which there is a widely used approximation [19]:

$$I_c = M \cdot (\log(M) + (c-1) \cdot log(log(M)) + O(M). \quad (4)$$

that can be simplified when the number of cells per bucket is one ($c = 1$) to:

$$I_1 = M \cdot (\log(M) + 0.577) \quad (5)$$

These estimates are lower bounds on the expected number of elements needed to completely fill the cuckoo hash, but Table 1 shows the estimates are accurate when the tables are large. Corresponding upper bounds can be obtained by treating the d tables as a single hash table with dM entries and replacing M with $d \cdot M$ in the equations above. Furthermore, theorem 1 shows that in the special case where $d = 2, c = 1$, the number of insertions needed can be computed and is approximately $M \cdot \log(2M) \approx M \cdot (\log(M) + 0.693)$ which is extremely close to the lower bound given above.

The complexity of the second part of the algorithm can be estimated by the number of elements that need to be tested to build the set of elements used in the attack. On average, the number of elements would be:

$$T = (M)^d \cdot (c \cdot d + 1) \quad (6)$$

From which it can be seen that the effort needed by the attacker grows exponentially with the number of tables d as discussed in [15].

For the multiple table implementation, an attack fails when the last empty cell of different cuckoo instances fall on the same table. The probability that each of the d cuckoo instances have the last empty cell on different tables and thus that the attack can be successful can be estimated by:

$$P_s = \frac{d!}{d^d} = \frac{(d-1)!}{d^{d-1}} \quad (7)$$

as the first cuckoo instance can have its empty cell on any of the d tables, the second on the remaining $d-1$, and so on. This again suggests that configurations with larger number of tables would be harder to attack.

4 Evaluation

To validate the proposed attack, it has been implemented in C++ for two of the most widely used cuckoo hash configurations: two tables with buckets of four cells (d = 2, c = 4) and four tables with single cell buckets (d = 4, c = 1). These two configurations achieve close to full memory utilization with the minimum number of tables and cells per bucket respectively [4].

In a first experiment, the cuckoo hash tables were filled with randomly generated elements until all cells except one were occupied. The number of elements on which insertion was tried to reach that point was recorded. The number of elements that mapped to the empty cell were also logged. The maximum number of cuckoo movements per insertion was set to 100. The simulations were run on a computer with an Intel Core I7 running Windows 10.

The results over 1000 runs are summarized in Tables 1, 2. Looking at the number of insertions needed to almost fill the tables, it can be seen that they match reasonably well the theoretical estimates discussed in the previous section when the number of cells S is large. For small values of S, the deviations are larger and the theoretical estimate becomes even lower than S in some cases.

The results for the percentage of runs in which less than c elements mapped to the empty cell is also consistent with the reasoning of the previous section showing that this occurs with high probability when M is large. It can be observed that the percentage increases with the table size which is again expected as the number of insertions grows more than linearly with table size as discussed before. This first experiment validates the hypothesis for configurations other than d = 2, c = 1, and shows the feasibility of constraining the location of an element when inserting it on the cuckoo hash tables with no knowledge of the implementation details.

The second experiment simulates the full attack. The same table and bucket configurations were used. The percentage of runs for which the algorithm produced a small set of elements that caused an insertion failure was logged. A run was considered successful when a set of 15 or less elements that caused and insertion failure was obtained. For the successful runs, the average number of elements tested to build the set was also logged, again randomly generated elements where used in the simulations. The results over 1000 runs are summarized in Tables 3, 4 for small tables values. For larger tables, it was not practical to run 1000 iterations and the attack was only tested until it was successful. The time for the run that was able to build the colliding elements is reported in Table 5 to give an indication of the practical complexity of the attack.

From the tables, it can be seen that the number of successful runs is close to the values predicted by the theoretical estimates presented in the previous section except when the tables are very small. This can be explained as for small tables, the percentage of cases on which there are no elements on other buckets that map to the bucket of the empty cell is lower as shown in Table 2. In those cases, the proposed procedure cannot ensure that the elements that are successfully inserted will be placed on a single bucket and thus an attacking set may not be found or it may be large.

As for the number of elements tested to build the set, the average over all the runs matches well the theoretical estimates of equation 6. This means that the time values can also be extrapolated to larger table sizes using Equation 6. For example, in switching ASICs, it is common to use $d = 4$ and tables of a few thousand entries [6], [21]. For tables of size 2048, extrapolating from Table 5 predicts that roughly one year will be needed to build the attack set which does not seem practical. However it is important to note that several copies of the first instance in Algorithm 1 can be run in parallel testing insertions. Most of the insertions (on average $\frac{M-1}{M}$) would not be successful. Therefore, all the copies can send their results to a single copy of the rest of the instances in Algorithm 1. This would enable an acceleration

TABLE 1: Average number of insertions to fill the cuckoo tables except for one cell

Number of cells (S)	$d = 4, c = 1$ Simulated	$d = 4, c = 1$ Equation (5)	$d = 2, c = 4$ Simulated	$d = 2, c = 4$ Equation (4)
128	147	129	139	93
512	753	695	639	540
2048	3698	3489	3056	2735
8192	17531	16797	13926	13045
32768	81581	78544	63153	60100
131072	371641	359603	278370	270691
524288	1674598	1620116	1224575	1199871

TABLE 2: Percentage of runs on which only $c - 1$ elements map to the empty cell

Number of cells (S)	$d = 4, c = 1$	$d = 2, c = 4$
128	81.6%	64.1%
512	96.2%	90.6%
2048	98.3%	96.9%
8192	99.1%	98.3%
32768	99.7%	99.2%
131072	99.3%	99.2%
524288	99.7%	99.6%

by a factor of up to M by using M copies of the first instance running on M different cores or processors. Coming back to the previous example, using a few hundred cores would reduce the time from one year to one day or less. Therefore, designers should not rely solely on the computational complexity of the attack to protect their implementations.

In summary, the experiments demonstrate the feasibility of the attack and the theoretical analysis. The proposed procedure may require a very large amount of computations when the number of tables d is large. Therefore, in those configurations, it may not be practical.

5 Discussion

In previous sections, it has been shown that an insertion failure on cuckoo hashing can be created with a small set of elements even when the attacker has no knowledge of the implementation details. The analysis and evaluation also show that the attack may not be computationally practical when the number of tables d is large. However, even in that scenario, the attacker may be able to degrade system performance. For example, the attacker may generate a single cuckoo instance filled except for one cell and test insertion of elements on it. Then a set of elements that pass insertion would collide on one of the tables. This means that if the cuckoo is filled with those elements, that table would only have elements on one bucket while the rest are empty. Therefore, the cuckoo hash would behave effectively as one with $d - 1$ tables. This reduces the number of elements that can be placed by a factor of $\frac{d-1}{d}$ due to the unused table but there is also an additional reduction as the remaining tables can achieve a lower occupancy as there are less choices to place an element [4]. More generally, an attacker can generate sets that collide on an arbitrary number of tables and thus trade the computational complexity for the disruption in capacity introduced in the cuckoo hash. The detailed study of these partial attacks is left for future work.

An interesting question that stems from exposing the vulnerability of cuckoo hashing is what can a designer do to mitigate or avoid the attack. The first thing that can be done is to use randomized or "salted" hash functions so that each instance of the cuckoo uses different hash functions [22]. This would force the attacker to use the cuckoo hash instance that will be attacked for all the tests in the attack procedure, making it unpractical in many cases as the large number of insertions and deletions can easily be detected as an anomaly. For configurations on which the proposed procedure can be run on the instance under attack, a possible mitigation strategy is to return an insertion failure when the occupancy of the cuckoo tables reaches a given threshold that is below full occupancy regardless of whether the new element can be placed on the cuckoo tables. This would reduce the ability of the attacker to create collisions as now the tables cannot be filled beyond the threshold. This however comes at the cost of a reduction on the cuckoo hash capacity. The study of this potential strategy is left for future work. Finally, if an insertion failure does occur, the hash tables could be rebuilt using different hash functions. This however has a significant cost and may not be an option in systems that have to be operational at all times or that do not have memory or hash functions available to perform the rehashing. In any case, it seems that the designer should take security as one of the design considerations for cuckoo hashing, specially if it is used in systems that are critical for the operation of services or networks.

6 Conclusion and Future Work

This paper has shown that cuckoo hash is vulnerable to an attacker that has no knowledge of its implementation details. In particular, it has been shown that an attacker can create a small set of elements that when inserted on the cuckoo hash would create an insertion failure. For some systems, this would lead to a reconfiguration of the cuckoo using different hash functions or enlarging the memory allocated. This would disrupt the system operation for a non negligible amount of time. Instead, for hardware based implementations it may not be possible to allocate more memory or to change the hash functions and thus the insertion failure could lead to a functional failure of the system. This is worrying as cuckoo hash is widely adopted to implement key value store in many computing and networking applications. The analysis and evaluation presented shows that implementations that use more tables

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3058336, IEEE Transactions on Dependable and Secure Computing

TABLE 3: Percentage of runs that built an attack set

Number of cells (S)	$d=4, c=1$ Simulated	$d=4, c=1$ Equation (7)	$d=2, c=4$ Simulated	$d=2, c=4$ Equation (7)
128	12.7%	9.4%	22.4%	50.0%
512	-	9.4%	45.0%	50.0%
2048	-	9.4%	49.5%	50.0%
8192	-	9.4%	48.4%	50.0%

TABLE 4: Average number elements tested to build the attack set

Number of cells (S)	$d=4, c=1$ Simulated	$d=4, c=1$ Equation (6)	$d=2, c=4$ Simulated	$d=2, c=4$ Equation (6)
128	3011758	5242880	2174	2304
512	-	$1.34 \cdot 10^9$	37002	36864
2048	-	$3.43 \cdot 10^{11}$	598691	589824
8192	-	$8.80 \cdot 10^{13}$	8648690	9437184

TABLE 5: Run time in seconds of a successful attack

Number of cells (S)	$d=4, c=1$	$d=2, c=4$
128	646	0
512	124066	2
2048	-	38
8192	-	562
32768	-	8671
131072	-	156994
524288	-	-

in the cuckoo or larger tables are harder to attack but even in those cases, partial attacks can be performed. Therefore, designers that use cuckoo hash in their systems should be aware of this vulnerability and take actions to mitigate it if needed. Among those, the use of randomized hash functions that are different for each instance seems a simple solution.

The analysis and evaluation of partial attacks for cuckoo hash implementations that use a large number of tables seems an interesting topic to extend the ideas presented in this paper. Another area for future work is the theoretical analysis of the probability that there are no elements on other buckets that map to the bucket where the empty cell is for configurations with $d > 2$ or $c > 1$. More generally, developing efficient mitigation techniques to address the cuckoo hash vulnerabilities has a practical interest to make implementations robust against potential attacks.

REFERENCES

[1] S. Li et al., "Achieving One Billion Key-Value Requests per Second on a Single Server," in IEEE Micro, vol. 36, no. 3, pp. 94-104, May-June 2016.

[2] W. Liang, W. Yin, P. Kang and L. Wang, "Memory efficient and high performance key-value store on FPGA using Cuckoo hashing," in Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL), 2016, pp. 1-4.

[3] R. Pagh and F. Rodler, "Cuckoo hashing," Journal of Algorithms, pp. 122-144, 2004.

[4] U. Erlingsson, M. Manasse and F. Mcsherry, "A cool and practical alternative to traditional hash tables," in Proceedings of the 7th Workshop on Distributed Data and Structures (WDAS), 2006.

[5] N. Le Scouarnec "Cuckoo++ hash tables: high-performance hash tables for networking applications" in Proc. of the Symposium on Architectures for Networking and Communications Systems (ANCS). July 2018, pp. 41–54.

[6] G. Levy, S. Pontarelli and P. Reviriego, "Flexible Packet Matching with Single Double Cuckoo Hash," in IEEE Communications Magazine, vol. 55, no. 6, pp. 212-217, June 2017.

[7] D. Zhou, B. Fan, H. Lim, M. Kaminsky and D.G. Andersen, "High performance Ethernet forwarding with Cuckooswitch", Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT '13, ACM, pp. 97–108.

[8] S. Pontarelli and P. Reviriego, "Cuckoo Cache: A Technique to Improve Flow Monitoring Throughput," in IEEE Internet Computing, vol. 20, no. 4, pp. 46-53, July-Aug. 2016.

[9] X. Chen, J. Li, X. Huang, J. Ma and W. Lou, "New Publicly Verifiable Databases with Efficient Updates," in IEEE Transactions on Dependable and Secure Computing, vol. 12, no. 5, pp. 546-556, 1 Sept.-Oct. 2015.

[10] S. A. Crosby and D. S. Wallach, "Denial of Service via Algorithmic Complexity Attacks," in Proc. USENIX Security Symposium, 2003.

[11] D. Clayton, C. Patton, and T. Shrimpton, "Probabilistic Data Structures in Adversarial Environments," in Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS '19), pp. 1317-1334, 2019.

[12] M. Naor and Y. Eylon, "Bloom Filters in Adversarial Environments", ACM Transactions on Algorithms, vol. 15, no. 3, June 2019,

[13] T. Gerbet, A. Kumar and C. Lauradoux, "The Power of Evil Choices in Bloom Filters," in Proc. of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 101-112, 2015.

[14] P. Reviriego and D. Ting, "Security of HyperLogLog (HLL) Cardinality Estimation: Vulnerabilities and Protection," in IEEE Communications Letters, vol. 24, no. 5, pp. 976-980, May 2020.

[15] U. Ben-Porat , A. Bremler-Barr, H. Levy and B. Plattner, "On the Vulnerability of Hardware Hash Tables to Sophisticated Attacks," in Proc. of IFIP NETWORKING, 2012.

[16] R. J. Tobin and D. Malone, "Hash pile ups: Using collisions to identify unknown hash functions," in Proc. International Conference on Risks and Security of Internet and Systems (CRiSIS), 2012.

[17] P. Reviriego and D. Larrabeiti, "Denial of Service Attack on Cuckoo Filter based Networking Systems", in IEEE Communications Letters, vol. 24, no. 7, pp. 1428-1432, July 2020.

[18] L. Devroye and P. Morin "Cuckoo hashing: further analysis," in Information Processing Letters, vol. 86, no. 4, pp. 215–219, 2003.

[19] M. Ferrante and M. Saltalamacchia, "The coupon collector's problem," Mater. Math., vol. 2014, no. 2, pp. 1–35, 2014.

[20] P. Erdös and A. Rényi, "On random graphs I," Publ. Mathematicae, Debrecen 6, 290–297.

[21] P. Bosshart et al, "Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN," in Proceedings of the ACM SIGCOMM 2013.

[22] D. Desfontaines, A. Lochbihler, D.A. Basin, "Cardinality Estimators do not Preserve Privacy," in Proceedings on Privacy Enhancing Technologies, no. 2, May 2019.