ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cuckoo filters in adversarial settings

Semester project

Keran Kocher

February 17, 2023

Advisors: Prof. Dr. Kenneth Paterson, Anupama Unnikrishnan

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

The Cuckoo filter is a probabilistic data structure for approximate membership queries. The structure is a space-efficient data store that supports the insertion of elements, deletions, and membership queries, where a small fraction of false positives is possible. They are deployed in situations where the filter could be manipulated to gain benefit, e.g., bypass a defense. Hence, we study the security of Cuckoo filters in adversarial environments. We define the syntax of the Cuckoo filter with deletions and extend it by applying a keyed pseudorandom function to the input. Then, we derive a correctness bound using a simulation-based definition.

Contents

Chapter 1

Introduction

Probabilistic data structures (PDS) are structures that involve randomized algorithms in order to trade off different properties such as memory cost, speed and accuracy. They are becoming increasingly popular with the rise of data collection and the scaling of many services globally. Indeed, computing statistics on huge datasets or streams of data becomes infeasible because of the memory cost. Therefore, probabilistic data structures are designed for tasks such as estimating the number of unique elements or determining whether an element belongs to a set, while requiring a memory utilization that is sublinear. However, these structures give approximate answers. For instance, the number of distinct elements in a set is not exact but can be given along with a standard error. Examples of popular structures are the Bloom filter, HyperLogLog or Count–min sketch. PDS are deployed in a wide range of applications, such as in distributed systems, search engines or data mining. More importantly, their usage is also frequent in critical domains, like detection of DoS attacks in networks.
Probabilistic data structures can be deployed in open environments where they might encounter adversarial inputs, i.e an attacker tries to influence the behavior of the structure in his favour. The data structures are usually designed for typical cases in an honest user setting. Their correctness and performance are also measured in that case. However, they are already widely deployed in adversarial environments. For example, the Bloom filter is a structure for membership queries in a set and is regularly used in network security. The detection of DoS attacks by filtering the traffic or the revocation of certificates (CRLite [12]) are notorious applications. A proposal [18] to mitigate DNS amplification attacks also involves Bloom filters. A DNS amplification attack exploits open recursive servers and IP source spoofing to flood a victim with DNS responses he did not request. The defense consists of recording outgoing requests to verify the one-to-one mapping with the incoming requests. Since the memory footprint can be huge, the requests

are stored in Bloom filters. If the corresponding outgoing request is found in the filter, the response is considered legitimate. However, a malicious request that is a false positive in the set is also considered legitimate. The authors assume the expected false positive rate that is computed in the honest setting, but a careful adversary could craft his DNS requests to increase that rate and thus avoid the filtering.
Hence, the current situation justifies the need for a security analysis of probabilistic data structures and the deployment of security measures when necessary. In the past years, researchers have started to study this problematic and our work aims to contribute to this effort by considering Cuckoo filters.

1.1 Cuckoo filter

The Cuckoo filter [6] is a probabilistic data structure for approximate membership query (AMQ). The filter lets a user insert and delete elements in a set, and then query it to determine whether an element is in the set (i.e. the element was previously inserted). Only a fingerprint of the element, obtained by hashing, is stored. Therefore, the memory cost of the PDS is sublinear at the cost of giving approximate answers. That is, if the query is negative, the element is truly not in the set (no false negative) whereas if the query is positive, the element can be a false positive with some probability. The filter trades off the required storage with the false positive rate. This technique is particularly useful when handling very large data collection. The filter is based on Cuckoo hashing [15], which is a hashing scheme that uses two hash functions instead of one, applied for instance in hash tables to allocate the keys. Cuckoo filters are a recent addition to the family of AMQ-PDS and are comparable to the popular Bloom filter, except that Cuckoo filters allow the deletion of elements. Bloom filter extensions (e.g. counting Bloom filter) exist to permit deletion, however Cuckoo filters perform better in most situations.

1.2 Applications

The Cuckoo filter was introduced in 2014 while the Bloom filter was conceived in 1970. Hence, the Bloom filter dominates when it comes to applications in the industry. Bloom filters are widely used in network security for instance [8]. Network filtering can be done using a counting Bloom filter that records the origin IP of incoming packets to prevent denial of service attacks. The filter is also suited in a distributed setting. Distributed databases can share their partition of data efficiently in the case of distributed queries, i.e they send a Bloom filter instead of the data. In general, an AMQ-PDS can be used to save bandwidth (e.g. in a disk or a network) by verifying the presence of an element before fetching it.

On the other hand, Cuckoo filters are rarely seen in practice. Although many implementations exist, the Redis database is the only application to our knowledge offering support for Cuckoo filters. It seems that even if deletion is needed, counting Bloom filters are still preferred nowadays, although Cuckoo filters perform better in many settings [6]. Still, there are many propositions to deploy Cuckoo filters. SPACF is a secure privacy-preserving authentication scheme for vehicular ad-hoc network which uses a combination of Cuckoo filters to verify a batch of signatures [4]. Similarly, TinyCR [17] is a mechanism to verify the validity of certificates in IoT devices, where performance and memory are decisive. In network security, many researchers propose to replace Bloom filters with Cuckoo for improved performance, in deep packet inspection [1] or IP lookup table [11] for example. Likewise, distributed data systems can benefit from Cuckoo filters improvements [13]. In privacy, they can be used to reduce the communication and computational costs in complex protocols like Private Set Intersection (PSI). A concrete example can be found in [9] for efficient private contact discovery in mobile. Overall, recent research suggests that Cuckoo filters have a great potential and might be increasingly deployed in practice.

The authors of the Cuckoo filter paper provide a in-depth comparison between Cuckoo (CF) and Bloom filters (BF) to understand what their strengths and weaknesses are. We present three metrics [6]:

- *Space efficiency:* the total size occupied w.r.t the number of items.
- *False positive rate:* the number of positive responses when querying random non-inserted elements.
- *Throughput:* the number of operations per second for insertions, deletions and queries.

We will only state the results of the benchmarks, but the details of the methodology can be found in the original paper [6]. In a Bloom filter, each element requires a constant number $1.44\log_2(1/\epsilon)$ of bits given the false positive rate ϵ. In a Cuckoo filter, the number of bits per element is $(\log_2(1/\epsilon)+3)/\alpha$ and depends on the load factor α of the data structure. We observe that the space efficiency is superior in Cuckoo filters when the load is sufficiently high. In practice, the load factor α of a Cuckoo filter can reach 95% with a suitable configuration. Therefore, CF are more efficient than BF when the false positive rate ϵ is sufficiently low ($\epsilon < 0.3\%$). This behavior comes from the false positive rate, which is independent of the number of insertions in the Cuckoo filter. On the other hand, this rate grows with the number of items in a Bloom filter.
Lookup throughput is constant w.r.t to the state in a Cuckoo filter, since the queried element is compared to a constant number of items. The lookup performance is therefore generally better in CF than in BF, except for low occupancy and negative queries where the BF can return as soon as a zero

bit is encountered. The opposite is observed when comparing the insertion throughput. Bloom filters are constant regardless of the occupancy while Cuckoo filters performance drops as the load factor grows. Typically, the performance is worse than a Bloom filter when the load is above 80%.

1.3 Related work

Previous works have studied the security of probabilistic data structures. Naor and Yogev [14] highlight the weakness of Bloom filters in an adversarial environment and formalize adversarial correctness using a game-based approach. They model an adversary that has a limited number of insertions and membership queries, that cannot see the internal state and that must find a false positive. Clayton *et al.* [2] extend their work and generalize to more data structures, namely Bloom filters, Counting filters and Count-min sketch. They also consider more settings, such as an adversary that can reveal the internal state. They provide security bounds and some countermeasures for each structure. However, their game-based framework requires to choose a winning condition for the adversary.
Paterson and Raynal [16] analyze the security of HyperLogLog, a probabilistic data structure to estimate the cardinality of a data set. They demonstrate attacks on HyperLogLog and introduce a simulation-based framework for a formal security analysis. Filić *et al.* [7] also follow a simulation-based method inspired by that of [16] to prove the adversarial correctness of any AMQ-PDS that respects some given rules. Additionally, they also consider the privacy of the structures. They apply their framework to Bloom and Cuckoo filters. Our work is based on their method, which we will describe as we go.
Regarding Cuckoo filters, Reviriego *et al.* describe an attack on a filter deployed in a networking system. They show that the filter can be adversarially disabled with a number of queries proportional to the filter size and the fingerprint space. Epstein [5] considers a simplified version of the Cuckoo filter to understand the performance guarantees in the honest setting. Notably, he removes the hashing of some term such that he can derive an expression for the probability that an insertion fails. Thus, he can provide further insights on parameters selection. Unfortunately, we do no know if his approach generalizes to the standard Cuckoo filter. Yeo [19] focuses on the use of Cuckoo hashing in cryptography. He proposes a new Cuckoo hashing construction and states the conditions to obtain a robust data structure such that an adversary cannot leak private data from the filter.

1.4 Outline

In this project, we specifically study the security of the Cuckoo filter. We base our work on the research from Filić *et al.* [7]. The authors studied the security of probabilistic data structures for approximate membership query (AMQ). They provide a simulation-based proof framework and prove the security of insertion-only AMQ-PDS such as the Bloom filter and the Cuckoo filter.
Simulation-based proofs consider two worlds. The real world models the probabilistic data structure under analysis in real conditions, as it is implemented. On the other hand, the ideal world simulates the data structure such that the simulation is consistent with the expected behavior but also achieves some security goals. In our case, the ideal world simulates a filter that stays non-adversarially influenced. Then, we can derive a security bound by defining a cryptographic game. In this game, an adversary is distinguishing between the two worlds by playing in either of them, i.e he interacts through an API with the filter's instantiation or the simulator. His advantage decides the security bound. Intuitively, the bound states the closeness between the data structure and its ideal version.
An advantage of simulation-based proof is the absence of a specific winning condition for the adversary, which leads to a broad security definition. In [7], the authors define the notion of adversarial correctness and derive a security bound for any insertion-only AMQ-PDS. Our goal is to extend the simulation-based proof to Cuckoo filters with deletions allowed.
We study the differences introduced by adding deletions in Cuckoo filters. We first define the syntax and the algorithmic behavior of a Cuckoo filter. In particular, we add the capability to delete elements and update the other functions accordingly. Based on that, we proceed with the security analysis and adapt the framework of [7]. Notably, we need to describe the adversarial model and the game setting. Besides, we update the ideal world's simulator to support deletion. Finally, we state a theorem for the adversarial correctness and demonstrate a bound for it.

1.5 Preliminaries

Notation. We use the same notation as in [7]. Given $m \in \mathbb{Z}_{\geq 1}$, $[m]$ is the set $\{1, 2, ..., m\}$. For a set S, $\mathcal{P}(S)$ is the power set of S, and $\mathcal{P}_{\text{lists}}(S)$ is the set of all lists with non-repeated elements from S. Given two sets $\mathfrak{D}$ and $\mathfrak{R}$, $\mathsf{Func}[\mathfrak{D}, \mathfrak{R}]$ is the set of functions from $\mathfrak{D}$ to $\mathfrak{R}$. When $F \leftarrow\!\!\$\ \mathsf{Func}[\mathfrak{D}, \mathfrak{R}]$ is sampled, F is a random function $\mathfrak{D} \xrightarrow{F} \mathfrak{R}$. The identity function $\mathsf{Id} : S \to S$ over a set S is written Id_S. If D is a distribution, $x \leftarrow\!\!\$\ D$ is a sample drawn from D. The number of elements in the set S is written $|S|$. In pseudo code, $a \leftarrow \perp^s$ is a list of s empty slots, $\mathsf{load}(a)$ is the number of entries in a and $a[i]$

is the i^{th} entry in a. Dictionaries (key-value stores) are initialized with empty value $\perp$ at each key, and $\perp < n, \forall n \in \mathbb{R}$. Variable assignment is denoted with an normal arrow $\leftarrow$ or a dollar arrow $\leftarrow\$$ if the input is randomized. For any randomized algorithm, an extra argument $r \in \mathcal{R}$ can be added to represent the coins, i.e the randomness, over the set of coins $\mathcal{R}$. If not specified, we assume the coins are sampled uniformly. The argument may be omitted when it is notationally convenient. We write $\mathsf{alg}^{\mathsf{f}_1,\ldots,\mathsf{f}_n}$ to denote an algorithm that can access oracles $\mathsf{f}_1, \ldots, \mathsf{f}_n$.

PRF experiment. Figure 1.1 defines the pseudorandom function experiment which formally defines the security of such functions. In brief, we consider the advantage of an adversary that must distinguish between a PRF and a perfectly random function.

$\mathrm{Exp}_R^{PRF}(\mathcal{B})$	Oracle $\mathsf{RoR}(x)$
1 : $K \leftarrow\$ \mathcal{K}; F \leftarrow\$ \mathsf{Func}[\mathfrak{D}, \mathfrak{R}]$	1 : **if** $b = 0 : y \leftarrow R_K(x)$
2 : $b \leftarrow\$ \{0,1\}; b' \leftarrow\$ \mathcal{B}^{\mathsf{RoR}}$	2 : **else** $: y \leftarrow F(x)$
3 : **return** b'	3 : **return** y

Figure 1.1: PRF experiment

Definition 1.1 (PRF experiment) *Consider the PRF experiment in Figure 1.1. We say a pseudorandom function family $R : \mathcal{K} \times \mathfrak{D} \to \mathfrak{R}$ is (q, t, ϵ)-secure if for all adversaries $\mathcal{B}$ running in time at most t and making at most q queries to RoR oracle in $\mathrm{Exp}_R^{PRF}(\mathcal{B})$ we have*

$$\mathsf{Adv}_R^{PRF}(\mathcal{B}) := |\Pr[b' = 1 \mid b = 0] - \Pr[b' = 1 \mid b = 1]| \leq \epsilon.$$

We say $\mathcal{B}$ is a (q, t)-PRF adversary.

Chapter 2

Cuckoo filters

2.1 Syntax and algorithms

We define the syntax for the Cuckoo filter with deletions and describe its exact behavior. The syntax is based on [7]. A Cuckoo filter is characterized by a set of public parameters $pp = (\lambda_I, \lambda_T, b, u)$, where

- λ_I is the bit length of the bucket's index. The hash function to compute indices is $H_I : \mathfrak{D} \to \{0,1\}^{\lambda_I}$ and there are 2^{λ_I} buckets in the filter.
- λ_T is the bit length of the fingerprint. The hash function to compute fingerprints is $H_T : \mathfrak{D} \to \{0,1\}^{\lambda_T}$.
- b is the size of a bucket (number of slots).
- u is the maximum number of evictions.

A Cuckoo filter consists of four algorithms. The complete description can be found in Figure 2.1. We refer to these definitions when the functions are mentioned in the following chapters.

- The setup algorithm $\sigma \leftarrow \mathsf{setup}(pp)$ initializes an empty state σ given the public parameters pp.
- The insertion algorithm $(a, \sigma') \leftarrow\!\!\$\ \mathsf{insert}(x, \sigma)$ inserts an element $x \in \mathfrak{D}$ in the state σ. It returns the bit a to indicate either a successful insertion ($a = \top$) or a failure ($a = \bot$). The insertion also returns the updated state σ'.
- The membership querying algorithm $b \leftarrow \mathsf{query}(x, \sigma)$ returns the bit b to indicate whether the element $x \in \mathfrak{D}$ is present ($b = \top$) or absent ($b = \bot$) in the state σ.
- The deletion algorithm $(c, \sigma') \leftarrow \mathsf{delete}(x, \sigma)$ deletes an element $x \in \mathfrak{D}$ from the state σ. It returns the bit c to indicate whether the element

$\mathsf{setup}(pp)$	$\mathsf{query}^{F,H_T,H_I}(x,\sigma)$
1 : $\lambda_I, \lambda_T, b, u \leftarrow pp$	1 : $x \leftarrow F(x)$
2 : // Initialize 2^{λ_I} buckets, b λ_T-bits slots	2 : $tag \leftarrow H_T(x)$
3 : **for** $i \in 2^{\lambda_I} : \sigma_i \leftarrow \perp^b$	3 : $i_1 \leftarrow H_I(x)$
4 : $\sigma_{\mathsf{evic}} \leftarrow \perp$	4 : $i_2 \leftarrow i_1 \oplus H_I(tag)$
5 : $\omega_{\mathsf{evic}} \leftarrow \perp$	5 : $a \leftarrow tag \in \sigma_{i_1}$ or $tag \in \sigma_{i_2}$
6 : **return** $\sigma \leftarrow (\sigma_i)_i, \sigma_{\mathsf{evic}}$	6 : $b \leftarrow tag = \sigma_{\mathsf{evic}}$ and $(i_1 = \omega_{\mathsf{evic}}$ or $i_2 = \omega_{\mathsf{evic}})$
	7 : **return** a or b

Figure 2.1: Syntax for setup and query

was successfully deleted ($c = \top$) or not ($c = \perp$). The deletion also returns the updated state σ'.

Overview. We give a brief overview on how the filter works. The state σ consists of 2^{λ_I} buckets and every bucket has b slots to store an element. On the insertion of element $x \in \mathfrak{D}$, a fingerprint is computed using the hash function H_T: $tag = H_T(x)$. The state will only contain the fingerprint, not the element itself. Then, two indices are computed using the H_I hash function: $i_1 = H_I(x)$ and $i_2 = i_1 \oplus H_I(tag)$. The key property of the indices is that, given an index, the second index can be computed only using the fingerprint of x and H_I. Finally, the fingerprint is inserted in one of the buckets given by i_1 and i_2. If both buckets are full, an element y (which is a stored fingerprint) is chosen at random in either buckets and moved to its secondary bucket, in which case the secondary index $i' = i \oplus H_I(y)$ is computed and y is inserted in the bucket given by i'. This procedure is called an eviction (of y). Note that evictions will be done multiple times consecutively if the secondary buckets are also full. Also, if the same element is inserted multiple times, the fingerprint must be inserted every time to maintain consistency with deletions. On the querying of element $x \in \mathfrak{R}$, the fingerprint and the two indices are computed. Then, the fingerprint of x is matched against the fingerprints stored in the two buckets. The insertion algorithm runs in constant time in expectation, while the query algorithm is always constant.

Disabling. We now explain how the filter reaches its capacity and insertions can fail. When the maximum number of consecutive evictions for a single insertion is reached (defined as the parameter u), the last evicted item is stored in a stash σ_{evic} and insertion is disabled as long as the stash is full. The insertion leading to that event is still successful, because no element are discarded. Consequently, the query and deletion algorithms must also check the element in the stash.

$\mathsf{insert}^{F,H_T,H_I}(x,\sigma)$

1 : $x \leftarrow F(x)$
2 : **if** $\sigma_{\mathsf{evic}} \neq \perp$: **return** $\perp$, σ // filter is full
3 : $tag \leftarrow H_T(x)$
4 : $i_1 \leftarrow H_I(x)$
5 : $i_2 \leftarrow i_1 \oplus H_I(tag)$
6 : **for** $i \in \{i_1, i_2\}$
7 : **if** $\mathsf{load}(\sigma_i) < b$
8 : $\sigma_i \leftarrow \sigma_i \diamond tag$
9 : **return** $\top, \sigma$
10 : // no empty slot available
11 : $i \leftarrow\!\!\$\ i_1, i_2$
12 : **for** $g \in [num]$
13 : $slot \leftarrow\!\!\$\ [b]$
14 : $evic \leftarrow \sigma_i[slot]$
15 : $\sigma_i[slot] \leftarrow tag$
16 : $tag \leftarrow evic$
17 : $i \leftarrow i \oplus H_I(tag)$
18 : **if** $\mathsf{load}(\sigma_i) < b$
19 : $\sigma_i \leftarrow \sigma_i \diamond tag$
20 : **return** $\top, \sigma$
21 : // max number of evictions reached, filter is full
22 : $\sigma_{\mathsf{evic}} \leftarrow tag$
23 : $\omega_{\mathsf{evic}} \leftarrow i$
24 : **return** $\top, \sigma$

$\mathsf{delete}^{F,H_T,H_I}(x,\sigma)$

1 : $x \leftarrow F(x)$
2 : $tag \leftarrow H_T(x)$
3 : $i_1 \leftarrow H_I(x)$
4 : $i_2 \leftarrow i_1 \oplus H_I(tag)$
5 : **if** $tag = \sigma_{\mathsf{evic}}$ and $(i_1 = \omega_{\mathsf{evic}}$ or $i_2 = \omega_{\mathsf{evic}})$
6 : $\sigma_{\mathsf{evic}} \leftarrow \perp$; $\omega_{\mathsf{evic}} \leftarrow \perp$
7 : **return** $\top, \sigma$ // The filter is open
8 : **if** $tag \in \sigma_{i_1}$
9 : $\sigma_{i_1} \leftarrow \sigma_{i_1} \setminus tag$
10 : **elseif** $tag \in \sigma_{i_2}$
11 : $\sigma_{i_2} \leftarrow \sigma_{i_2} \setminus tag$
12 : **else** : **return** $\perp, \sigma$, // nothing to delete
13 : **if** $\sigma_{\mathsf{evic}} \neq \perp$
14 : $(tag, i_1) \leftarrow (\sigma_{\mathsf{evic}}, \omega_{\mathsf{evic}})$
15 : $\sigma_{\mathsf{evic}} \leftarrow \perp$; $\omega_{\mathsf{evic}} \leftarrow \perp$
16 : // reinsert the element from the stash
17 : $i_2 \leftarrow i_1 \oplus H_I(tag)$
18 : **for** $i \in \{i_1, i_2\}$
19 : **if** $\mathsf{load}(\sigma_i) < b$
20 : $\sigma_i \leftarrow \sigma_i \diamond tag$
21 : **return** $\top, \sigma$
22 : $i \leftarrow\!\!\$\ i_1, i_2$
23 : **for** $g \in [num]$
24 : $slot \leftarrow\!\!\$\ [b]$
25 : $evic \leftarrow \sigma_i[slot]$
26 : $\sigma_i[slot] \leftarrow tag$
27 : $tag \leftarrow evic$
28 : $i \leftarrow i \oplus H_I(tag)$
29 : **if** $\mathsf{load}(\sigma_i) < s$
30 : $\sigma_i \leftarrow \sigma_i \diamond tag$
31 : **return** $\top, \sigma$
32 : $\sigma_{\mathsf{evic}} \leftarrow tag$; $\omega_{\mathsf{evic}} \leftarrow i$
33 : **return** $\top, \sigma$

Figure 2.1: Syntax for insert and delete

Deletion. We extend the syntax of [7] by adding the *delete* algorithm. Deletion is as described in the original paper [6], except that that the eviction stash σ_{evic} is now supported. On the deletion of element $x \in \mathfrak{D}$, the fingerprint and the two indices are computed. Similar to querying, the fingerprint of x is compared to the fingerprints in the buckets and the stash, and one fingerprint is removed if it matches. If many fingerprints are matching, it is sufficient to remove any one of them. For instance, the same fingerprint could reside in the two buckets and correspond to two different elements. But we can safely delete any of the two since the secondary bucket depends solely on the fingerprint (the primary bucket of one is the secondary bucket of the other regardless of the original element).

Non-permanent disabling. Additionally, we must also determine whether the filter can reopen once elements are deleted. Following the official implementation [3], we decide that the filter can accept insertions again after an element is deleted. Thus, the filter is not permanently disabled. This choice also makes sense for the usability of the filter. In production, the filter could reach capacity because of high utilization (e.g. peak network traffic) and then recover when the situation becomes stable again. Therefore, on deletion, the filter opens again if the removed element was in the stash. Additionally, if an element in a bucket is successfully deleted, the filter takes the element in the stash (if any) and inserts it again. If this insertion is successful, the stash stays empty. Otherwise, the stash is full again and the filter remains closed. Note that the state of the filter is made of two variables: σ represents the tags in the filter and ω stores indices. In particular, it is necessary to store the index of an evicted element such that it can be reinserted afterwards.

To analyze the state of the filter in the adversarial setting, we must first know what to expect in the honest setting. We can then identify the deviation between the two settings. The input domain $\mathfrak{D}$ and its distribution is not known in general and is application-dependent. Thus, we cannot derive an expected state in that case. For this reason, [7] introduces the F-decomposability property and the notion of non-adversarially influenced (NAI) to describe the non-adversarial setting.

2.2 The non-adversarial setting

Definition 2.1 (Function-decomposability) *Let* Π *be an AMQ-PDS and* $F \leftarrow\!\!\$ \mathsf{Func}[\mathfrak{D}, \mathfrak{R}]$ *with* $\mathfrak{R} \subset \mathfrak{D}$ *be a random function. Let* $\mathsf{Id}_{\mathfrak{R}}$ *be the identity function*

n-NAI-gen$^F(pp)$

1 : $\sigma^{(0)} \leftarrow\$ \mathsf{setup}(pp)$
2 : $[x_1, ..., x_n] \leftarrow\$ \{S \in \mathcal{P}_{\text{lists}}(\mathfrak{D}) \mid |S| = n\}$
3 : **for** $j = 1, ..., n : (b, \sigma^{(j)}) \leftarrow\$ \mathsf{insert}^F(x_j, \sigma^{(j-1)})$
4 : **return** $\sigma^{(n)}$

Figure 2.2: Algorithm returning non-adversarially influenced (NAI) state

over $\mathfrak{R}$. *We say that* Π *is F-decomposable if we can write* $\forall x \in \mathfrak{D}, \sigma \in \Sigma$

$$\begin{aligned}\mathsf{insert}^F(x, \sigma) &= \mathsf{insert}^{\mathsf{Id}_{\mathfrak{R}}}(F(x), \sigma),\\ \mathsf{query}^F(x, \sigma) &= \mathsf{query}^{\mathsf{Id}_{\mathfrak{R}}}(F(x), \sigma),\\ \mathsf{delete}^F(x, \sigma) &= \mathsf{delete}^{\mathsf{Id}_{\mathfrak{R}}}(F(x), \sigma).\end{aligned}$$

The input to the algorithms is first transformed by a random function F whose output is uniformly random independently from the input. Therefore, the distribution of the input $\mathfrak{D}$ can be ignored and the algorithm can expect a uniform input in $\mathfrak{R}$. In practice, a truly random function can be replaced by a PRF with certain security guarantees.

NAI. We define the non-adversarially influenced state for a Cuckoo filter as a state close to one generated by inserting n distinct random elements in the filter. This definition is introduced by [7] to formalize the state in the honest setting.

Definition 2.2 ((n, ϵ)-NAI) *Let* $\epsilon > 0$, *and let* n *be a non-negative integer. Let* Π *be a Cuckoo filter with public parameters* pp *and state space* Σ, *such that its* insert *algorithm makes use of oracle access to function* F. *Let* alg *be a randomised algorithm outputting values in* Σ. *Let* $\sigma, \sigma^{(n)}$ *be random variables representing respectively the output of* alg *and of the randomised algorithm* n-NAI-gen$^F(pp)$ *described in Figure 2.2. We say that* alg *outputs an* (n, ϵ)*-non-adversarially-influenced state (denoted by* (n, ϵ)*-NAI) if* σ *is* ϵ*-statistically close to* $\sigma^{(n)}$.

False positive. In the context of AMQ-PDS, a false positive is a membership query that is positive for an element that was not previously inserted. False positives are undesirable and are a critical consideration in AMQ-PDS. Indeed, these structures trade off their memory footprint with their false positive rate. In Cuckoo filters, on the querying of some x that was not previously inserted, the fingerprint of x can match another fingerprint in the buckets (a hash collision) and the query would be falsely positive. First, we formally define the false positive probability for a NAI state.

Definition 2.3 (NAI false positive probability) *Let Π be a Cuckoo filter with public parameters pp, using function F sampled from a distribution D_F to instantiate its functionality. Let n be a non-negative integer. Define the NAI false positive probability after n distinct insertions as*

$$P_{\Pi,pp}(FP|n) := \begin{bmatrix} F \leftarrow\!\$\ D_F \\ \sigma \leftarrow\!\$\ n\text{-NAI-gen}^F(pp) \\ x \leftarrow\!\$\ \mathfrak{D} \setminus V : \top \leftarrow \mathsf{query}^F(x,\sigma) \end{bmatrix},$$

where V is the list $[x_1, ..., x_n]$ sampled on line 2 of n-NAI-gen$^F(pp)$.

In Cuckoo filters, we might encounter states that are adversarially generated despite the randomness. Because of the structure of the filter, the adversary can potentially target the set of random elements that are successfully inserted. The placement of each element is non-deterministic because of the two possible slots and depends on the current state. Therefore, a user could obtain an adversarially influenced state in comparison to the one of n-NAI-gen$^F(pp)$ with a suitable sequence of insertions and deletions. Thankfully, we can show that the original bound of the false positive probability [6] also holds for adversarially generated states. Hence, we define the universal false positive probability and show that the bound is valid for any state.

Definition 2.4 (Universal false positive probability) *Let Π be a Cuckoo filter with deletions, public parameters pp and state space Σ. Define $\overline{P_{\Pi,pp}}(FP)$ the upper bound of the false probability of Π in any state $\sigma \in \Sigma$.*

Lemma 2.5 *Let Π be a Cuckoo filter with deletions, public parameters $pp = (\lambda_I, \lambda_T, b, u)$ and state space Σ. Define $P_{\Pi,pp}(FP|\sigma)$ the false positive probability of the filter given a state $\sigma \in \Sigma$. Then,*

$$\forall \sigma \in \Sigma,\ P_{\Pi,pp}(FP|\sigma) \leq \overline{P_{\Pi,pp}}(FP) = 1 - (1 - 2^{-\lambda_T})^{2b+1}.$$

Proof By definition, a false positive is observed when the membership query of a non-inserted element is positive. We consider the query of a random element x. If x is random, then its fingerprint $tag = H_T(x)$ is also random. Thus, the probability that *any* element y matches x is $2^{-\lambda_T}$, λ_T the number of bits in a fingerprint. Then, the probability that n distinct elements do not match x is $(1 - 2^{-\lambda_T})^n$. Finally, we want the probability that at least one element matches x, which is $1 - (1 - 2^{-\lambda_T})^n$. This probability increases with n, so the upper bound is reached when n is maximal. In a Cuckoo filter, a queried element is compared to at most $2b + 1$ elements, the two buckets and the stash. Therefore, the false probability in a filter with any state is bounded by $1 - (1 - 2^{-\lambda_T})^{2b+1}$. □

False negative. False negatives cannot happen in insertion-only AMQ-PDS. However, if deletions are possible, then false negatives can arise. In Cuckoo

filters, the deletion of a non-inserted element can create a false negative. Indeed, on the deletion of some x that was not previously inserted, most of the time the state will remain unchanged. However, the fingerprint of x can still match a fingerprint stored in one of its buckets. In this case, an element would be deleted and a membership query on that element would be falsely negative (if the element was only inserted one time).

PRF-wrapped Cuckoo filters. Cuckoo filters are not *function-decomposable* by default. To have this property, the proposal of [7] is to preprocess the input of insert, query and delete by a random function $F : \mathfrak{D} \to \mathfrak{R}$. We wrap the algorithm by first applying F to the input, and then calling the corresponding procedure. The PRF-wrapped Cuckoo filter is described in Figure 2.1. By doing so, we can easily achieve *F-decomposability*. This property comes at the cost of a higher false positive rate (due to the collisions introduced by F). The false positive rate increase was proven to be bounded by $\mathcal{O}(|\mathfrak{R}|^{-1})$ and can be made arbitrarily small by increasing the codomain of F. We state the lemma from [7].

Lemma 2.6 *Let n be a non-negative integer, let Π be a PRF-wrapped insertion-only Cuckoo filter with public parameters $pp = (\lambda_I, \lambda_T, b, u)$, wrapped using a PRF $R_K : \mathfrak{D} \to \mathfrak{R}$, and let Π' be an original Cuckoo filter with deletions and the same parameters pp as Π.*

$$\text{Let } \overline{P_{\Pi,pp}}(FP|n) := \overline{P_{\Pi',pp}}(FP|n) + \frac{(2b+2)^2}{2|\mathfrak{R}|}.$$
$$\text{Then, } P_{\Pi,pp}(FP|n) \leq \overline{P_{\Pi,pp}}(FP|n).$$

Proof Since we defined the NAI false positive probability (Def 2.3) with the inserted elements being distinct, the PRF can break this assumption due to collisions. Therefore, we apply the birthday problem to bound the probability that there is collision in the set of $2b+2$ elements ($2b+1$ elements in the buckets and the stash, and the inserted element). See [7] for a complete proof. □

Disabling probability. The disabling probability is another central metric for an AMQ-PDS, since this expression also defines the expected load factor of the filter. We first define the *first-time disabling probability*, the probability that an insertion fails for the first time given n elements were already successfully inserted.

Definition 2.7 (First-time disabling probability) *Let Π be an insertion-only Cuckoo filter with public parameters pp. Let n be a non-negative integer. Define*

the first-time disabling probability as

$$P_{\Pi,pp}(D|n) := \left[\begin{array}{c} \sigma^{(0)} \leftarrow\$ \mathsf{setup}(pp) \\ [x_1,...,x_n] \leftarrow\$ \{S \in \mathcal{P}_{lists}(\mathfrak{D}) \mid |S| = n\} \\ \textbf{for } j = 1,...,n : (\top, \sigma^{(j)}) \leftarrow\$ \mathsf{insert}^F(x_j, \sigma^{(j-1)}) \\ x \leftarrow\$ \mathfrak{D} \setminus V : \bot \leftarrow \mathsf{insert}^F(x, \sigma^{(n)}) \end{array}\right],$$

where V is the list $[x_1, ..., x_n]$ sampled on line 2.

Unfortunately, no closed-form expression exists for Cuckoo filters up to this day. In our case, the disabling probability also intervenes in the adversarial correctness. Thus, we propose a bound of the probability. The task is not trivial because the structure of the Cuckoo filter is hard to analyze. In the end, the probability to be disabled is the probability that the stash is filled at some point. This event mainly depends on the number of full buckets. We need to determine the distribution of the full buckets, which is the key and challenging step. We must consider all ways the filter can construct its state. Each element has two buckets, which are assigned either at random if both have empty slots, to the second bucket if the first is full, or another element is evicted to another bucket if both are full. Furthermore, the secondary bucket is not uniformly distributed if $\lambda_T < \lambda_I$, because the secondary index depends on the fingerprint which has only 2^{λ_T} values. Consequently, evicted elements are not moved uniformly and some buckets may be filled faster than other. More details can be found in the appendix A. We define a loose upper bound of the first-time disabling probability.

Definition 2.8 (Upper bound on first-time disabling probability) *Let Π be an insertion-only Cuckoo filter with public parameters pp. Define $\overline{P_{\Pi,pp}}(D|n)$ the upper bound of the first-time disabling probability (Def 2.7).*

Lemma 2.9 *Let Π be an insertion-only Cuckoo filter with public parameters $pp = (\lambda_I, \lambda_T, b, u)$. Let $l = \lfloor \frac{n}{b} \rfloor$. Then,*

$$\overline{P_{\Pi,pp}}(D|n) \leq \frac{l}{m}.$$

Proof The disabling probability given l buckets are full is trivially bounded by $\frac{l}{m}$, the probability that a primary index points to a full bucket. This probability increases with l, so the upper bound is reached when l is maximal. Given that n elements were inserted in the filter, the maximal number of full buckets l is $\lfloor \frac{n}{b} \rfloor$. □

A tighter bound is shown in the analysis of the disabling probability in appendix A.

Chapter 3

Security Analysis

We now proceed to the security analysis of the Cuckoo filter with deletions. In the following, we describe the simulation-based framework for analyzing the adversarial correctness of Cuckoo filters.

We formally define the game Real-or-Ideal in Figure 3.1. The game consists of an adversary $\mathcal{A}$, a simulator $\mathcal{S}$ and a distinguisher $\mathcal{D}$. The adversary $\mathcal{A}$ interacts with the filter's instantiation Π using three oracles. First, the filter is initialized and is empty at the beginning. Then, $\mathcal{A}$ can insert any element in Π through the Insert oracle and delete through the Delete oracle. The adversary can also use the Query oracle to perform a membership query in Π. Hence, $\mathcal{A}$ can execute the three possible operations in a Cuckoo filter.
The game first flips a coin d to determine whether the world is real ($d = 0$) or ideal ($d = 1$). In the first case, an empty Cuckoo filter is initialized and the adversary is given the three oracles to perform insertions, deletions and membership queries on it. In the second case, the adversary is interacting with the simulator which is defined in the next section. Observe that from $\mathcal{A}$'s point of view, he is also interacting with the three oracles in the simulator. The adversary $\mathcal{A}$ produces some output *out*, which is given to the distinguisher $\mathcal{D}$. Then, $\mathcal{D}$ returns the final guess ($d = 1$ or $d = 0$). Note that the output of the adversary is not restricted.

As in [10], we study the security in the "private" setting, where the adversary cannot reveal and observe the state of the filter. Moreover, the adversary cannot reinsert an element already inserted in the filter. Indeed, the insertion of $2b$ times the same element inevitably disables the structure, which is not the case in the simulator. However, the adversary can reinsert an element that was deleted. Because of this rule, we cannot allow the adversary to reveal the state as he would be able to deduce that the simulator inserts freshly sampled elements. He can insert a deleted element a second time and observe that the newly inserted fingerprint is different from the first time. We call this restricted adversary a *no-reinsertion* adversary.

Definition 3.1 (*No-reinsertion* adversary) *An adversary $\mathcal{A}$ is a no-reinsertion adversary iff for every query* $\mathsf{Insert}(x)$ *from $\mathcal{A}$, x was not previously inserted by $\mathcal{A}$ or x was inserted and then deleted.*

In this setting, the harm $\mathcal{A}$ can do in the real world can be quantified by the ability of $\mathcal{D}$ to distinguish between the two worlds. We can now define the notion of adversarial correctness using the Real-or-Ideal game we just described.

Definition 3.2 (Adversarial correctness) *Let Π be a Cuckoo filter with deletions and public parameters pp, and let R_K be keyed function family. We say Π is $(q_{in}, q_{qry}, q_{del}, t_a, t_s, t_d, \epsilon)$-adversarially correct under no reinsertion if, for all no-reinsertion adversaries (Def 3.1) $\mathcal{A}$ running in time at most t_a and making q_{in}, q_{qry}, q_{del} queries to oracles* $\mathsf{Insert}, \mathsf{Query}, \mathsf{Delete}$ *respectively in the Real-or-Ideal game (Figure 3.1) with the simulator $\mathcal{S}$ (Figure 3.2) that runs in time at most t_s, and for all distinguisher $\mathcal{D}$ running in time at most t_d, we have*

$$\mathsf{Adv}^{RoI}_{\Pi,\mathcal{A},\mathcal{S}}(\mathcal{D}) := |\Pr[Real(\mathcal{A},\mathcal{D}) = 1] - \Pr[Ideal(\mathcal{A},\mathcal{D},\mathcal{S}) = 1]| \leq \epsilon.$$

Real-or-Ideal$(\mathcal{A}, \mathcal{S}, \mathcal{D}, pp)$

1 : $d \leftarrow\$ \{0,1\}$
2 : **if** $d = 0$
3 : $\quad K \leftarrow\$ \mathcal{K}; F \leftarrow R_K$
4 : $\quad \sigma \leftarrow \mathsf{setup}(pp)$
5 : $\quad out \leftarrow\$ \mathcal{A}^{\mathsf{Insert},\mathsf{Query},\mathsf{Delete}}$
6 : **else**
7 : $\quad out \leftarrow\$ \mathcal{S}(\mathcal{A}, pp)$
8 : **return** $d' \leftarrow\$ \mathcal{D}(out)$

Oracle $\mathsf{Insert}(x)$

1 : $(b, \sigma) \leftarrow\$ \mathsf{insert}^F(x, \sigma)$
2 : **return** b

Oracle $\mathsf{Query}(x)$

1 : **return** $\mathsf{query}^F(x, \sigma)$

Oracle $\mathsf{Delete}(x)$

1 : $(b, \sigma) \leftarrow\$ \mathsf{delete}^F(x, \sigma)$
2 : **return** b

Figure 3.1: Real-or-Ideal game

We construct a simulator in Figure 3.2. We base our definition on a master thesis from Ella Kummer [10]. She does a similar study but for counting filters with deletions. The simulator defines three oracles that simulate the functions of a Cuckoo filter. To construct those oracles, we still instantiate a Cuckoo filter in the background. However, instead of letting the adversary directly interact with the filter, the simulator proxies the requests.

Insertion. On the insertion of x, the simulator inserts a random element y

instead of the given one. The simulator stores the pair x and y. So, if an element is inserted a second time, the simulator does not insert it again in the filter. The counter ctr_{in} counts the number of successful insertions. Overall, random elements are inserted to ensure that the adversary cannot influence the state by targeting some elements.

Membership query. On the querying of x, the response is necessarily positive if the element was previously inserted by the adversary. Otherwise, a random element is queried and the result is returned. The goal is to simulate the honest false positive rate of a Cuckoo filter. Observe that the membership query's result of a non-inserted element cannot change unless new elements are inserted or deleted. Hence, the data structures FP_{bool}, LQCTR_{in} and LQCTR_{del} (Last Query Counter) are necessary for consistency. If x was a false positive and no elements were deleted since, then x must still be a false positive (line 5-7). If x was not a false positive and no elements were inserted since, then the query's response must still be negative (line 8-10).

Deletion. On the deletion of x, the corresponding element y is deleted if x was previously inserted by the adversary. Recall that a random element y is inserted in the filter instead of x. Therefore, we maintain the consistency between insertions and deletions. Consequently, no false negatives are possible in the simulator. The counter ctr_{del} counts the number of deletions.

Overall, we remove the dependency between insert/delete and query for non-inserted elements. The adversary learns nothing useful from querying an element before inserting it. Furthermore, the simulator is consistent from the adversary's perspective.

We now proceed to state and prove the correctness bound for Cuckoo filters with deletions.

Theorem 3.3 (Adversarial correctness) *Let q_{in}, q_{qry} and q_{del} be non-negative integers, and let $t_a, t_d > 0$. Let $F : \mathfrak{D} \to \mathfrak{R}$. Let Π be the (wrapped) Cuckoo filter deletions defined in Figure 2.1) with public parameters pp and oracle access to F. If $R_K : \mathfrak{D} \to \mathfrak{R}$ is a $(q_{in} + q_{qry} + q_{del}, t_a + t_d, \epsilon)$-secure pseudorandom function with key $K \leftarrow\!\!\$\ \mathcal{K}$, then Π is $(q_{in}, q_{qry}, q_{del}, t_a, t_s, t_d, \epsilon')$-adversarially correct under no reinsertion with respect to the simulator in Figure 3.2, where*

$$\epsilon' = \epsilon + q_{in} \cdot P_{\Pi,pp}(D|q_{in} - 1) + (2q_{qry} + q_{del}) \cdot \overline{P_{\Pi,pp}}(FP).$$

Proof We introduce two games Real-or-G and G-or-Ideal (Figure 3.3) using an intermediary step G. We then state and prove two lemmas which bound the advantage of the distinguisher in the new games, that is we show the closeness of first *Real* and G and then G and *Ideal*. In the end, we combine the lemmas to prove the theorem. □

Simulator $\mathcal{S}(\mathcal{A}, pp)$

$F \leftarrow\$ \mathsf{Func}[\mathfrak{D}, \mathfrak{R}]$
$\sigma \leftarrow \mathsf{setup}(pp)$
// whether an element is inserted and its value
$\mathsf{inserted}_{bool} \leftarrow \{\}$
$\mathsf{inserted}_{value} \leftarrow \{\}$
// whether an element is a False Positive
$\mathsf{FP}_{bool} \leftarrow \{\}$
// number of insertion/deletion at the Last Query
$\mathsf{LQCTR}_{in} \leftarrow \{\}$
$\mathsf{LQCTR}_{del} \leftarrow \{\}$
// counter of total insertion/deletion
$\mathsf{ctr}_{in} \leftarrow 0$
$\mathsf{ctr}_{del} \leftarrow 0$
return $\mathcal{A}^{\mathsf{InsertSim},\mathsf{QuerySim},\mathsf{DeleteSim}}$

Oracle $\mathsf{QuerySim}(x)$

$c \leftarrow \mathsf{query}^{\mathsf{Id}_{\mathfrak{R}}}(y \leftarrow\$ \mathfrak{R}, \sigma)$
if $\mathsf{inserted}_{bool}[x] = \top$
  $c \leftarrow \top$
elseif $\mathsf{FP}_{bool}[x] = \top$ and $\mathsf{LQCTR}_{del} = \mathsf{ctr}_{del}$
  // x was a FP and no deletion happened since
  $c \leftarrow \top$
elseif $\mathsf{FP}_{bool}[x] = \bot$ and $\mathsf{LQCTR}_{in} = \mathsf{ctr}_{in}$
  // x was not a FP and no insertion happened since
  $c \leftarrow \bot$
else
  $\mathsf{FP}_{bool}[x] \leftarrow c$
$\mathsf{LQCTR}_{in}[x] \leftarrow \mathsf{ctr}_{in}$
$\mathsf{LQCTR}_{del}[x] \leftarrow \mathsf{ctr}_{del}$
return c

Oracle $\mathsf{InsertSim}(x)$

if $\mathsf{inserted}_{bool}[x] = \bot$
  $(a, \sigma) \leftarrow \mathsf{insert}^{\mathsf{Id}_{\mathfrak{R}}}(y \leftarrow\$ \mathfrak{R}, \sigma)$
  if $a = \top$
    $\mathsf{ctr}_{in} \leftarrow \mathsf{ctr}_{in} + 1$
    $\mathsf{inserted}_{bool}[x] \leftarrow \top$
    $\mathsf{inserted}_{value}[x] \leftarrow y$
  endif
else
  $a \leftarrow \top$
return a

Oracle $\mathsf{DeleteSim}(x)$

if $\mathsf{inserted}_{bool}[x] = \top$
  $(b, \sigma) \leftarrow \mathsf{delete}^{\mathsf{Id}_{\mathfrak{R}}}(\mathsf{inserted}_{value}[x], \sigma)$
else
  $b \leftarrow \bot$
if $b = \top$
  $\mathsf{ctr}_{del} \leftarrow \mathsf{ctr}_{del} + 1$
  $\mathsf{inserted}_{bool}[x] \leftarrow \bot$
return b

Figure 3.2: Simulator

Lemma 3.4 *The difference in probability of an arbitrary t_d-distinguisher $\mathcal{D}$ outputting 1 in experiments of game Real-or-G in Figure 3.3 with a no-reinsertion $(q_{in}, q_{qry}, q_{del}, t_a)$-Cuckoo filter adversary $\mathcal{A}$ is bounded by the maximal PRF advantage ϵ of a $(q_{in} + q_{qry} + q_{del}, t_a + t_d, \epsilon)$-PRF adversary attacking R_K:*

$$\mathsf{Adv}^{\textit{Real-or-G}}_{\Pi,\mathcal{A}}(\mathcal{D}) := |\Pr[\textit{Real}(\mathcal{A}, \mathcal{D}) = 1] - \Pr[G(\mathcal{A}, \mathcal{D}) = 1]| \leq \epsilon.$$

Proof In G, the keyed PRF used in the PRF-wrapped Cuckoo filter is replaced with a random function $F \leftarrow\$ \mathsf{Func}[\mathfrak{D}, \mathfrak{R}]$. In the Real-or-G game,

Real-or-G$(\mathcal{A}, \mathcal{D}, pp)$	G-or-Ideal$(\mathcal{A}, \mathcal{S}, \mathcal{D}, pp)$
1 : $d \leftarrow\$ \{0,1\}$	1 : $d \leftarrow\$ \{0,1\}$
2 : **if** $d = 0$ // Real	2 : **if** $d = 0$ // G
3 : $K \leftarrow\$ \mathcal{K}; F \leftarrow\$ R_K$	3 : $F \leftarrow\$ \mathsf{Func}[\mathfrak{D}, \mathfrak{R}]$
4 : **else** // G	4 : $\sigma \leftarrow \mathsf{setup}(pp)$
5 : $F \leftarrow\$ \mathsf{Func}[\mathfrak{D}, \mathfrak{R}]$	5 : $out \leftarrow\$ \mathcal{A}^{\mathsf{Insert,Query,Delete}}$
6 : $\sigma \leftarrow \mathsf{setup}(pp)$	6 : **else** // Ideal
7 : $out \leftarrow\$ \mathcal{A}^{\mathsf{Insert,Query,Delete}}$	7 : $out \leftarrow\$ \mathcal{S}(\mathcal{A}, pp)$
8 : $d' \leftarrow\$ \mathcal{D}(out)$	8 : $d' \leftarrow\$ \mathcal{D}(out)$
9 : **return** d'	9 : **return** d'

Figure 3.3: Intermediate game G

$\mathcal{A}$ interacts either with a PRF-wrapped filter (*Real*) or with a filter wrapped with F (G). In Figure 3.4, we construct an adversary $\mathcal{B}$ of the PRF experiment (Figure 1.1). The adversary $\mathcal{B}$ invokes the adversary $\mathcal{A}$ of the Real-or-G game, and the oracle RoR handles the queries to Π in Insert, Query and Delete. This construction proves that the PRF experiment exactly reduces to the Real-or-G game (and vice-versa). Indeed, when $b = 0$, RoR applies a keyed PRF R_K, which is equivalent to $d = 0$ in the Real-or-G game. The same argument is true for $b = 1$, $\mathcal{B}$ is running G for $\mathcal{A}$ where a truly random function F in RoR is used to handle $\mathcal{A}$ oracle queries to Π. Hence, since R_K is a $(q_{in} + q_{qry} + q_{del}, \epsilon)$-secure PRF by assumption, we can deduce that

$$\mathsf{Adv}^{\text{Real-or-G}}_{\Pi,\mathcal{A}}(\mathcal{D}) = \mathsf{Adv}^{PRF}_{R}(\mathcal{B}) \leq \epsilon.$$

Adversary $\mathcal{B}^{\mathsf{RoR}}$
1 : $F \leftarrow\$ \mathsf{RoR}$
2 : $\sigma \leftarrow \mathsf{setup}(pp)$
3 : **return** $d' \leftarrow\$ \mathcal{D}(\mathcal{A}^{\mathsf{Insert,Query,Delete}})$

Figure 3.4: PRF adversary $\mathcal{B}$

Lemma 3.5 *Let $\boldsymbol{E}^i_\lambda$ be the event that denotes the divergence between $G*$ and Ideal in the i^{th} response of oracle λ to $\mathcal{A}$:*

$$\boldsymbol{E}^i_\lambda = [\textit{The first mismatch is observed during the } i^{th} \textit{ query to the oracle } \lambda].$$

Then, let $\boldsymbol{E}$, $\boldsymbol{E}'$, *and* $\boldsymbol{E}''$ *be the events that denote the divergence for each oracle:*

$$\begin{aligned}\boldsymbol{E} &= [\boldsymbol{E}^i_{\mathit{QuerySim}} \textit{ for some } i \in [q_{qry}]],\\ \boldsymbol{E}' &= [\boldsymbol{E}^i_{\mathit{DeleteSim}} \textit{ for some } i \in [q_{del}]],\\ \boldsymbol{E}'' &= [\boldsymbol{E}^i_{\mathit{InsertSim}} \textit{ for some } i \in [q_{in}]].\end{aligned}$$

The difference in probability of an arbitrary distinguisher $\mathcal{D}$ *outputting 1 in experiments of game G-or-Ideal with a no-reinsertion* $(q_{in}, q_{qry}, q_{del}, t_a)$*-Cuckoo filter adversary* $\mathcal{A}$ *is bounded by* $\Pr[\boldsymbol{E}] + \Pr[\boldsymbol{E}'] + \Pr[\boldsymbol{E}'']$.

$$\begin{aligned}\mathsf{Adv}^{\textit{G-or-Ideal}}_{\Pi,\mathcal{A},\mathcal{S}}(\mathcal{D}) &:= |\Pr[G(\mathcal{A},\mathcal{D}) = 1)] - \Pr[\mathit{Ideal}(\mathcal{A},\mathcal{D},\mathcal{S}) = 1] \,|\\ &\leq \Pr[\boldsymbol{E}] + \Pr[\boldsymbol{E}'] + \Pr[\boldsymbol{E}''].\end{aligned}$$

Proof We now prove a bound on the closeness of G and *Ideal*. We again introduce an intermediary step $G*$. In $G*$, the simulator is modified in a way that displayed in Figure 3.5 using framed instructions. We show that G and $G*$ behave in the same way from the adversary's point of view. First, both use the same random function F when interacting with the filter. On **insertion**, the element given by the adversary is inserted in the filter using insert^F and the result is returned (line 3-4, 12). A divergence happens when an already inserted element is given, but this event is not possible under the *no-reinsertion* assumption. On **deletion**, the element is deleted using delete^F and the result returned in any case (line 1,6,10). On **querying**, we distinguish two cases. If the element was previously inserted, the result of query^F is returned (line 2,4,16). If not, the simulator either returns a cached result of the query if the state did not change since the insertion (line 5-10) or otherwise the result of query^F (line 12). In conclusion, queries in G and $G*$ always trigger the same updates and agree on the result.

We finally show that the divergence between $G*$ and *Ideal* can be characterized by the three events defined earlier.
In InsertSim, we observe that the given element x is inserted in $G*$ while a new random element is inserted in *Ideal*. Thus, a difference can happen and be noticed by an adversary when $a^{\mathit{Ideal}} \neq a^{G*}$. For instance, an insertion could fail in *Ideal* because the filter is disabled, but not in $G*$ where the filter is still open because different elements were inserted. The probability that this discrepancy is the first one observed by the adversary is $\Pr[\mathbf{E}'']$.
In DeleteSim, the simulator calls delete on x in any case in $G*$, but in *Ideal* the element is deleted only if it was previously inserted. Thus, we are interested in the event $b^{\mathit{Ideal}} \neq b^{G*}$. Typically, a difference is observed when a non-inserted element is successfully deleted in $G*$ and creates a false negative, which is not possible in *Ideal*. The probability that this discrepancy is the first one observed by the adversary is $\Pr[\mathbf{E}']$.

Simulator $\mathcal{S}(\mathcal{A}, pp)$

$F \leftarrow\!\$\ Funcs[\mathfrak{D}, \mathfrak{R}]$
$\sigma \leftarrow \mathsf{setup}(pp)$ // Ideal
$\sigma' \leftarrow \mathsf{setup}(pp)$ // $G*$
// whether an element is inserted and its value
$\mathsf{inserted}_{bool} \leftarrow \{\}$
$\mathsf{inserted}_{value} \leftarrow \{\}$
// whether an element is a False Positive
$\mathsf{FP}_{bool} \leftarrow \{\}$
// number of insertion/deletion at the Last Query
$\mathsf{LQCTR}_{in} \leftarrow \{\}$
$\mathsf{LQCTR}_{del} \leftarrow \{\}$
// counter of total insertion/deletion
$\mathsf{ctr}_{in} \leftarrow 0$
$\mathsf{ctr}_{del} \leftarrow 0$
return $\mathcal{A}^{\mathsf{InsertSim,QuerySim,DeleteSim}}$

Oracle InsertSim(x)

if $\mathsf{inserted}_{bool}[x] = \bot$
  $(a^{Ideal}, \sigma) \leftarrow \mathsf{insert}^{\mathsf{Id}_{\mathfrak{R}}}(y \leftarrow\!\$\ \mathfrak{R}, \sigma)$
  $(a^{G*}, \sigma') \leftarrow \mathsf{insert}^{F}(x, \sigma')$
  $a \leftarrow a^{Ideal}$ $\boxed{a \leftarrow a^{G*}}$ // Ideal $\boxed{G*}$
  if $a = \top$
    $\mathsf{ctr}_{in} \leftarrow \mathsf{ctr}_{in} + 1$
    $\mathsf{inserted}_{bool}[x] \leftarrow \top$
    $\mathsf{inserted}_{value}[x] \leftarrow y$
  endif
else
  $a \leftarrow \top$
return a

Oracle QuerySim(x)

$c^{Ideal} \leftarrow \mathsf{query}^{\mathsf{Id}_{\mathfrak{R}}}(y \leftarrow\!\$\ \mathfrak{R}, \sigma)$
$c^{G*} \leftarrow \mathsf{query}^{F}(x, \sigma')$
if $\mathsf{inserted}_{bool}[x] = \top$
  $c \leftarrow \top$ $\boxed{c \leftarrow c^{G*}}$ // Ideal $\boxed{G*}$
elseif $\mathsf{FP}_{bool}[x] = \top$ and $\mathsf{LQCTR}_{del} = \mathsf{ctr}_{del}$
  // x was a FP and no deletion happened since
  $c \leftarrow \top$
elseif $\mathsf{FP}_{bool}[x] = \bot$ and $\mathsf{LQCTR}_{in} = \mathsf{ctr}_{in}$
  // x was not a FP and no insertion happened since
  $c \leftarrow \bot$
else
  $c \leftarrow c^{Ideal}$ $\boxed{c \leftarrow c^{G*}}$ // Ideal $\boxed{G*}$
  $\mathsf{FP}_{bool}[x] \leftarrow c$
$\mathsf{LQCTR}_{in}[x] \leftarrow \mathsf{ctr}_{in}$
$\mathsf{LQCTR}_{del}[x] \leftarrow \mathsf{ctr}_{del}$
return c

Oracle DeleteSim(x)

$(b^{G*}, \sigma') \leftarrow \mathsf{delete}^{F}(x, \sigma')$
if $\mathsf{inserted}_{bool}[x] = \top$
  $(b^{Ideal}, \sigma) \leftarrow \mathsf{delete}^{\mathsf{Id}_{\mathfrak{R}}}(\mathsf{inserted}_{value}[x], \sigma)$
else
  $b^{Ideal} \leftarrow \bot$
$b \leftarrow b^{Ideal}$ $\boxed{b \leftarrow b^{G*}}$ // Ideal $\boxed{G*}$
if $b = \top$
  $\mathsf{ctr}_{del} \leftarrow \mathsf{ctr}_{del} + 1$
  $\mathsf{inserted}_{bool}[x] \leftarrow \bot$
return b

Figure 3.5: Simulator with $G*$

In QuerySim, if the element x was previously inserted, the result is always positive in *Ideal*, but in $G*$ the result of the membership query is returned and x can be a false negative. Otherwise, the oracle either returns a cached version of the query if the state allows it or returns the result of the mem-

bership query. An element can be a false positive in one filter but not in the other, since the filters contain different elements in $G*$ and *Ideal*. Thus, we consider the event $c^{Ideal} \neq c^{G*}$ when the element was not inserted or $c^{G*} \neq \top$ when the element was inserted. The probability that this discrepancy is the first one observed by the adversary is $\Pr[\mathbf{E}]$.
Therefore, $G*$ (and hence G) is indistinguishable from *Ideal* up to the realization of one of the event $\mathbf{E}$, $\mathbf{E}'$ or $\mathbf{E}''$, which concludes the proof:

$$|\Pr[G(\mathcal{A},\mathcal{D})=1)] - \Pr[Ideal(\mathcal{A},\mathcal{D},\mathcal{S})=1]\ | \leq \Pr[\mathbf{E}] + \Pr[\mathbf{E}'] + \Pr[\mathbf{E}''].$$

We derive a bound for the three events in the following. □

Lemma 3.6 (Divergence in QuerySim) *Let $\boldsymbol{E} = [\boldsymbol{E}^i_{\mathsf{QuerySim}}$ for some $i \in [q_{qry}]]$. Then,*

$$\Pr[\boldsymbol{E}] \leq 2q_{qry} \cdot \overline{P_{\Pi,pp}}(FP).$$

Proof We focus on the QuerySim oracle. We consider an adversary that inserts q_{in} elements $V = \{y_1, ..., y_{q_{in}}\}$ and queries q_{qry} elements $U = \{x_1, ..., x_{q_{qry}}\}$. We first observe that we only need to consider queries on elements that were not inserted. Indeed, a query response on an inserted element that differs in game *Ideal* and $G*$ can have two causes. First, if the element is a false negative in $G*$, the difference happens earlier in DeleteSim. Second, the element could have been successfully inserted in one game but not the other, in which case the difference happens in InsertSim. Thus, we can assume that $V \cap U = \emptyset$, otherwise the event $\mathbf{E}$ cannot happen since we are interested in the first mismatch across all oracles. We bound the probability of the event $\mathbf{E}$ by using the union bound and simplifying the terms:

$$\begin{aligned}
\Pr[\mathbf{E}] &= \Pr[\mathbf{E}^i_{\mathsf{QuerySim}} \text{ for some } i \in [q_{qry}]] \\
&\leq \sum_{i=1}^{q_{qry}} \Pr[c_i^{Ideal} \neq c_i^{G*}] \\
&= \sum_{i=1}^{q_{qry}} \Pr[c_i^{Ideal} = \top, c_i^{G*} = \bot] + \Pr[c_i^{Ideal} = \bot, c_i^{G*} = \top] \\
&\leq \sum_{i=1}^{q_{qry}} \Pr[c_i^{Ideal} = \top] + \Pr[c_i^{G*} = \top].
\end{aligned}$$

We first compute the probability in the *Ideal* game. We can replace c_i^{Ideal} by its definition in the simulator. We denote $\sigma^{(j)}$ the state of the filter after j insertions $\{\tilde{y}_1, ..., \tilde{y}_j\}$. Recall that for each element y_k inserted by the adversary, the simulator inserts a sampled element $\tilde{y}_k$. Moreover, when the adversary queries an non-inserted element x_k, the simulator queries a sampled element $\tilde{x}_k$. Note that the exact behavior of the filter depends on the randomness of the simulator *Ideal* for the sampled elements, and of the adversary $\mathcal{A}$ for the

state $\sigma^{(j)}$. The randomness is represented by respectively *Ideal*'s coins and $\mathcal{A}$'s coins. Thus,

$$\Pr[c_i^{Ideal} = \top] = \Pr_{\substack{\textit{Ideal's coins} \\ r \xleftarrow{\$} R \\ \mathcal{A}\textit{'s coins}}}[c \leftarrow \mathsf{query}^{\mathsf{Id}_{\mathfrak{R}}}(\tilde{x} \xleftarrow{r}\$\, \mathfrak{R}, \sigma^{(j)}) : c = \top].$$

This is the probability that, given a state $\sigma^{(j)}$, a random element $\tilde{x}$ is a false positive. Thus, we can bound by the universal false positive probability $\overline{P_{\Pi,pp}}(FP)$ (Def 2.4), which applies on any state given that the queried element is random:

$$\Pr[c_i^{Ideal} = \top] \leq \overline{P_{\Pi,pp}}(FP).$$

Similarly, we bound the probability in the $G*$ game. Unlike the *Ideal* game, the elements of the adversary are inserted and queried. However, Cuckoo filters are function-decomposable and F is a random function. So,

$$\begin{aligned}\Pr[c_i^{G*} = \top] &= \Pr_{\substack{\textit{Ideal's coins} \\ G*\textit{'s coins} \\ \mathcal{A}\textit{'s coins}}}\left[c \leftarrow \mathsf{query}^{\mathsf{F}}(x_i, \sigma^{(j)}) : c = \top\right] \\ &= \Pr_{\substack{\textit{Ideal's coins} \\ G*\textit{'s coins} \\ \mathcal{A}\textit{'s coins}}}\left[c \leftarrow \mathsf{query}^{\mathsf{Id}_{\mathfrak{R}}}(F(x_i), \sigma^{(j)}) : c = \top\right].\end{aligned}$$

We observe that we are again computing the probability that a random element $F(x_i)$ matches an inserted element and is a false positive. We assume that F is a perfectly random function. Furthermore, the adversary cannot learn about F. Because we are in the *Ideal* world, the adversary does not learn the responses to his queries insert^F, delete^F or query^F by construction. Therefore, we are in a situation similar to the *Ideal* game, where the queried element is completely random:

$$\Pr[c_i^{G*} = \top] \leq \overline{P_{\Pi,pp}}(FP).$$

Finally, putting everything together:

$$\begin{aligned}\Pr[\mathbf{E}] &\leq \sum_{i=1}^{q_{qry}} \Pr[c_i^{Ideal} = \top] + \Pr[c_i^{G*} = \top] \\ &\leq \sum_{i=1}^{q_{qry}} \overline{P_{\Pi,pp}}(FP) + \overline{P_{\Pi,pp}}(FP) \\ &= 2q_{qry} \cdot \overline{P_{\Pi,pp}}(FP). \qquad \square\end{aligned}$$

Lemma 3.7 (Divergence in DeleteSim) *Let* $\boldsymbol{E}' = \Pr[\boldsymbol{E}^i_{\mathsf{DeleteSim}}$ *for some* $i \in [q_{del}]]$. *Then,*

$$\Pr[\boldsymbol{E}'] \leq 2q_{del} \cdot \overline{P_{\Pi,pp}}(FP).$$

Proof We now bound the second event $\mathbf{E}'$ which focuses on the DeleteSim oracle. The adversary deletes q_{del} elements $Z = \{z_0, ..., z_{q_{del}}\}$. Similarly to the first event, we first distinguish the two possible cases of mismatch:

$$\begin{aligned}\Pr[\mathbf{E}'] &= \Pr[\mathbf{E}^i_{\mathsf{DeleteSim}} \text{ for some } i \in [q_{del}]] \\ &= \sum_{i=1}^{q_{del}} \Pr[\mathbf{E}^i_{\mathsf{DeleteSim}}, b_i^{Ideal} = \top, b_i^{G*} = \bot] \\ &\quad + \Pr[\mathbf{E}^i_{\mathsf{DeleteSim}}, b_i^{Ideal} = \bot, b_i^{G*} = \top].\end{aligned}$$

We can show that $\Pr[\mathbf{E}^i_{\mathsf{DeleteSim}}, b_i^{Ideal} = \top, b_i^{G*} = \bot] = 0$. If the deletion is successful in the *Ideal* game but not in $G*$, then either the element was inserted in *Ideal* but not in $G*$, in which case the divergence first happens in InsertSim, or the element is a false negative in $G*$, in which case the mismatch happens earlier in DeleteSim. Thus, we can ignore that case and apply the union bound for the second case:

$$\begin{aligned}\Pr[\mathbf{E}'] &= \Pr[\mathbf{E}^i_{\mathsf{DeleteSim}} \text{ for some } i \in [q_{del}]] \\ &= \sum_{i=1}^{q_{del}} \Pr[\mathbf{E}^i_{\mathsf{DeleteSim}}, b_i^{Ideal} = \bot, b_i^{G*} = \top] \\ &\leq \sum_{i=1}^{q_{del}} \Pr[b_i^{Ideal} = \bot, b_i^{G*} = \top].\end{aligned}$$

We now bound $\Pr[b_i^{Ideal} = \bot, b_i^{G*} = \top]$. We observe that the only way $b_i^{Ideal} = \bot$ is when the element was not inserted. Thus,

$$\begin{aligned}\Pr[b_i^{Ideal} = \bot, b_i^{G*} = \top] &= \Pr[b_i^{G*} = \top, z_i \text{ was not inserted}] \\ &\leq \Pr[b_i^{G*} = \top \mid z_i \text{ was not inserted}].\end{aligned}$$

Furthermore, we see from the simulator that $((b, \sigma') \leftarrow \mathsf{delete}^F(z, \sigma') : b = \top)$ if and only if $(c \leftarrow \mathsf{query}^F(z, \sigma') : c = \top)$. Therefore,

$$\begin{aligned}&\Pr[b_i^{G*} = \top \mid z_i \text{ was not inserted}] \\ &= \Pr_{\substack{\textit{Ideal's coins} \\ \textit{G*'s coins} \\ \mathcal{A}\textit{'s coins}}}[(b_i^{G*}, \sigma'^{(i)}) \leftarrow \mathsf{delete}^F(z_i, \sigma'^{(i-1)}) : b_i^{G*} = \top \mid z_i \text{ was not inserted}] \\ &= \Pr_{\substack{\textit{Ideal's coins} \\ \textit{G*'s coins} \\ \mathcal{A}\textit{'s coins}}}[c \leftarrow \mathsf{query}^F(z_i, \sigma'^{(i-1)}) : c = \top \mid z_i \text{ was not inserted}],\end{aligned}$$

which is a probability we have already bounded for the event $\mathbf{E}$. Putting

everything together:

$$\begin{aligned}\Pr[\mathbf{E}'] &\leq \sum_{i=1}^{q_{del}} \Pr_{\substack{\textit{Ideal's coins}\\ \textit{G*'s coins}\\ \textit{A's coins}}}[c \leftarrow \mathsf{query}^F(z_i, \sigma'^{(i-1)}) : c = \top \mid z_i \text{ was not inserted}] \\ &\leq \sum_{i=1}^{q_{del}} \overline{P_{\Pi,pp}}(FP) \\ &= q_{del} \cdot \overline{P_{\Pi,pp}}(FP). \end{aligned}$$ □

Lemma 3.8 (Divergence in InsertSim) *Let* $\boldsymbol{E}'' = \Pr[\boldsymbol{E}^i_{\mathit{InsertSim}}$ *for some* $i \in [q_{in}]]$. *Then,*

$$\Pr[\boldsymbol{E}''] \leq 2q_{in} \cdot P_{\Pi,pp}(D|q_{in} - 1).$$

Proof We finally bound the third event which focuses on the InsertSim oracle:

$$\begin{aligned}\Pr[\mathbf{E}''] &= \Pr[\mathbf{E}^i_{\mathsf{InsertSim}} \text{ for some } i \in [q_{in}]] \\ &\leq \sum_{i=1}^{q_{in}} \Pr[\mathbf{E}^i_{\mathsf{InsertSim}}, a_i^{Ideal} \neq a_i^{G*}] \\ &= \sum_{i=1}^{q_{in}} \Pr[\mathbf{E}^i_{\mathsf{InsertSim}}, a_i^{Ideal} = \top, a_i^{G*} = \bot] + \Pr[\mathbf{E}^i_{\mathsf{InsertSim}}, a_i^{Ideal} = \bot, a_i^{G*} = \top] \\ &\leq \sum_{i=1}^{q_{in}} \Pr[\mathbf{E}^i_{\mathsf{InsertSim}}, a_i^{Ideal} = \bot] + \Pr[\mathbf{E}^i_{\mathsf{InsertSim}}, a_i^{G*} = \bot]. \end{aligned}$$

We first compute the probability in the *Ideal* game that an insertion fails for the first time during the i^{th} insertion query. We replace with the definition in the simulator:

$$\Pr[\mathbf{E}^i_{\mathsf{InsertSim}}, a_i^{Ideal} = \bot] = \Pr_{\substack{\textit{Ideal's coins}\\ r \leftarrow\$ R\\ \textit{A's coins}}}[(a, \sigma^{(i)}) \leftarrow \mathsf{insert}^{\mathsf{Id}_{\mathfrak{R}}}(\tilde{y} \xleftarrow{r}\$ \mathfrak{R}, \sigma^{(i-1)}) : a = \bot].$$

This is the probability that the filter in state $\sigma^{(i-1)}$ is disabled. The state $\sigma^{(i-1)}$ contains at most $i-1$ elements and we are computing the probability of the first insertion failure. Thus, we argue that we can bound it by the first-time disabling probability $P_{\Pi,pp}(D|i-1)$ (Def 2.7). First, the inserted element $\tilde{y}$ is completely random. Then, the elements inserted in σ^{i-1} are also random. Note that the fingerprints do not play any role in the disabling probability, rather only the load of the buckets is determinant. Nonetheless, we must carefully study the consequences of deletions on the state. We observe that deleting an element cannot itself increase the disabling probability, because it can only decrease a bucket's load. Moreover, a deletion can prevent a future eviction or leave an empty slot for another element (if the deleted

element's bucket was full), but in any case the disabling probability does not increase. For instance, if $\mathcal{A}$ deletes z_1 in the full bucket m_1 such that z_2 may be placed into m_1, m_1 is full with and without the deletion, but in the latter the filter contains one more element. Therefore, the adversary cannot increase the load of targeted buckets and we can bound the probability by using an NAI state of $i-1$ elements. Finally, we know that $P_{\Pi,pp}(D|i) \leq P_{\Pi,pp}(D|n)$ for all $i < n$, so the disabling probability is maximal when all elements in V are inserted, $n = q_{in} - 1$. Thus,

$$\begin{aligned}
\Pr[\mathbf{E}^i_{\mathsf{InsertSim}}, a_i^{Ideal} = \perp] &= \Pr_{\substack{\textit{Ideal's coins} \\ r \leftarrow\$ R \\ \mathcal{A}\textit{'s coins}}}[(a, \sigma^{(i)}) \leftarrow \mathsf{insert}^{\mathsf{Id}_{\mathfrak{R}}}(\tilde{y} \xleftarrow{r}\$ \mathfrak{R}, \sigma^{(i-1)}) : a = \perp] \\
&\leq \Pr\left[\begin{array}{c} \sigma^{(0)} \leftarrow\$ \mathsf{setup}(pp) \\ \tilde{y}_1, ..., \tilde{y}_{i-1} \leftarrow\$ \mathfrak{R} \\ \textbf{for } j = 1, ..., i-1 : \\ (\top, \sigma^{(j)}) \leftarrow\$ \mathsf{insert}^{\mathsf{Id}_{\mathfrak{R}}}(y_j, \sigma^{(j-1)}) \\ \tilde{y} \leftarrow\$ \mathfrak{R} : \perp \leftarrow \mathsf{insert}^{\mathsf{Id}_{\mathfrak{R}}}(\tilde{y}, \sigma^{(i-1)}) \end{array}\right] \\
&\leq P_{\Pi,pp}(D|i-1) \\
&\leq P_{\Pi,pp}(D|q_{in} - 1).
\end{aligned}$$

In the $G*$ game, the proof is similar. We are using the function-decomposable property of Cuckoo filters. Since F is a random function, the output $F(y_i)$ is random and we can apply the same bound as in the *Ideal* game. This only holds if $F(y_i)$ is not already in $\sigma^{(i-1)}$. Under the no reinsertion assumption, $\mathcal{A}$ cannot insert the same element twice, or he must delete it first. In both cases, the state $\sigma'^{(i-1)}$ does not contain $F(y_i)$. The same is true for every inserted element. Therefore, $\sigma'^{(i-1)}$ is constructed using at most $i-1$ distinct and random elements:

$$\begin{aligned}
\Pr[\mathbf{E}^i_{\mathsf{InsertSim}}, a_i^{G*} = \perp] &= \Pr_{\substack{\textit{Ideal's coins} \\ G*\textit{'s coins} \\ \mathcal{A}\textit{'s coins}}}[(a, \sigma'^{(i)}) \leftarrow \mathsf{insert}^{F}(y_i, \sigma'^{(i-1)}) : a = \perp] \\
&\leq \Pr\left[\begin{array}{c} \sigma^{(0)} \leftarrow\$ \mathsf{setup}(pp) \\ [y_1, ..., y_{i-1}] \leftarrow\$ \{S \in \mathcal{P}_{\text{lists}}(\mathfrak{D}) \mid |S| = i-1\} \\ \textbf{for } j = 1, ..., i-1 : \\ (\top, \sigma^{(j)}) \leftarrow\$ \mathsf{insert}^{F}(x_j, \sigma^{(j-1)}) \\ y \leftarrow\$ \mathfrak{D} \setminus V : \perp \leftarrow \mathsf{insert}^{F}(y, \sigma^{(i-1)}) \end{array}\right] \\
&= P_{\Pi,pp}(D|i-1) \\
&\leq P_{\Pi,pp}(D|q_{in} - 1).
\end{aligned}$$

Finally,

$$\begin{aligned}
\Pr[\mathbf{E}''] &\leq \sum_{i=1}^{q_{in}} \Pr[\mathbf{E}^i_{\mathsf{InsertSim}}, a_i^{Ideal} = \perp] + \Pr[\mathbf{E}^i_{\mathsf{InsertSim}}, a_i^{G*} = \perp] \\
&\leq \sum_{i=1}^{q_{in}} P_{\Pi,pp}(D|q_{in}-1) + P_{\Pi,pp}(D|q_{in}-1) \\
&= 2q_{in} \cdot P_{\Pi,pp}(D|q_{in}-1). \qquad \square
\end{aligned}$$

We finally conclude and prove the adversarial correctness Theorem 3.3 by combining Lemmas 3.4 to 3.8. Thus,

$$\begin{aligned}
\mathsf{Adv}^{\mathrm{RoI}}_{\Pi,\mathcal{A},\mathcal{S}}(\mathcal{D}) &= |\Pr[\mathrm{Real}(\mathcal{A},\mathcal{D})=1] - \Pr[\mathrm{Ideal}(\mathcal{A},\mathcal{D},\mathcal{S})=1]\,| \\
&\leq |\Pr[\mathrm{Real}(\mathcal{A},\mathcal{D})=1] - \Pr[\mathrm{G}(\mathcal{A},\mathcal{D})=1]\,| \\
&\quad + |\Pr[\mathrm{G}(\mathcal{A},\mathcal{D})=1] - \Pr[\mathrm{Ideal}(\mathcal{A},\mathcal{D},\mathcal{S})=1]\,| \\
&= \mathsf{Adv}^{\text{Real-or-G}}_{\Pi,\mathcal{A}}(\mathcal{D}) + \mathsf{Adv}^{\text{G-or-Ideal}}_{\Pi,\mathcal{A},\mathcal{S}}(\mathcal{D}) \\
&\leq \epsilon + \Pr[\mathbf{E}] + \Pr[\mathbf{E}'] + \Pr[\mathbf{E}''] \\
&\leq \epsilon + q_{in} \cdot P_{\Pi,pp}(D|q_{in}-1) + (2q_{qry} + q_{del}) \cdot \overline{P_{\Pi,pp}}(FP).
\end{aligned}$$

Chapter 4

Conclusion

We have provided a formal security analysis of Cuckoo filters with deletions. We have first extended the syntax to describe the behavior of deletions. Using a simulation-based framework, we have defined the adversarial correctness of Cuckoo filters with deletions and have demonstrated a bound for it. The bound depends on the Cuckoo filter's disabling and false positive probability. We have also defined the universal false positive probability and have shown that the false positive bound holds for any state of the Cuckoo filter.
We notice that the proof diverges from [7] since the state generated by an adversary cannot be non-adversarially influenced (NAI) even in the ideal world. Indeed, the authors of [7] have shown in their proof that the state can be considered statistically close to NAI in the insertion-only setting thanks to the keyed PRF. In the case of counting filters, [10] also demonstrates a bound that depends on the NAI false positive probability, even when deletions are allowed. Generally, reducing the problem to the NAI case is convenient since we know how to analyse that setting.
However, in Cuckoo filters, deletions allow to reset the state and to conduct adaptive attacks. More importantly, the outcome of an insertion is non-deterministic and depends on the current state. Therefore, the possible sequences of insertions and deletions can produce different states, which could be adversarially influenced. Nevertheless, we were still able to derive bounds by carefully analysing the construction of Cuckoo filters and defining the universal false positive probability. In that way, we could bound the correctness without necessarily having a NAI state. This observation could also simplify the insertion-only proof in [7], but do not generalize to other AMQ-PDS.

This work can be extended in several directions:

- Remove the no reinsertion assumption and extend the framework to any adversary. This requires to handle the trivial disabling of the filter

and to update the proof where distinct elements are assumed.

- Derive a tighter bound for the adversarial correctness. Currently, the bound is linear with the number of queries, which implies its value is reasonably low only when the adversary has few queries.
- Analyze the security in the ”public” setting, where the adversary can reveal the internal state.
- Provide concrete insights on the deployement of Cuckoo filters given the security analysis.

Appendix A

Disabling probability

We derive a bound on the probability that a Cuckoo filter is disabled. Recall that a filter is disabled when, on insertion, elements are consecutively evicted a given number of times and an element still cannot be placed in its bucket. The element is then placed in the stash and the filter is disabled as long as the stash is full.

We will assume in the disabling proof that the number of tags 2^{λ_T} is smaller than the number of buckets 2^{λ_I} (which is the case in practice). This assumption implies that the indices of buckets are not uniformly distributed. Indeed, $i_1 = H_I(x)$ is uniformly distributed because the domain of x is greater than the domain of H_I, but $i_2 = i_1 \oplus H_I(tag)$ is not because the number of tags is smaller than number of possible buckets. Therefore, there are only 2^{λ_T} possible indices i_2. This observation makes the derivation non-trivial.

Theorem A.1 (First-time disabling probability) *Let n be a non-negative integer. Let Π be an insertion-only Cuckoo filter with public parameters $pp = (\lambda_I, \lambda_T, b, u)$ and $m = 2^{\lambda_I}$. Then, the probability that an insertion fails for the first time when n elements are already in the filter is bounded by*

$$n^{b+2} \left(\frac{n}{rb+n}\right)^{u+1} \left(\frac{1}{m}(1+\frac{1}{2^{\lambda_T}})\right)^{b},$$

where $r = (m - \left\lfloor \frac{n}{b} \right\rfloor)(1 - (1 - \frac{2^{\lambda_T}}{m})^{\lfloor n/b \rfloor})$.

First, we observe that we can condition the disabling probability on the number of full buckets, which are the cause of evictions. We define B_l^n the event that l buckets are full after n insertions. Then,

$$\begin{aligned}
&\Pr[(n+1)^{th} \text{ insertion fails}] \\
&= \Pr[n^{th} \text{ insertion fills the stash}] \\
&= \sum_{l=2}^{m} \Pr[n^{th} \text{ insertion fills the stash} \mid B_l^{n-1}] \Pr[B_l^{n-1}].
\end{aligned}$$

We first compute the probability of having l full buckets. Then, bounding the probability of filling the stash becomes trivial knowing the number of full buckets.

Lemma A.2 *Let n be a non-negative integer. Let Π be an insertion-only Cuckoo filter with public parameters $pp = (\lambda_I, \lambda_T, b, u)$ and $m = 2^{\lambda_I}$. Let*

$$P_k(n) = \Pr\left[\begin{array}{c} n \text{ elements are inserted when } k \text{ buckets are full and} \\ \text{the } (k+1)^{th} \text{ bucket is filled on the last insertion} \end{array}\right].$$

Let the sequences of non-negative integers $n_0, n_1, ..., n_l$ be such that:

1. $b \leq n_k \leq n - b(l-k-1)\ \forall k < l$
2. $0 \leq n_l \leq n$
3. $\sum_k n_k = n$

Let $C(n_0, ..., n_{l-1})$ be the number of possible ways the sequence of $n_0, ..., n_{l-1}$ insertions is observed, where n_i is the number of elements inserted when i buckets are full. Then, the probability to have l full buckets after n insertions is bounded by

$$\binom{m}{l} \sum_{n_0,...,n_{l-1}} C(n_0, ..., n_{l-1}) \cdot P_0(n_0) P_1(n_1) ... P_{l-1}(n_{l-1}).$$

Proof We have n insertions, each happening at time $t_i \in [n]$. A bucket can receive an element at time t_i with a probability depending on k, but k is increasing during the insertion process. Thus, we must consider all possible sequence of insertions, i.e buckets can receive their elements when $k = 0, ..., l$. Therefore, we define the sequence $n_0, ..., n_l$ where n_i is the number of elements inserted when i buckets are full. Moreover, we define P_k to model the states of the filter and their transition. For example, $P_0(n_0) \cdot P_1(n_1) \cdot P_2(n_2)$ is the probability that n_0 elements are inserted when 0 buckets are full and before the first bucket becomes full, n_1 element when 1 bucket is full and n_2 elements when 2 buckets are full.
Besides computing the probability of each sequence, we must also count the number of possible ways a sequence is observed. For instance, let n_0 be the number of elements inserted before the first bucket m_1 becomes full. So m_1 receives a new element at b distinct times $t_i \in [n_0]$. Furthermore, we know that m_1 receives an element at time t_{n_0}, the time the bucket becomes full. Therefore, there are $\binom{n_0-1}{b-1}$ ways m_1 is filled with b elements in n_0 time slots. Then, we count the possible ways of assigning the remaining $n_0 - b$ time slots to other buckets, but each bucket can only receive up to $b-1$ elements. $C(n_0, ..., n_{l-1})$ is the number of possible ways the sequence $n_0, ..., n_{l-1}$ is observed and can be written as:

$$C(n_0, ..., n_{l-1}) = \binom{n_0-1}{b-1}\binom{n_0+n_1-b-1}{b-1} ... \binom{\sum_{i<l} n_i - b(i-1) - 1}{b-1}.$$

Observe that n_l is not included, because we can ignore what happens after the last bucket is filled. The final expression is the sum over all possible sequences of the probability of the sequence and the number of ways to allocate elements in the filter. A factor $\binom{m}{l}$ chooses the l full buckets in the filter. □

Lemma A.3 *Let n and k be non-negative integers. Let Π be an insertion-only Cuckoo filter with public parameters $pp = (\lambda_I, \lambda_T, b, u)$ and $m = 2^{\lambda_I}$. Let r_k be a random variable with a Binomial distribution $B(m-k, p)$. Then, the probability $p_k(n)$ that n elements are inserted in a given bucket when k buckets are full is*

$$p_k(n) = \left(\frac{1}{m}(1 + \frac{k}{r_k})\right)^n \cdot \frac{r_k}{m-k} + \left(\frac{1}{m}\right)^n \cdot \frac{m-k-r_k}{m-k}.$$

Proof When no buckets are full, the distribution is uniform, i.e $p_0(n) = (1/m)^n$. We start by considering p_1. When 1 bucket is full, a set of active (not full) buckets are biased. Suppose that an element being inserted has one of its two buckets that is full, then the element will be placed in the second bucket in any case. Since the second bucket's index i_2 has only $2^{\lambda_T} < m$ possibilities, then 2^{λ_T} buckets have a higher chance to receive an element. The same holds when the element's buckets are both full, but in this case another element is evicted to its secondary bucket.
Let B be the set of full buckets and R the set of active (not full) buckets reachable from the set of full buckets:

$$R = \{i_r \in [m] : i_r = i_b \oplus H_T(tag) \text{ for some } i_b \in B \text{ and } tag \in \{0,1\}^{\lambda_T}\}.$$

We compute the probability that a bucket receives an element in the two cases, whether the bucket is biased or not:

$$E_k = [\text{an element is inserted into bucket } w, w \in R, |B| = k],$$
$$E'_k = [\text{an element is inserted into bucket } w, w \notin R, |B| = k].$$

We first consider the case when 1 bucket is full. Thus, the number of reachable buckets R is 2^{λ_T}. We insert an element x and $H_I(x) = i_1$. Then,

$$\begin{aligned}
\Pr[E_1] &= \Pr[E_1 \mid i_1 \in B] \Pr[i_1 \in B] + \Pr[E_1 \mid i_1 \notin B] \Pr[i_1 \notin B] \\
&= \Pr[E_1 \mid i_1 \in B] \cdot \frac{1}{m} + \Pr[E_1 \mid i_1 \notin B] \cdot \frac{m-1}{m} \\
&= \frac{1}{2^{\lambda_T}} \cdot \frac{1}{m} + \frac{1}{m-1} \cdot \frac{m-1}{m} \\
&= \frac{1}{m}(1 + \frac{1}{2^{\lambda_T}}),
\end{aligned}$$

$$\begin{aligned}\Pr[E'_1] &= \Pr[E'_1 \mid i_1 \in B]\Pr[i_1 \in B] + \Pr[E'_1 \mid i_1 \notin B]\Pr[i_1 \notin B] \\ &= 0 \cdot \frac{1}{m} + \frac{1}{m-1} \cdot \frac{m-1}{m} \\ &= \frac{1}{m}\,.\end{aligned}$$

As expected, unbiased buckets still have a uniform probability of $1/m$ while biased buckets have a higher chance to receive an element.
We can generalize to $k > 1$ full buckets. We denote r_k the number of reachable buckets from the k full buckets, i.e $r_k = |R|$. Then,

$$\begin{aligned}\Pr[E_k] &= \Pr[E_k \mid i_1 \in B]\Pr[i_1 \in B] + \Pr[E_k \mid i_1 \notin B]\Pr[i_1 \notin B] \\ &= \frac{1}{r_k} \cdot \frac{k}{m} + \frac{1}{m-k} \cdot \frac{m-k}{m} \\ &= \frac{1}{m}(1 + \frac{k}{r_k}), \\ \Pr[E'_k] &= \frac{1}{m}\,.\end{aligned}$$

We can derive $p_k(n)$, the probability that $n \leq b$ elements are inserted in any bucket w.

$$\begin{aligned}p_k(n) &= \Pr[E_k]^n \Pr[w \in R] + \Pr[E'_k]^n \Pr[w \notin R] \\ &= \Pr[E_k]^n \cdot \frac{r_k}{m-k} + \Pr[E'_k]^n \cdot \frac{m-k-r_k}{m-k} \\ &= \left(\frac{1}{m}(1 + \frac{k}{r_k})\right)^n \cdot \frac{r_k}{m-k} + \left(\frac{1}{m}\right)^n \cdot \frac{m-k-r_k}{m-k}.\end{aligned}$$ □

Lemma A.4 *Let n be a non-negative integer. Let Π be an insertion-only Cuckoo filter with public parameters $pp = (\lambda_I, \lambda_T, b, u)$ and $m = 2^{\lambda_I}$. Let r_k be a random variable with a Binomial distribution $B(m-k, p)$. Let*

$$P_k(n) = \Pr\left[\begin{array}{c}\textit{n elements are inserted when k buckets are full and} \\ \textit{the } (k+1)^{th} \textit{ bucket is filled on the last insertion}\end{array}\right].$$

Then,

$$P_k(n) \leq \left(\frac{1}{m}(1 + \frac{k}{r_k})\right)^n.$$

Proof To give the exact expression of P_k, we have to consider all possible ways of filling j buckets with n_i elements, $i \in [j]$, $j \in [m]$ and $\sum_i n_i = n$. Hence, we can write P_k as a function of p_k:

$$P_k(n) = p_k(n_0) \cdot p_k(n_1) \cdot ...$$

To simplify, we first bound $p_k(n)$ by considering that all buckets are biased:

$$\begin{aligned} p_k(n) &= \left(\frac{1}{m}(1+\frac{k}{r_k})\right)^n \cdot \frac{r_k}{m-k} + \left(\frac{1}{m}\right)^n \cdot \frac{m-k-r_k}{m-k} \\ &\leq \left(\frac{1}{m}(1+\frac{k}{r_k})\right)^n \cdot \frac{r_k}{m-k} + \left(\frac{1}{m}(1+\frac{k}{r_k})\right)^n \cdot \frac{m-k-r_k}{m-k} \\ &= \left(\frac{1}{m}(1+\frac{k}{r_k})\right)^n. \end{aligned}$$

Then, P_k can simply be bounded using the expression above:

$$\begin{aligned} P_k(n) &= p_k(n_0) \cdot p_k(n_1) \cdot ... \\ &\leq \left(\frac{1}{m}(1+\frac{k}{r_k})\right)^{n_0} \cdot \left(\frac{1}{m}(1+\frac{k}{r_k})\right)^{n_1} \cdot ... \\ &= \left(\frac{1}{m}(1+\frac{k}{r_k})\right)^n. \end{aligned}$$

□

Lemma A.5 *Let Π be an insertion-only Cuckoo filter with public parameters $pp = (\lambda_I, \lambda_T, b, u)$ and $m = 2^{\lambda_I}$. Let r_k be the random variable of the number of reachable active buckets from the set of k full buckets. Then r_k follows a Binomial distribution $B(m-k, p)$, where*

$$p = 1 - \left(1 - \frac{2^{\lambda_T}}{m}\right)^k.$$

Proof For every full bucket i_1, the set of secondary indices i_2 is sampled uniformly from the set of all m indices because H_I and H_T are assumed to be perfectly random hash functions. Thus, r_k follows a binomial distribution where r buckets are sampled uniformly out of $m-k$ with probability p. We remove k because we are only considering active buckets. So,

$$\Pr[r_k = r] = \binom{m-k}{r} p^r (1-p)^{m-k-r}.$$

We now derive the probability p that a bucket is selected. This probability p depends on the factor k since the bucket can be sampled by any of the k full buckets, each sampling 2^{λ_T} buckets. Thus,

$$\begin{aligned} p &= \Pr[\text{the bucket is selected by at least one of the } k \text{ full buckets}] \\ &= 1 - \Pr[\text{the bucket is not selected by the } k \text{ full buckets}] \\ &= 1 - \Pr[\text{the bucket is not selected by 1 full bucket}]^k \\ &= 1 - (1 - \Pr[\text{the bucket is selected by 1 full bucket}])^k \\ &= 1 - \left(1 - \frac{2^{\lambda_T}}{m}\right)^k. \end{aligned}$$

We conclude that $r_k \sim Bin(m-k, p)$ where $p = 1 - \left(1 - \frac{2^{\lambda_T}}{m}\right)^k$. □

Lemma A.6 *Let Π be an insertion-only Cuckoo filter with public parameters $pp = (\lambda_I, \lambda_T, b, u)$ and $m = 2^{\lambda_I}$. Let r_k be a random variable with a Binomial distribution $B(m-k, p)$. Then, the probability that, on the insertion of an element, Π evicts u elements consecutively given l buckets are full is bounded by*

$$\frac{l}{m} \cdot \left(\frac{l}{r_l + l}\right)^{u+1}.$$

Proof An eviction happens when a new element is mapped to two full buckets, or when an evicted element is mapped to a secondary ful bucket. Thus,

$$\begin{aligned}
&\Pr[u \text{ elements are evicted on the insertion of } x \mid B_l^n] \\
=&\Pr[\text{the two buckets of } x \text{ are full}, u \text{ elements are evicted} \mid B_l^n] \\
=&\Pr\left[\begin{array}{l} \text{the primary bucket of } x \text{ is full} \\ \text{the secondary bucket of } x \text{ is full} \\ u \text{ secondary buckets are full} \end{array} \middle| \; B_l^n \right] \\
\leq&\frac{l}{m} \cdot \frac{l}{r_l + l} \cdot \left(\frac{l}{r_l + l}\right)^{u} = \frac{l}{m} \cdot \left(\frac{l}{r_l + l}\right)^{u+1}.
\end{aligned}$$

We approximate the probability that a secondary bucket is full by $\frac{l}{r_l + l}$, the proportion of full buckets in the set $R \cup B$. Indeed, we know that the primary bucket is full, i.e in B, so the secondary bucket is in $R \cup B$ (either full or active). The probability to pick a full bucket in $R \cup B$ is therefore $\frac{|B|}{|B \cup R|}$. □

Proof (Theorem A.1) Recall that the disabling probability can be expressed as

$$\sum_{l=2}^{m} \Pr[n^{th} \text{ insertion fills the stash} \mid B_l^{n-1}] \Pr[B_l^{n-1}].$$

First, observe that $k = \lfloor n/b \rfloor$ is the maximal number of full buckets when n elements are inserted in the filter. Therefore, the sum can be bounded by multiplying k times the maximum of the expression inside the sum. The probability to have l full buckets is maximal when $l = 1$ (we could also take $l = 2$). Then,

$$\begin{aligned}
\Pr[1 \text{ bucket is full}] &= \binom{m}{1} \sum_{n_0, n_1} C(n_0, n_1) \cdot P_0(n_0) \\
&= m \cdot \sum_{n_0, n_1} \binom{n}{b-1} \left(\frac{1}{m}(1 + \frac{1}{2^{\lambda_T}})\right)^{n_0} \\
&\leq m \cdot n \cdot n^{b-1} \left(\frac{1}{m}(1 + \frac{1}{2^{\lambda_T}})\right)^{b}.
\end{aligned}$$

The probability that the stash is filled is increasing with the number of full buckets. Thus,

$$\Pr[n^{th} \text{ insertion fills the stash} \mid B_l^{n-1}] \leq \frac{k}{m} \cdot \left(\frac{k}{r_k + k}\right)^{u+1}.$$

Finally, by bringing both bounds together, approximating r_k by its expected value and replacing k with $\lfloor \frac{n}{b} \rfloor$, we obtain:

$$\begin{aligned}
&\frac{n}{b}\frac{k}{m} \cdot \left(\frac{k}{r_k + k}\right)^{u+1} m \cdot n \cdot n^{b-1} \left(\frac{1}{m}(1 + \frac{1}{2^{\lambda_T}})\right)^b \\
\leq & n^{b+2} \left(\frac{n}{rb + n}\right)^{u+1} \left(\frac{1}{m}(1 + \frac{1}{2^{\lambda_T}})\right)^b . \qquad \square
\end{aligned}$$

Bibliography

[1] Mohammad Al-hisnawi and Mahmood Ahmadi. Deep packet inspection using cuckoo filter. In *2017 Annual Conference on New Trends in Information and Communications Technology Applications (NTICT)*, 2017.

[2] David Clayton, Christopher Patton, and Thomas Shrimpton. Probabilistic data structures in adversarial environments. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1317–1334, New York, NY, USA, 2019. Association for Computing Machinery.

[3] Efficient Computing. Cuckoo filter implementation. https://github.com/efficient/cuckoofilter.

[4] Jie Cui, Jing Zhang, Hong Zhong, and Yan Xu. Spacf: A secure privacy-preserving authentication scheme for vanet with cuckoo filter. *IEEE Transactions on Vehicular Technology*, 2017.

[5] David Eppstein. Cuckoo filter: Simplification and analysis. *CoRR*, abs/1604.06067, 2016.

[6] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, 2014.

[7] Mia Filić, Kenneth G. Paterson, Anupama Unnikrishnan, and Fernando Virdia. Adversarial correctness and privacy for probabilistic data structures. Cryptology ePrint Archive, Paper 2022/1186, 2022. https://eprint.iacr.org/2022/1186.

[8] Shahabeddin Geravand and Mahmood Ahmadi. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks*, 2013.

[9] Danie Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1447–1464, USA, 2019. USENIX Association.

[10] Ella Kummer. Counting filters in adversarial settings. Master's thesis, ETH, https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/appliedcrypto/education/theses/Master_Thesis_Kummer_Ella.pdf, 2022.

[11] Minseok Kwon, Pedro Reviriego, and Salvatore Pontarelli. A length-aware cuckoo filter for faster ip lookup. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2016.

[12] James Larisch, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. Crlite: A scalable system for pushing all tls revocations to all browsers. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 539–556, 2017.

[13] Sharafat Mosharraf and Muhammad Abdullah Adnan. Improving lookup and query execution performance in distributed big data systems using cuckoo filter, 09 2021.

[14] Moni Naor and Eylon Yogev. Bloom filters in adversarial environments. Cryptology ePrint Archive, Paper 2015/543, 2015. https://eprint.iacr.org/2015/543.

[15] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[16] Kenneth G. Paterson and Mathilde Raynal. Hyperloglog: Exponentially bad in adversarial settings. Cryptology ePrint Archive, Paper 2021/1139, 2021. https://eprint.iacr.org/2021/1139.

[17] Xiaofeng Shi, Shouqian Shi, Minmei Wang, Jonne Kaunisto, and Chen Qian. On-device iot certificate revocation checking with small memory and low latency. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1118–1134, New York, NY, USA, 2021. Association for Computing Machinery.

[18] Changhua Sun, Bin Liu, and Lei Shi. Efficient and low-cost hardware defense against dns amplification attacks. In *IEEE GLOBECOM 2008 - 2008 IEEE Global Telecommunications Conference*, 2008.

[19] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. Cryptology ePrint Archive, Paper 2022/1455, 2022. https://eprint.iacr.org/2022/1455.