Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Counting filters in adversarial settings

Master Thesis

Ella Kummer

September 11, 2022

Advisors: Prof. Dr. Kenny Paterson, Dr. Anu Unnikrishnan, Mia Filić

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

Abstract

Counting filters are Probabilistic Data Structures (PDS) that support Approximate Membership Queries (AMQ). They use space and time-efficient representations of data in order to respond to membership queries about the data set. Counting filters are often used in environments where adversaries can manipulate the inputs, for example to increase the false positive and false negative rates of the filter. We study the security of Counting filters. First, we investigate attacks on Counting filters. Then, we use an existing simulation-based framework to analyse the correctness of insertion-only Counting filters, and we extend it to analyse the correctness of Counting filters with deletions. We compute the adversary's advantage when Counting filters are secured replacing hash functions with keyed pseudorandom functions in their construction.

Contents

Chapter 1

Introduction

Data sets are often queried in order to answer elementary questions such as whether an items exist in the data set, what are the most frequent items, or what are the number of unique items. In order to perform these queries, deterministic approaches such as hash tables were implemented. However, since data sets are becoming larger and more complex, these traditional approaches become infeasible [23].

As a response, probabilistic data structures (PDS) were introduced. Probabilistic data structures use space and time-efficient representations of data in order to respond to queries about the data set. Nonetheless, these structures do not provide exact responses, introducing non-zero error probabilities.

Some of the most popular probabilistic data structure are Bloom filters, HyperLogLog, or Count-Min Sketch. Bloom filters [6] support membership queries and are used to test whether an element is a member of a set. HyperLogLog [14] estimates the number of distinct items in data sets. Finally, Count-Min Sketch [9] provides an estimate of the counts for items and finds the most frequent items.

Their various applications include packet routing scheme in networks [7], data mining of biological data [5], and detection of heavy hitters for DoS attacks and anomaly exposure [29].

Currently, evaluations of the correctness of PDS are usually performed in a non-adversarial setting. However, these PDS are often used in adversarial settings where the adversary may be able to choose inputs and queries adaptively to influence those bounds. These attacks on PDS can become very harmful for the real systems as they can disrupt or reduce their availability. As an example, considering the previously mentioned case of detection of heavy hitters for DoS attacks, if an attacker is able to influence the PDS as desired, the DoS attacks might not be detected before causing serious damage to the availability of the system.

By consequence, these probabilistic data structures need to be analyzed in adversarial setting. Possible attacks should be investigated and the adversarial scenarios clearly defined.

In this thesis, we focus on a specific probabilistic data structure: the Counting filter, introduced for the first time by Li Fan et al. in [13]. The Counting filter is an extension of the well known Bloom filter but uses counters instead of bits. This allows Counting filters to answer set membership queries, in addition to allow deletions as in Cuckoo filters [12].

The main advantages of Counting filters are that, as in Cuckoo filters, they allow deletions, and as in Bloom filters, they provide answers to queries in $O(1)$ time. However, compared to Bloom filters, since deletions are allowed, false negatives arise and introduce a new error probability. Additionally, more attacks exist against Counting filters compared to Bloom filters due to deletions. Compared to Cuckoo filters, Counting filters are not based on recursive entry "kicking" as the filter approaches its maximum capacity, thus less elements might be able to be inserted in the filter. We notice that previous work were released comparing Bloom filters and Cuckoo filters [12] [25], but never including Counting filters.

In the literature Counting filter usually rely on hash functions. However, we are going to argue that using hash functions is not secure in any reasonable setting. In our proofs, we are going to replace these hash functions with pseudo-random functions (PRFs). A PRF is a keyed function which is deterministic but indistinguishable from a truly random function of the input. We will model the PRFs as truly random functions that the Counting filter oracle has access to. Using a PRF instead of a hash function is the usual proposition to secure the filter [8].

1.1 Related Work

Before probabilistic data structures were introduced, simple data structure were used such as hash tables. In 1993, Lipton et.al. [20] showed that adversaries can degrade the performances of systems relying on hash tables, by adaptively choosing the inputs. Later, Crosby et. al. [10] presented denial of service attacks against hash table implementations.

Regarding PDS, several works were released on Bloom filters under adversarial environments. Gerbet et al. [16] constructed adversary models for Bloom filters, computed the worst-case parameters in adversarial settings, and proposed several countermeasures to mitigate their attacks. Naor and Yogev [22] also considered Bloom filters in the adversarial model, analysed adversarial correctness of Bloom filters in a game-based setting, and proposed a construction of a more robust Bloom filter. From a more practical

point of view, Antikainen et al. [3] presented DoS attacks against broad classes of Bloom-filter-based protocols.

In our work, we will use simulation-based security frameworks. Paterson and Raynal [24] focused on Hyperloglog and introduced attacks and simulation-based definitions to study its correctness under adversarial inputs. In a more general work, Filić et. al. [21] developed simulation-based security definitions for analysing the security of Approximate Membership Queries (AMQ)-PDS. They used their security models to analyse Bloom and Cuckoo filters. We will extend their work in Chapter 4.

Focusing on Counting filters, two main works were released on Counting filters in adversarial setting. Reviriego and Rottenstreich [26] described two attacks against Counting filters, and Clayton et al. [8] provided a provable-security treatment of Bloom filter and Counting filters in adversarial environments. They derived a bound on the adversarial correctness of Counting filters using a game based approach. This approach requires the introduction of a winning condition for the adversary.

1.2 Outline

Our goal is to study Counting filters in adversarial environments using a simulation based approach. Compared to Clayton et al. [8] using a game based approach, using a simulation based approach allows us to remove the requirement of a winning condition for the adversary.

In Chapter 2, we present Counting filters. We explain their syntax and algorithms. We also give concrete examples of Counting filter's applications, and introduce the adversarial scenarios.

In Chapter 3, we investigate the most important attacks on Counting filters. We explain the adversary's goals and how to achieve them in practice. Additionally, we implement a specific attack on Counting filters where the adversary has access to insert, delete, and reveal queries. We show how much harm an adversary can cause in this model. Finally, we discuss the impact of these attacks on the different applications of Counting filters.

In Chapter 4, we derive a bound on the correctness of insertion-only Counting filters under adversarial inputs and queries. In this scenario, the adversary has access to the filter's content, and is allowed to insert elements. However, the adversary is not allowed to delete elements. We use an existing simulation-based security framework for AMQ-PDS and extend it to insertion-only Counting filters.

In Chapter 5, we extend the simulation-based framework from Chapter 4, and we derive a bound on the correctness of Counting filters with deletions under adversarial inputs and queries. In this case the filter cannot

be revealed to the adversary, and the adversary is allowed to insert and also delete elements. The derived correctness bounds allow us to learn how much more damage an adversary can cause to the structure in an adversarial setting.

Finally, we conclude in Chapter 6.

1.3 Preliminaries

In this work, we use the same notation as in [21]. We repeat it for clarity.

We denote by $[m]$, the set $\{1, 2, ..., m\}$ for $m \in \mathbb{Z}_{\geq 1}$. Given a set S, we write $P(S)$ the power set of S, and $\mathsf{P}_{\mathsf{lists}}(S)$ the set of all lists with non-repeated elements form S. We denote as $\{\}$ a key-value store in an algorithm where every index is initialised with the value $\perp$, and we consider $\perp < n, \forall n \in \mathbb{R}$. We write as $a \leftarrow b$ the assignment of value b to the variable a. If the assigned value is output by a randomised algorithm, we use $\leftarrow\$$ instead. We denote by $|S|$, the number of elements in a set S. We denote the identity function over a set S as $Id_S : S \rightarrow S$. We write as $\text{Funcs}[\mathfrak{D}, \mathfrak{R}]$, the set of functions from a set $\mathfrak{D}$ to a second set $\mathfrak{R}$. We denote by $F \leftarrow\$ \text{Funcs}[\mathfrak{D}, \mathfrak{R}]$ a random function F such that $\mathfrak{D} \xrightarrow{\text{F}} \mathfrak{R}$. We write as $x \leftarrow\$ \mathcal{D}$ the sampling of variable x according to a probability distribution $\mathcal{D}$. We denote by $U(S)$, the uniform distribution over a finite set S.

We will use a pseudo-random function (PRF) $F : \mathcal{D} \rightarrow \mathfrak{R}$, with the finite set $\mathcal{D} = \bigcup_{l=0}^{L} \{0,1\}^l$ for some large but finite value of L , and $\mathfrak{R}$ depending on the Counting filter algorithms and public parameters.

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

Figure 1.1: The PRF experiment.

Definition 1.1 [21]. *Consider the PRF experiment in Figure 1.1. We say a pseudorandom function family $R : \mathcal{K} \times \mathcal{D} \rightarrow \mathfrak{R}$ 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 Exp_R^{PRF}, we have:*

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

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

Chapter 2

Counting filters: syntax, algorithms, applications

In this chapter, we introduce Counting filters. We give their syntax, algorithms, and present some applications.

2.1 Syntax and algorithms

A Counting filter is a structure designed to represent a set of elements and support membership queries. A vector of m counters, initially all set to 0, is allocated to insert up to n elements. The $maxValue$ determines the maximum value counters can reach, and therefore the number of bits assigned per counter. Additionally, an independent mapping function $F : \mathfrak{D} \rightarrow \Re \equiv [m]^k$ is assigned to the filter. This function takes as input the element x to be inserted or deleted in the filter, and outputs k integers with range $[1, \ldots, m]$ corresponding to counters positions to be updated in the filter. This mapping function is usually a hash function or a pseudorandom function. In some case, the mapping function producing k outputs cab be substitute by k mapping functions.

We denote by pp the public parameters m, k, and $maxValue$ of Counting filters. Additionally, we denote the state of the Counting filter as $\sigma \in \sum$, where $\sum$ denotes the space of possible states of Π. Let the set of elements that can be inserted into the Counting filter be denoted by $\mathfrak{D}$. We explain the three different type of queries the Counting filter supports: Insert, Delete and Query.

- The setup algorithm $\sigma \leftarrow \mathsf{setup}(pp)$ sets up the initial state of an empty Counting filter with public parameters pp. It always has to be called first in order to initialise the Counting filter.
- The insert algorithm allows the user to insert an input x into the filter.

setup(pp)

```
1 : m, k, maxValue ← pp
2 : σ ← zeros[m]
3 : return σ
```

qry$^F(x, \sigma)$

```
1 : M ← F(x)
2 : for i ∈ M
3 :     if σ[i] = 0
4 :         return ⊥
5 : return ⊤
```

insert$^F(x, \sigma)$

```
1 : M ← F(x)
2 : for i ∈ M
3 :     if σ[i] < maxValue
4 :         σ[i] += 1
5 : return ⊤, σ
```

delete$^F(x, \sigma)$

```
1 : M ← F(x)
2 : // Element needs to be inserted or false positive
3 : if σ[i] > 0, ∀i∈M
4 :     for i∈M
5 :         if σ[i] > 0
6 :             σ[i] -= 1
7 :     return ⊤, σ
8 : else
9 :     // Element not previously inserted
10 :    return ⊥, σ
```

Figure 2.1: Syntax instantiation for Counting filter.

Given an element $x \in \mathfrak{D}$, the insertion algorithm $(b, \sigma') \leftarrow \mathsf{insert}(x, \sigma)$ increments the counters at positions $c_1, \ldots, c_k$, if they are $< maxValue$. Counters $c_1, \ldots, c_k$ correspond to the outputs of the mapping function F applied to element x such that $F(x) = (c_1, \ldots, c_k)$. The algorithm returns the state of the Counting filter σ' after the insertion and a bit $b = \top$ representing whether the insertion succeeds.

- The delete algorithm allows the user to delete an input x from the filter. Given an element $x \in \mathfrak{D}$, the deletion algorithm $(b, \sigma') \leftarrow \mathsf{delete}(x, \sigma)$ decrements, if they all are > 0, the counters at positions $F(x) = c_1, \ldots, c_k$. It returns the state of the Counting filter σ' after the deletion and a bit $b \in \{\top, \bot\}$ indicating if the deletion succeeded or not.

- The qry algorithm performs a membership query. Given an element $x \in \mathfrak{D}$, $b \leftarrow \mathsf{qry}(x, \sigma)$ checks σ's counters at positions $c_1, \ldots, c_k$, with $F(x) = c_1, \ldots, c_k$. If every counter is > 0, x is considered in the filter and the algorithm returns a bit $b = \top$. Otherwise, if any of the counters is equal to 0, x is considered not inserted in the filter and the algorithm returns a bit $b = \bot$.

Definition 2.1 (Counting filter) *Let $m, k, maxValue$ be positive integers. We define an $(m, k, maxValue)$-Counting filter with algorithms defined in Figure 2.1, with $pp = (m, k, maxValue)$, and $F : \mathfrak{D} \to \Re \equiv [m]^k$.*

We have to decide what happens when a counter reaches its maximum value in the filter and we try to update the filter with an element which requires this counter. We consider the following solution.

We allow increment in the insert function of corresponding counters which are not yet at $maxValue$. A first observation going in the direction of choosing are implementations found online [4] [28] [1]. We want to simulate the common management of Counting filters and these implementations implement this solution. Additionally, this issue is addressed in the original paper about Counting filters [13]. The authors advise to increment counters which are $< maxValue$, and keep the counters which are $= maxValue$ at their maximum value. However, they warn that this might lead to a situation allowing false negatives.

2.2 False positive and false negative probability

False positives

An element $a \in A$ is considered to be a false positive when the bits at positions $F(a) = c_1, \ldots, c_k$ are set to non-zero values and a **qry** query on a returns $\top$, while a was never previously inserted in the filter.

This is the result of counters incremented by different insertions and the union of their set covering the result of non-inserted elements. For example, we consider a filter of size $m = 3$ and $k = 2$. We insert element $a \in A$ such that $F(a) = \{1, 2\}$, and element $b \in A$ such that $F(b) = \{2, 3\}$. For an element $c \in A$ such that $F(c) = \{1, 3\}$, it will appear that the element is in the filter while it was never inserted.

We recall, from the literature, the false positive probability of Counting filters.

Lemma 2.2 ***[18].*** *Let Π be an $(m, k, maxValue)$-Counting filter using a random function $F : \mathfrak{D} \to \Re \equiv [m]^k$. We denote by $\mathsf{Pr}_{\Pi,pp}(FP|n)$ the false positive probability of a Counting filter after n elements were inserted. Then,*

$$\mathsf{Pr}_{\Pi,pp}(FP|n) := (1 - [1 - \frac{1}{m}]^{kn})^k \approx (1 - e^{\frac{-kn}{m}})^k. \tag{2.1}$$

False negatives

An element $a \in A$ is considered to be a false negative when there exists at least one counter at positions $F(a) = c_1, \ldots, c_k$ equal to 0 and a **qry** query on a returns $\bot$, while a was previously inserted in the filter and never removed.

False negatives can happen in a Counting filter due to the deletion of false positive elements. It is straightforward to see that if a false positive element is deleted, it affects counters previously incremented for inserted elements. These inserted elements have their counters decremented, and possibly now equal to 0. Consequently, in this last case, the elements now appears not to be inside the filter when queried. For example, we consider a filter of size $m = 3$ and $k = 2$. We insert element $a \in A$ such that $F(a) = \{1, 2\}$, and element $b \in A$ such that $F(b) = \{2, 3\}$. Let's assume there exist an element $c \in A$ such that $F(c) = \{1, 3\}$ and thus qry(c)= $\top$. If element c is deleted then qry(b)=$\perp$ while b was never deleted.

In our work, we will not require the false negative probability, and therefore, we do not need to define a corresponding expression.

2.2.1 Parameter selection

In order to achieve the optimal false positive probability in a non-adversarial environment, parameters can be derived from one another. Fixing m and n, k can be derived as

$$k \;=\; \ln 2 \,(\frac{m}{n}) \approx 0.7 \,(\frac{m}{n}). \tag{2.2}$$

For this choice of k, the false positive probability is then equal to 2^{-k}. Alternatively, given the optimal value for k, n, and $p \in \mathbb{R}$, setting

$$m = -\frac{n \;\ln p}{(\ln 2)^2}$$

keeps the false positive probability below p.

Additionally, we assign to $maxValue$ the value $2^x - 1$, where x is the number of bits to be assigned to each counter such that the probability that a counter reaches its maximum is small in an non-adversarial setting. Usually, four bits are enough when k is chosen optimally [17] [13].

2.3 The non-adversarial setting

2.3.1 F-decomposability

The function-decomposability property states that the input to an AMQ-PDS is always first transformed using some function F before any further processing is applied. We recall the definition from [21].

Definition 2.3 ***[21].*** *Let Π be an insertion-only AMQ-PDS and let $F \leftarrow\!\$\; \mathsf{Funcs}[\mathfrak{D}, \mathfrak{R}]$ with $\mathfrak{R} \subset \mathfrak{D}$ be a random function that Π has oracle access to. Let $Id_{\mathfrak{R}}$ be the identity function over $\mathfrak{R}$. We say that Π is F-decomposable if we can write:*

$$\mathsf{insert}^{F}(x, \sigma; r) = \mathsf{insert}^{Id_{\mathfrak{R}}}(F(x), \sigma; r) \forall x \in \mathfrak{D}, \sigma \in \sum, r \in R,$$

$$\mathsf{delete}^{F}(x,\sigma;r) = \mathsf{delete}^{Id_{\Re}}(F(x),\sigma;r) \forall x \in \mathfrak{D}, \sigma \in \sum, r \in R,$$

$$\mathsf{qry}^{F}(x,\sigma) = \mathsf{qry}^{Id_{\Re}}(F(x),\sigma) \forall x \in \mathfrak{D}, \sigma \in \sum,$$

where $\mathsf{insert}^{Id_{\Re}}$, $\mathsf{delete}^{Id_{\Re}}$, *and* $\mathsf{qry}^{Id_{\Re}}$ *cannot internally evaluate F due to not having oracle access to it and F being truly random.*

Lemma 2.4 *Counting filters with oracle access to a random function F are F-decomposable.*

Proof. In Counting filters, F represents the random function which outputs the indexes of counters in the filter corresponding to the input. F is used on the inputs to the **insert**, **delete** and **qry** algorithms in Figure 2.1. We set $\Re = [m]^k$ and $\Re \subset \mathfrak{D}$. Since $Id_{\Re}(F(x)) = F(x)$ for any $x \in \mathfrak{D}$, this concludes the proof.

2.3.2 The non-adversarially-influenced state

The NAI state represents the Counting filter in honest setting and the n-NAI-gen algorithms emulates the behavior of an honest user. Since Counting filters are F-decomposable, the mapping between the inputs and the filter's counters does not depend on the distribution of the inputs. Additionally, as we consider sets and not multisets, the NAI state can be defined by the insertion of n distinct elements from a uniform distribution.

Below, we give now the definition of a non-adversarially-influenced state.

Definition 2.5 ((n,ϵ)-NAI) ***[21].*** *Let $\epsilon > 0$, and let n be a non-negative integer. Let Π be a Counting filter with public parameters pp and state space $\sum$, such that its **insert** algorithm makes use of oracle access to functions F. Let alg be a randomised algorithm outputting values in $\sum$. Let σ and $\sigma^{(n)}$ be random variables representing respectively the outputs of alg and of the randomised algorithm n-NAI-gen 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 $\sigma^{(n)}$.*

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

1 : $\sigma^{(0)} \leftarrow\$ \mathrm{setup}(pp)$
2 : $[\mathsf{x}_1, ..., \mathsf{x}_n] \leftarrow\$ U(S \in \mathsf{P}_{\mathsf{lists}}(\mathfrak{D}) || S| = n)$
3 : **for** $j = 1, ..., n$
4 : $\quad (b, \sigma^{(j)}) \leftarrow\$ \mathbf{insert}^{\mathsf{F}}(\mathsf{x}_j, \sigma^{(j-1)})$
5 : **return** $\sigma^{(n)}$

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

Then, we give the defintion of the false positive probability for a non-adversarially-influenced state.

Definition 2.6 (NAI false positive probability) ***[21].*** *The NAI false positive probability captures the probability that an honest user would experience a false positive membership query result after inserting n elements into Π. Let Π be a Counting filter with public parameters pp, using function F sampled from distribution D_F to instantiate its functionality. Let n be a non-negative integer. We define the NAI false positive probability after n insertions as*

$$\mathsf{Pr}_{\Pi,pp}[FP|n] := \mathsf{Pr}\left[\begin{array}{c} F \leftarrow\!\!\$\ D_F \\ \sigma \leftarrow\!\!\$\ n - \mathsf{NAI} - gen(pp) \\ x \leftarrow\!\!\$\ U(\mathfrak{D}\backslash\mathcal{V}) \end{array} : \top \leftarrow \mathsf{qry}^F(x,\sigma)\right] \tag{2.3}$$

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

Finally, we give the probability that any counter reaches a specific value in a non-adversarially-influenced state.

Definition 2.7 ***[13].*** *Let σ represent a non-adversarially-influenced state as in Definition 2.5. Let l be the number of current elements inside σ, and let m and k be the public parameters of σ. Then, the probability that any counter is greater or equal to a value j is equal to:*

$$\mathsf{Pr}[max(c) \geq j] \leq m \cdot (\frac{e \cdot l \cdot k}{j \cdot m})^j. \tag{2.4}$$

2.4 Adversarial scenarios

We model the adversaries' access to the Counting filters through oracles. The adversarial scenario is defined by the oracles the adversaries have access to.

Oracle **Reveal**()	Oracle **Insert**(x)	Oracle **Delete**(x)
1 : **return** σ	1 : $(b,\sigma) \leftarrow\!\!\$\ \mathsf{insert}^F(x,\sigma)$	1 : $(b,\sigma) \leftarrow\!\!\$\ \mathsf{delete}^F(x,\sigma)$
Oracle **Qry**(x)	2 : **return** b	2 : **return** b
1 : $b \leftarrow\!\!\$\ \mathsf{qry}^F(x,\sigma)$		
2 : **return** b		

Figure 2.3: Oracle access for adversaries.

We give in Figure 2.3 all oracles available to adversaries in the most general setting.

Reveal reveals the filter's content to A, **Query** answers membership queries, **Insert** inserts an element provided by A into the filter, and **Delete** deletes an element provided by A into the filter. These oracles call the qry, insert and delete functions defined in the Counting filter's syntax in Figure 2.1 in Chapter 2.

We define a Counting filter as public when the adversary has access to the **Reveal** oracle. If the adversary is not allowed the access, we define the Counting filter as private.

2.5 Applications of Counting filters

In this section we present three applications of Counting filters in the real world.

2.5.1 Cache sharing among Web proxies

Caching allows to temporarily store copies of files in cache or temporary storage location for re-access, and has been recognized as one of the most important techniques to reduce bandwidth consumption. To gain the full benefits of caching, proxy caches behind a common bottleneck link should cooperate and share their caches, thus further reducing the traffic through the bottleneck. We call the process "Web cache sharing."

Fan et. al. [13] introduced for the first time counting filters in order to reduce Web traffic and alleviate network bottlenecks sharing caches among Web proxies. Storing summaries of their respective caches as counting filters allows to reduce memory requirement.

Each proxy represents its own cached documents inserting the list of corresponding URLS in a Counting filter. When a document is added into the cache, its corresponding counters are incremented. When it is deleted from the cache, the counters are decremented. Each proxy keeps a summary of the directory of cached documents of each participating proxy. When a cache miss occurs, a proxy first queries these summaries to see if the request might be a cache hit in other proxies. If it appears so, it sends a query messages to those proxies. Otherwise, the proxy sends the request directly to the Web server.

If a request is not a cache hit when the summary indicates so, this is a false positive regarding to the Counting filter and the penalty is a wasted query message. If the request is a cache hit when the summary indicates otherwise, this is a false negative and the penalty is a higher miss ratio. The summaries do not need to be up-to-date and accurate at all times. However, the errors affect the total cache hit ratio or the interproxy traffic, but do not affect the

correctness of the caching scheme. False positives and false negatives are the trade-off of this proposition.

2.5.2 Longest Prefix Matching

Classless Inter-Domain Routing [15] requires Internet routers to perform Internet Protocol (IP) Lookup. For each packet traversing the router, they need to retrieve the corresponding forwarding information, by searching variable-length address prefixes in order to find the longest matching prefix of the IP destination address. Longest Prefix Matching plays a fundamental role in the performance of Internet routers, and is a computationally intensive task which often happens to be the performance bottleneck in high-performance Internet routers.

Dharmapurikar et.al. [11] proposed an algorithm for Longest Prefix Matching using a Counting Bloom filter paired with a Bloom filter. They showed that their technique results in a search engine providing better scalability and performance than current approaches for IP routing lookups using IPv4 BGP tables and for IPv6.

The algorithm works as follows.
The system uses as many Bloom and Counting filters as the length of input addresses, and associates one Bloom and one Counting filter with each unique prefix length.

Counting filters are initialised with the insertion of the associated set of prefixes, and Bloom filters are derived from the Counting filters such that each bit in the Bloom filter is set to 1 if the corresponding counter in the Counting filter is > 0.

For each prefix length, one hash table is constructed. Each table is initialized with the set of corresponding prefixes from the forwarding table, where each hash entry is a [prefix, next hop] pair.

The search for the longest prefix matching probes all the Bloom filters in parallel. The filter associated with length one prefixes is probed using the one-bit prefix of the address, the filter associated with length two prefixes is probed using two-bit prefix of the address, etc. Each filter outputs if the given prefix is a match or no. Then, hash tables are probed going from the longest to the shortest associated prefixes. The search continues until a match is found or the vector is exhausted. If a match is found, routers retrieve the corresponding next hop value from the hash table. In [11], the authors do not mention what happens if no match is found. We consider that in the case that no next hop is found, the packet is forwarded to everyone.

An overall view of the scheme can be found in Figure 2.4.

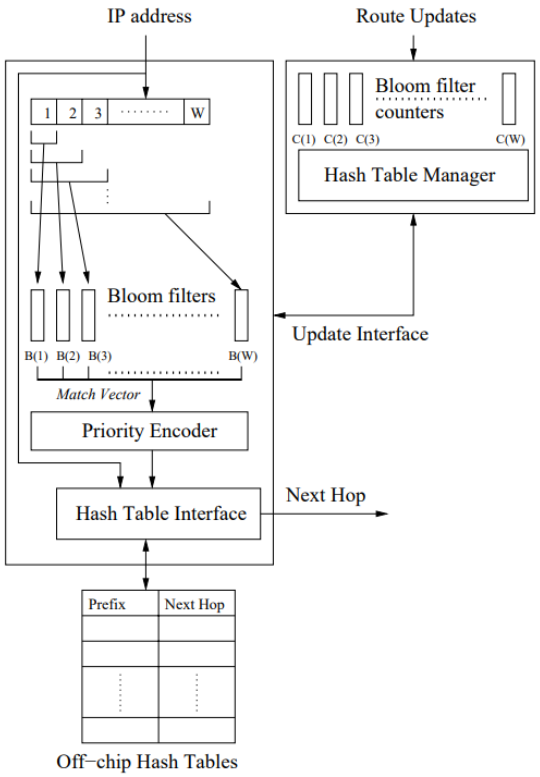

Figure 2.4: General scheme for Longest Prefix Matching [11]

From an architectural point of view, Bloom filters are stored in embedded memory, and the respective Counting filters are maintained by a separate control processor responsible for managing route updates. Updates are distributed to Bloom filters through an update interface.

In terms of scalability, we see that memory resources scale linearly with the number of prefixes in the forwarding table. Regarding performance, the authors state that it can be held constant if the performance by the number of dependent memory accesses per lookup.

2.5.3 Pattern Matching and Anti-Evasion

Intrusion detection systems (IDSs) are devices which analyze all ingoing traffic and detect potentially malicious data in order to protect a network. These deep packet inspection uses standard pattern matching techniques. However, these methods can be evaded by splitting packets into several ones (e.g. TCP and IP fragmentation) or by changing the malicious strings

slightly. In order to detect attacks when evasion happens, IDSs need to reassemble the overall packet flows before applying standard pattern-matching algorithms. This process requires a large amount of memory and time to respond to potential threats.

Antichi et. al [2] introduced an efficient system for anti-evasion that can be implemented in devices. By observing that, due to the continuous creation of new viruses and attacks, the set of their corresponding signatures to be detected changes very frequently, they based their system on Counting filters.

The main idea of the system is to initialize for protocols TCP[1], UDP[2], and ICMP[3], a substring detector where their corresponding traffic is forwarded to.

For each substring detector, a Counting filter is created a priori. Each counting filter, called a "substring counting filter" (subCBF), contains the set of three-byte-long substrings of strings which are parts of valid attacks for a protocol.

Each substring detector processes all its ingoing traffic, by moving along the inspection window of three bytes. Precisely, it applies the hash function to each subgroup of three bytes of the incoming flow, and if all corresponding counter are > 0 in the subCBF, the substring is considered as detected.

When a substring is detected, for each attack this substring belongs to, a new Counting filter called a "striCBF" is initialized. StriCBF are built a priori with insertions of all substrings which are parts of the same attack.

StriCBFs are handled by pattern-matching engines (PMEs) whose goal is to determine if the detected substring is actually a piece of a string and an attack, or a false positive. A membership query to all StriCBFs is performed for each substring from the data flow coming from the substring detector. If there is a match, the corresponding counter in the StriCBFs are decremented. If any of the striCBF is completely reset to zero, the attack is detected and the flow is blocked.

The authors report a detection rate of 99% and false positive rate of 1% or less.

These Counting filters allow rapid inclusion of new virus definitions, without requiring to rebuild the overall structure. We are able to add or remove a new set of attack's signatures by splitting the string into substrings, applying the hash function to these substrings, and incrementing or decrementing corresponding counter in the Counting filter.

[1] https://www.ietf.org/rfc/rfc793.txt
[2] https://www.ietf.org/rfc/rfc768.txt
[3] https://www.rfc-editor.org/rfc/rfc792

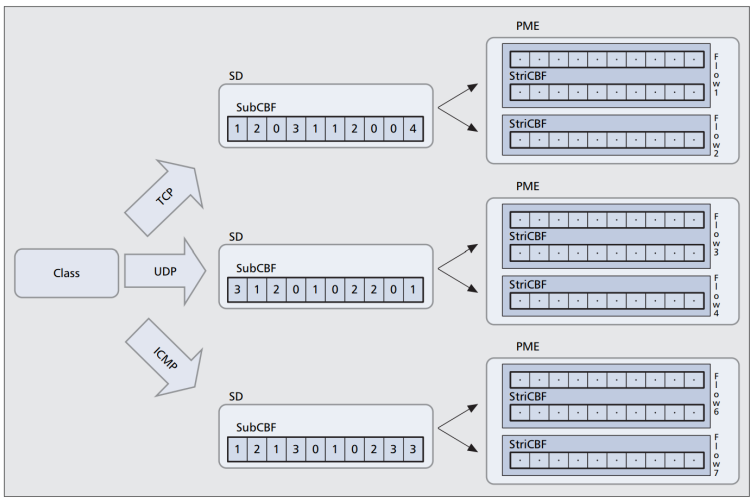

Figure 2.5: General scheme for Pattern Matching and Anti-Evasion [2]

An overall view of the scheme can be found in Figure 2.5.

Chapter 3

Attacks on Counting filters

In this chapter, we describe the most important attacks on Counting filters. We consider their goal and how they can be achieved whether the mapping function is a hash function or a PRF. We define the adversaries' capabilities regarding the adversarial scenarios defined in 2.4. We also implement a pollution attack and discuss the results. Finally, we examine the impact of the attacks on the various applications of Counting Filter.

3.1 Adversary's goals

In this section we cover the main attacks on counting filter and their goals.

Pollution attack. A pollution attack aims at maximizing the number of positions that store a value > 0 in order to maximize the false positive probability of the filter. This usually is done by carefully selecting the elements that are inserted [8].

Reversed pollution attack. A reversed pollution attack aims at minimizing the number of positions that store a value > 0 in order to minimize the false positive probability of the filter, and minimize the number of elements considered inserted inside the filter. This usually is done by carefully selecting the elements that are deleted, the same way elements are selected for a pollution attack.

Target set coverage attack. In a target-set coverage attack, instead of polluting the filter, the adversary wants to build a set R such that when inserted, a non-inserted set T appears to be legitimately inserted in the filter. Specifically, the attacker searches for a cover set $R \in \{0,1\}^*$, such that for a given small target set $T \in \{0,1\}^*$, $qry(x) = \top$ for each $x \in T$. When $|T| < |R|$, such a set exists. However, depending on the size of the target set, the size of the cover set, and the parameters of the filter, finding this set may be computationally infeasible [8].

Reversed target set coverage attack. In a reversed target set coverage attack, the adversary wants to build a set R such that when deleted, a non-deleted set T appears to be legitimately deleted from the filter, i.e. not in the filter. Specifically, the attacker searches for a cover set $R \in \{0,1\}^*$, such that for a given small target set $T \in \{0,1\}^*$, $qry(x) = \perp$ for each $x \in T$, where each $x \in T$ was previously inserted.

Correlation attacks. Correlation attacks compare Counting filters in order to deduce content similarities by looking at the indexes overlaps [27].

Privacy attacks. Filters can leak information such as the approximate total number of elements inserted. For example, if the filter is public, a user can sum up all the counters and divide by the number of hash functions to recover the numbers of inserted elements [27].

We note that in this work, we will only focus on (reversed) pollution attacks and (reversed) target set coverage attacks.

3.2 Attacks when hash function is used

In this section, we show that as long as the mapping function F of the filter in Figure 2.1 is a hash function and the adversary has access to the insert oracle from Figure 2.3, the filter is insecure against (reversed) pollution attacks and (reversed) target set coverage attacks.

The adversary's goal is to maximize the number of positions that store a counter > 0, or target specific counters by carefully selecting the elements that are inserted. Specifically, the adversary wants to update counters $c_1 \ldots, c_k$ for k outputs of the hash function.

Since hash functions are public and deterministic, the attacker can mount a brute-force attack offline and find x such that $\mathrm{F}(x) = c_1 \ldots, c_k$ for targeted counters c_i. This attack can be held offline and the adversary is able to keep a local representation of the filter.

We showed that Counting filters are always unsafe using hash functions. When the adversary has access to the insert oracle, targeted counters are incremented. Therefore, we showed that Counting filters are always unsafe against pollution attacks and target set coverage attacks when using hash functions when adversaries are able to at least insert elements. Additionally, when the adversary is allowed to delete elements in the filter, the filter is also insecure against reversed pollution attacks and reversed target set coverage attacks, where targeted counters are decremented.

3.3 Attacks when PRF is used

In this section, we investigate attacks on Counting filters when the mapping function F in Figure 2.1 is a PRF.

We show that compared to Bloom filters where using a PDF secures the filter [8], this solution alone is not enough for Counting filters.

3.3.1 Scenario 1: Insert and Delete oracles

First attack proposition

The first attack shows how an adversary can mount a pollution attack with access to **Insert** and **Delete** oracles in Figure 2.3.

An adversary derives n different sets S_i, for $i = 1, .., n$, of same cardinality and computes the current false positive probability FP without any new insertion. Then, for each i, the adversary inserts the set S_i, computes the new false positive probability FP_i, and then delete the set S_i from the filter. After computing each FP_i, the adversary inserts the set S_i such that S_i has the biggest corresponding FP_i. By doing this, the adversary chooses the set which increments the maximum counters. The set corresponds to the set S_i such that it shares the least counters with the elements inside the filter.

Second attack proposition

The second attack allows the adversary to derive information about the filter and the mapping function when having access to **Insert** and and **Delete** oracles in Figure 2.3.

This attack is possible when an insertion is blocked by the filter because a corresponding counter is full. We will consider this setting in Chapter 5 when allowing deletions.

The main point of the attack is that the adversary is able to use the output $\perp$ of a blocked insertion as an oracle. First, the adversary tries to insert an element. The element cannot be inserted and the adversary sees that the function insert returns value $\perp$. Then, the adversary deletes elements from a set S, from the filter. After deletions, the adversary tries to insert x for the second time. If the insertion succeeds and returns $\top$, the adversary learns that x shares counters with at least one element from the set S. Using this information, the adversary is able to derive information about the filter and the mapping function.

3.3.2 Scenario 2: Insert, Reveal, and Delete oracles

Clayton et. al. proposed attack

Clayton et al.[8] proposed an attack similar to the target-set coverage attack, even when a PRF is used. This attack is based on the relative power of the **Reveal**, **Insert** and **Delete** oracles depicted in Figure 2.3.

First, the adversary calls the **Reveal** query to get an empty representation. The adversary then inserts an element into the filter and sees exactly what are the outputs of each of the mapping function by looking at the incremented counters. For each insertion, the adversary can delete the previous input. By doing this repeatedly, he can determine the outputs of the PRF for x different inputs using $2x$ queries. These **Reveal** and **Delete** queries effectively provides an oracle for the secretly-keyed PRF. Using this last, the adversary can construct the test and target set used for the target-set coverage attack. The adversary then inserts each element of the test set into the filter.

Second attack proposition

We propose a second attack where the adversary has access to the three oracles shown in Figure 2.3: **Reveal**, **Insert**, and **Delete**.

This attack aims to achieve the same false positive probability than a brute force attack which continuously inserts random elements and never deletes. However, this attacks lowers the number of inserted elements using deletions.

First, the adversary performs a **Reveal** query and learns the current state of the filter. Then, the adversary inserts an element x and performs a second **Reveal** query. This reveal allows the adversary to discover the impact of its newly inserted element x in the filter; precisely he is able to count how many new counter where incremented and are now > 0, compared to the state previous insertion. The adversary decides if he wants to keep the newly inserted element inside the filter or delete it, regarding if he is happy of the impact of the inserted element in the filter or not.

We mention that in practice, these attacks may not always be feasible for the adversary.

3.4 Implementation of pollution attack described in 3.3.2

In this section we implement an attack on a public Counting filter. We show that this filter is not safe against pollution attacks defined in section 3.1, as long as an adversary is allowed reveal, insertions, and deletions queries.

Therefore, in this attack, we consider Scenario 2 from Section 3.3.2 where the adversary has access to the **Reveal**, **Insert** and **Delete** oracles depicted in Figure 2.3.

We implement the second attack from Section 3.3.2. In the case of our implementation, we enforce to have minimum one new counter incremented to keep the element in the filter after insertions and consider the adversary satisfied. This variable cab easily be changed, we can impose the minimum of incremented counters desired. The implementation can be found on Gitlab [19].

Attack parameters

Regarding the technicalities of our implementation, we fix the size of the filter m (:= 10′000), the maximum value per counter (:= 15 $\sim$ 4 *bits* [13]), and the maximum number of insertions n (from 100 to 10′000). Then, we derive k, the number of outputs of our PRF, using equation 2.2.

We use a HMAC with SHA256 as a keyed PRF for the mapping function $F : \mathfrak{D} \rightarrow \Re \equiv [m]^k$ where $\mathfrak{D} = \bigcup_{l=0}^{2^8}\{0,1\}^l$. In order to avoid modulo bias, we sample the k outputs separately and compute the i^{th} function such as $k[i]$ = HMACSHA256(i—x).

For each value of n, we compute 2^6 times the false positive probability in order to obtain an accurate average. The computed false positive probability is obtained by querying to the filter 2^{15} non-inserted elements, and computing $P(FP)$ = (#positive queries)/ 2^{15}.

Results

As a test case, we run the algorithm with no deletions allowed, and therefore, we only insert random elements. In Figure 3.1, we observe that when only random insertion are performed, the computed false positive probability matches almost perfectly the theoretical false positive probability, which validates our implementation.

We observe some jumps in the false positive probability in Figure 3.1. These jumps are due to the changes of value of parameter k displayed in Figure 3.2. Indeed, theoretically k is a rational number, while in practice it becomes an integer. Thus, k must be rounded in the implementation and this rounding impacts the false positive probability, which therefore, does not correspond to the optimal one.

The variance plot in Figure 3.3 shows a variance of 10^{-5} for our false positive computation results, which indicates that these computations are accurate.

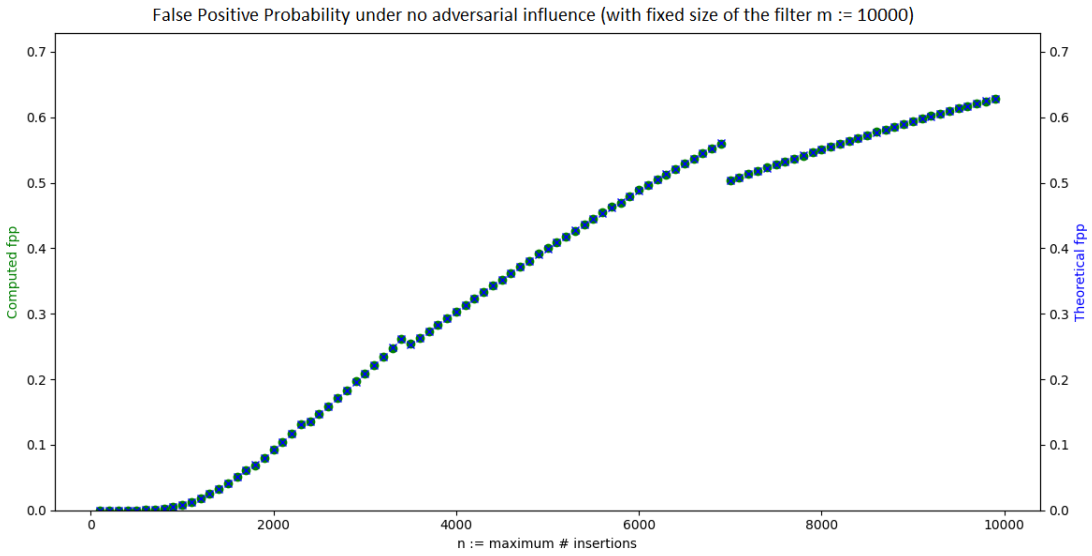

Figure 3.1: Non-adversarial false positive probability

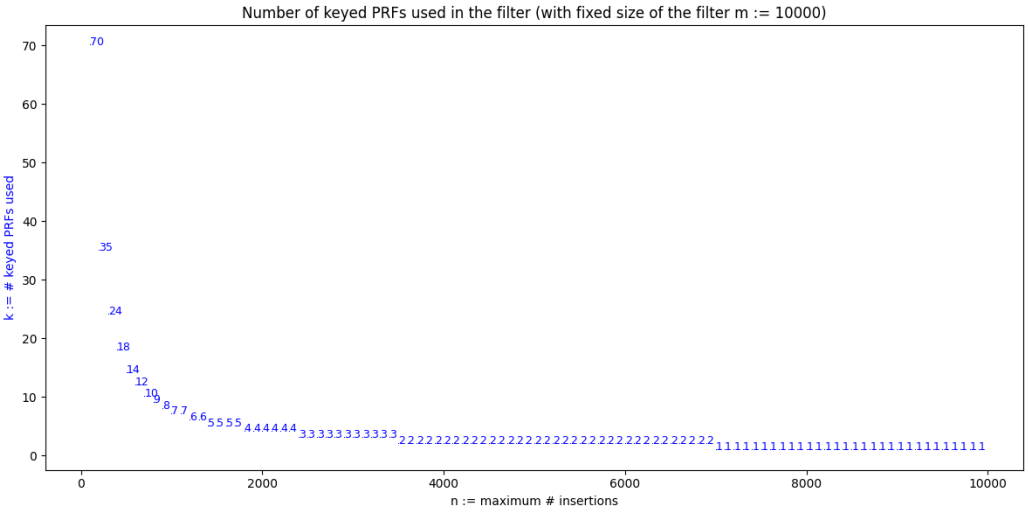

Figure 3.2: Number of hash functions

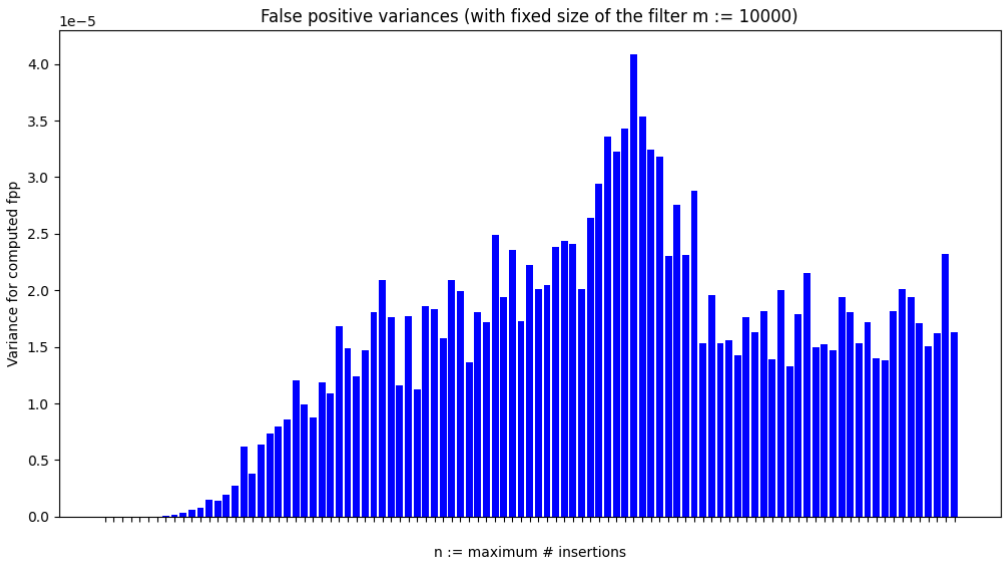

Figure 3.3: False positive results variance without deletions

We focus now on the attack. We observe in Figure 3.4 that we manage to generate a higher false positive rate, and the adversarialFP does not match the theoretical FP anymore. Figure 3.7 plots the accuracy of these results.

We notice that the difference between both false positive probabilities are not constant. For each fixed k value, the theoretical and computed FP are close at the start, and then, the adversarial FP increases faster than the theoretical FP until a new k value is used. Additionally, the gap between both FP increases with the values of n. Indeed, the larger n is, the larger the gap at the jump to a new k value. Figure 3.6 shows the ratios between the adversarially computed FP and the theoretical FP and emphasize that the larger n is, the more impact the adversary has. Regardless of jumps due to changes of k, the adversarial advantage keeps growing linearly.

Finally, we also plot in Figure 3.5 the number of tested elements in order to reach n complete adversarial insertions. We observe that the number of trials increases exponentially. For example, when $n = 100$ we only need 100 trials, while when $n = 5000$ we need 6595 trials, and when $n = 9990$ we need 46002 trials. These results allow to indicate the time and query complexity of the attack.

We conclude that an adversary is able to mount attacks by trial and error when insertions, deletions, and reveals are allowed. By consequence, when hash functions are used, they must be replaced by PRFs, and the filter must be kept secret from the adversary (i.e. no reveal query), at least as long as

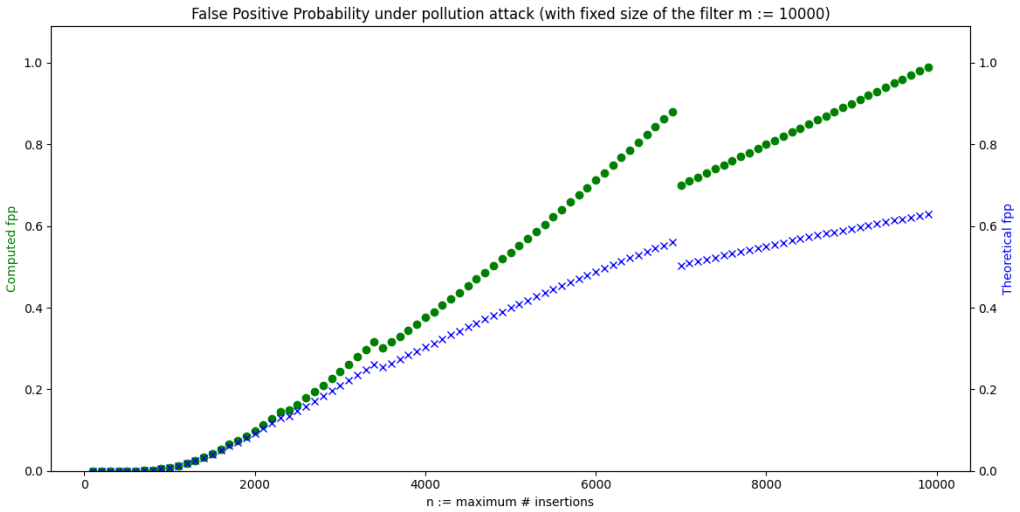

Figure 3.4: False positive probability under adversarial power

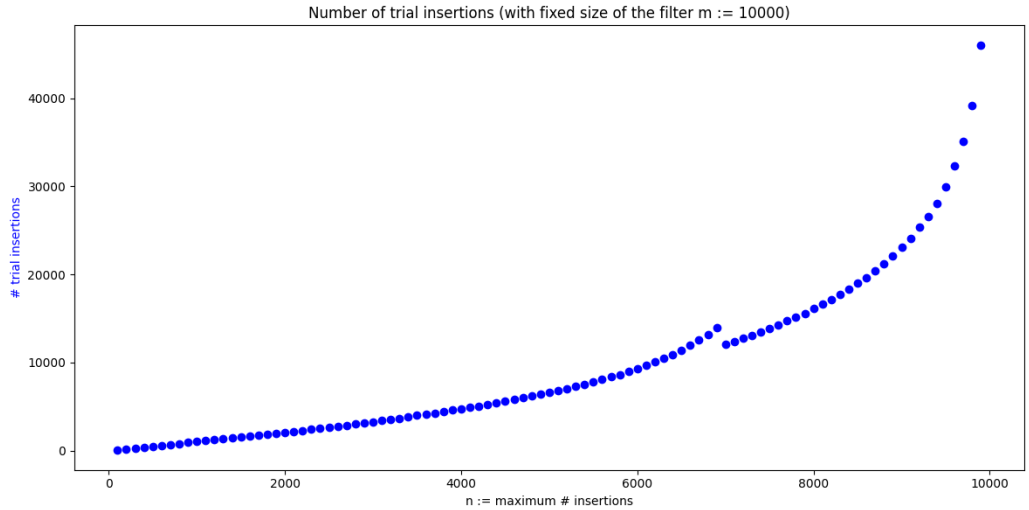

Figure 3.5: Number of trial insertions with deletions

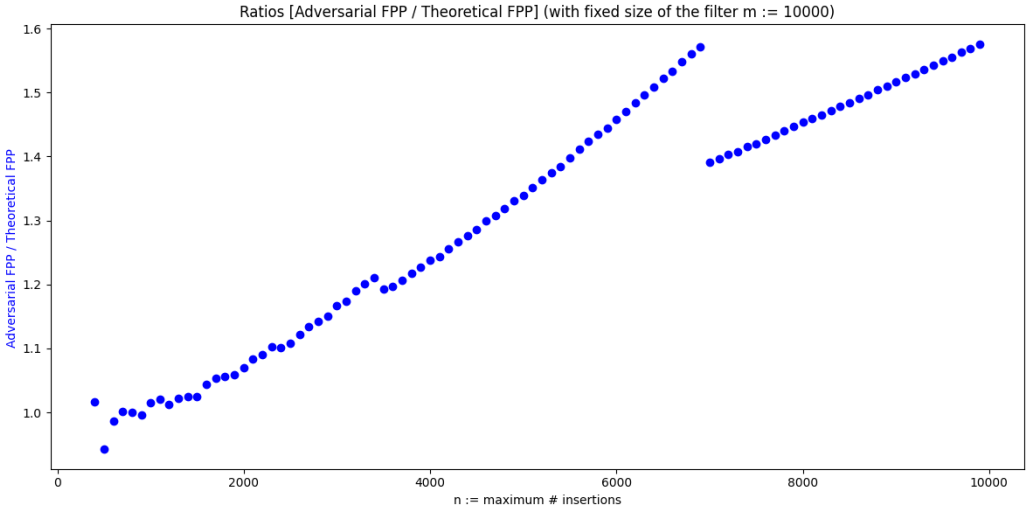

Figure 3.6: Ratios [adversarial/theoretical] false positive probability

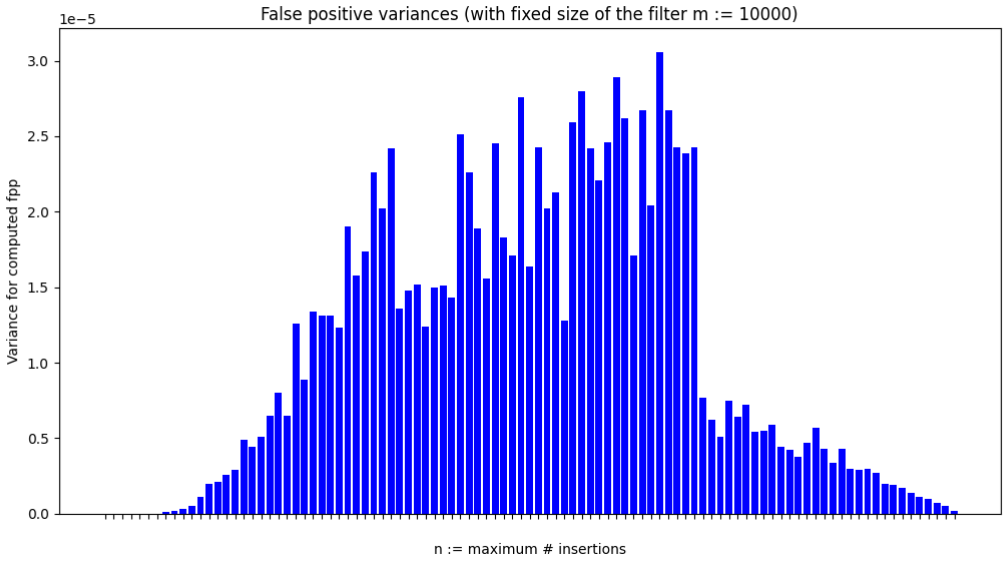

Figure 3.7: False positive results variance with deletions

deletions are allowed.

3.5 Impact of attacks on applications

In this section, we discuss the impact of (reversed) pollution attacks and (reversed) target set coverage attacks on the described applications of Counting filters in Section 2.5.

However, we mention that these attacks may not always be possible to be carried out, depending on scenarios and mapping functions.

3.5.1 Cache sharing among web proxies

In section 2.5.1, we described cache sharing among web proxies using Counting filters. We consider attacks on a Counting filter where its proxy is malicious, or under the control of an attacker. This Counting filter will be sent to other proxies.

In [13], the authors did not specify how the Counting filters are managed by their corresponding proxies. We assume that the proxies can only interact with their own Counting filter through an interface which allows them to have access to **Insert**, **Delete**, **Reveal** oracles, and with an additional **Reveal** oracle to reveal other proxies filters. This corresponds to Scenario 2 described in Section 3.3.2. Additionally, we assume that each proxy is only able to send this specific Counting filter to other proxies. Otherwise, trivial attacks would be allowed, such as sending an empty filter, or sending a filter where counters were incremented and decremented directly by the proxy.

Pollution attack. If, an attacker in control of a proxy can mount a pollution attack, he increases the false positive probability rate of his Counting filter. This filter is sent to proxies to check if their own requests might be a cache hit in this proxy, and due to the pollution attack, the rate of potential cache hit increases. If a request is not a cache hit when the summary indicates so (i.e. a false positive), the penalty is a wasted query message. Proxies' query messages can overload the network, and reduce the availability of the systems. This can prevent honest proxies from accessing cached documents. Additionally, each unnecessary hit between proxies costs bandwidth and adds latency to the client's request.

Target set coverage attack. If the attacker can predict in advance which documents are going to be requested, the adversary can mount a target set coverage attack in order to ensure that the requested documents become false positives. However, in the case where it is hard to predict which documents might be accessed, there is no point for an adversary to forge a target set coverage attack and a pollution attack might be more efficient.

We note that these last two attacks could also be carried out in Scenario 1 described in Section 3.3.1, with no **Reveal** oracle.

Reversed pollution attack. If the adversary is able to mount a reversed pollution attack, the attacker is able to produce false negatives. If the request is a cache hit when the summary indicates otherwise (i.e. a false negative), the penalty is a higher miss ratio. This will force the proxy to send a request directly to the Web server, which is more expensive than contacting a proxy with the cached document. Additionally, since the false negative rate is higher due to the attack, more proxies will try to contact the web server. This will generate more latency and the web server availability might be compromised.

Reversed target set coverage attack. If the attacker can predict in advance which documents are going to be requested, the adversary can mount a reversed target set coverage attack in order to ensure that the requested documents become false negatives. As in the reversed pollution attack, this will force the proxy to send a request directly to the Web server, which is more expensive than contacting a proxy with the cached document.

We note that the adversary has a restricted impact as he cannot use a false positive more than once since after each unsuccessful cache access, the proxy contacts the web server and the web page is added to the cache. Similarly with false negatives, after contacting the web server, the proxy adds the web page to its cache.

We note that the adversary has a restricted impact. Indeed, a false positive has a one-time effect since after each unsuccessful cache access, the proxy contacts the web server and the web page is added to the cache. Similarly with false negatives, after contacting the web server, the proxy adds the web page to its cache.

3.5.2 Longest Prefix Matching

In section 2.5.2, we described the use of Counting filters for the search of longest prefix matching in routers. We consider as adversarial, a malicious router with has access to its Counting filter through an interface with access to **Insert**, **Delete**, **Reveal** oracles. This corresponds to Scenario 2 described in Section 3.3.2.

We consider the hash-tables storing the next hops to be non-modifiable as they are stored off-chip and are not part of our interest. This reduces the amount of possible attacks as this ensures that no route can be added to hijack traffic.

Pollution attack. As explained in section 2.5.2, after finding a prefix match in the Bloom filter derived from the Counting filter, the hash tables are probed

from the longest to shortest prefix in order to find a match with the next hop information. A pollution attack increases the false positive probability and might create longer prefix matches which are false positives. This will cause more hash tables lookup and potentially more latency. However, since all hash tables are probed, the next hop corresponding to the shorter legitimate prefix will still be found.

Target set coverage attack. If the attacker can predict in advance which prefixes are going to be received by the router for matching, the adversary can mount a target set coverage attack. A target set coverage attack ensures that longer prefixes than the ones registered in hash tables are false positives, and therefore, are matches for the specific received prefixes. As for the pollution attack, this will ensure more hash tables lookup and therefore, more latency.

We note that these last two attacks could also be carried out in Scenario 1 described in Section 3.3.1, with no **Reveal** oracle.

Reversed pollution attack. If the attacker can mount a reversed pollution attack where he deletes non-inserted elements from the Counting filter, the false negative rate increases.

Due to the higher false negative rate, when receiving a packet, all the filters might return no prefix match. As previously argued in Section 2.5.2, if there happens to be a match for a prefix in the filter but the prefix is not present in the table, the packet is forwarded to everyone.

Otherwise, the longest prefix match might become a false negative, but a shorter prefix might remain a match. In this case, the packet is forwarded to more people.

In both of these situations, the adversary might be able to overload and add latency in the network due to the increase in packets distributed in the network.

Reversed target set coverage attack. An attacker can mount a reversed target set coverage attack in order to receive a packet intended for a receiver with whom he shares a common sub-prefix.

The attacker can use this attack to delete elements such that then, the longer prefix which does not cover the attacker's address becomes a false negative. Then, the intended receiver still receives the packet, as well as the attacker now.

Finally, we note that it is impossible for an attacker to cut access to a resource. The attacker is only able to ensure that more receivers than intended receive the packet. In the worst case, the packet is sent to everyone.

3.5.3 Pattern Matching and Anti-Evasion

In section 2.5.3, we described a system used for pattern matching and anti-evasion which uses two levels of counting filters. First, it uses the subCBF to detect a first potential packet from an adversarial flow, and then it uses striCBF to ensure that the packet is part of an attack. The first Counting filter (subCBF) needs one match, while the second one (striCBF) needs to have all inserted elements matched (i.e. all counters decreased to 0).

We consider an adversary which has access to Counting filters through an interface with access to **Insert**, **Delete**, **Reveal** oracles. This corresponds to Scenario 2 described in Section 3.3.2.

Attacks on subCBF.

Pollution attack. If a pollution attack happens on the subCBF, the false positive rate of the subCBF increases. Therefore, more genuine packets might be suspected due to the higher rate of false positives. Thus, this increase the chance of suspecting a legitimate flow.

An attacker can also use this higher false positives rate in order to perform a denial of service attack. This is possible by identifying and sending a large number of packets (from different flows) which are false positives in the subCBF. These packets will trigger the initialization of striCBFs. Then, more packets will be forwarded to the pattern-matching engines and queried to striCBF, which might reduce the availability of the systems and degrade their performance.

Target set coverage attack. If the adversary can predict which packets are coming, he can mount a target set coverage attack in order to ensure that these packets are false positives in the filter and therefore, become suspects. This increases the chance to mark the legitimate flow these packets belong to, as a detected attack.

We note that these attacks could also be carried out in Scenario 1 described in Section 3.3.1, with no deletions.

Reversed pollution attack. If an adversary is able to mount a reversed pollution attack, the false negative rate is increased. Therefore, the Intrusion detection systems might not recognize some attacks, as their corresponding packets will not be matches in the subCBFs, and by consequence never trigger second level filters.

Reversed target set attack. If an adversary knows which packets belong to attacks, mounting a reversed target set attack allows him to ensure that these packets are false negatives in the filter, and therefore the flow these packets belong to, will remain undetected.

Attacks on striCBF.

Pollution attack. If the adversary manages to increment new counters, the packets from the attacker's flow might never match these and thus the flow will be undetected.

We note that this attack could also be carried out in Scenario 1 described in Section 3.3.1, with no deletions.

Reversed pollution attack. If the adversary manages to mount a reversed pollution attack, this might higher the false negative rate. Thus, less packets' matches will be expected in the striCBF, and the IDS might detect some flows which are not attacks.

Reversed target set coverage attack. The adversary can achieve the same goal as in the reversed pollution attack, but with more precision. With a reversed target set coverage attack, the adversary can ensure false negatives for packets he knows are not part of the flow currently under supervision. Therefore, these packets are not expected anymore in order to mark the flow as malicious.

Attacks on on subCBF and striCBF together.

The adversary has more power and can combine the attacks mentioned above.

First, the adversary can perform a pollution attack or a target set coverage on subCBFs, the IDS might detect non-adversarial packets and initialize striCBFs. Then, the attacker can mount a reversed pollution attack or a reversed target set coverage attack, in order to delete elements in striCBFs. This might produce more false negatives and detect flows which are not attacks. Therefore, combining these steps, the adversary has a higher chance to mark fair flows as malicious while no attack is happening, and by consequence discards trustworthy packets.

On the other hand, the adversary can mount a reverse pollution attack or a reversed target set coverage attack in the subCBFs in order to delete elements. Then, the adversary can mount a pollution attack on the striCBFs. This will increase the probability that adversarial flows remain undetected.

We observe that this application is slightly different from the two previous ones. In our current example, the filter is only locally used in order to protect the network, and does not communicate with or impact external entities. Therefore, if the filters are only accessed by entities which are part of the network, these entities have no reason to attack their own network. In order for the aforementioned attacks to make sense, the filters need to be able to be accessed by external entities.

Chapter 4

Security analysis of insertion-only Counting filters

Insertion-only Counting filters are Counting filters where elements can only be inserted and queried. In this Chapter, we analyse the adversarial correctness of insertion-only Counting filters using a simulation-based approach. Filić et al. [21] developed a simulation-based approach to derive security bounds of insertion-only AMQ-PDS. We will apply this existing framework to our insertion-only Counting filter to demonstrate the limits of any adversaries'abilities.

In order to do this, we modify the Counting filter syntax defined in Figure 2.1. We get rid of the $\mathsf{delete}^{\mathsf{F}}$ algorithm, and modify the $\mathsf{insert}^{\mathsf{F}}$ algorithm in Figure 4.1.

This new insert algorithm enforces that the filter does not change if an element is considered already in the filter.

We motivate our modification in the insert algorithm. First, all of our use-cases for Counting filters described in section 2.5 do not need reinsertion of the same elements multiple times as they use Counting filter for membership queries and deletions. In different applications of Counting filters, they do not appear to be used to count and insert the same elements multiple times. Furthermore, not changing the filter when reinsertion allows us to use the framework in [21] in Chapter 4 since the NAI definition 2.3 is similar. Indeed, because we do not allow reinsertion, the non-adversarially-influenced (NAI) state which represents the honest setting is simply defined by the insertion of n different elements from a uniform distribution. Finally, the false positive probability is always derived regarding the number of element inserted with uniformly random mappings [18]. The false positive probability is computed with the assumption that after all n elements are inserted and all k mapping functions applied to each of those n elements,

setup(pp)

1 : $m, k, maxValue \leftarrow pp$
2 : $\sigma \leftarrow \text{zeros}[m]$
3 : **return** σ

qry$^F(x, \sigma)$

1 : $M \leftarrow \mathsf{F}(\mathsf{x})$
2 : **for** $i \in M$
3 : **if** $\sigma[i] = 0$
4 : **return** $\perp$
5 : **return** $\top$

insert$^F(x, \sigma)$

1 : $M \leftarrow \mathsf{F}(\mathsf{x})$
2 : **if** $\sigma[i] > 0$, $\forall i \in M$
3 : // Element already considered inserted
4 : **return** $\top, \sigma$
5 : **else**
6 : **for** $i \in M$
7 : **if** $\sigma[i] < maxValue$
8 : $\sigma[i] += 1$
9 : **return** $\top, \sigma$

Figure 4.1: Syntax instantiation for insertion-only Counting filter.

the probability that an arbitrary bit in the Counting filter is 0 is $\mathrm{P}[c_i = 0] = (1 - \frac{1}{m})^{kn}$.

This new condition does not influence the false positive probability since this condition affects only counters already > 0. Enforcing this condition directly in the Counting filter's insert algorithm, allows us to define a bound on any adversary against the given Counting filter.

4.1 Consistency rules

In this section we give the consistency rules, defined in [21], for our insertion-only Counting filter Π.

Element permanence

If for all $x \in \mathfrak{D}$, $\sigma \in \sum$ such that $\top \leftarrow qry(x, \sigma)$, and for any sequence of insertions resulting in a later state σ',

$$b \leftarrow qry(x, \sigma') \Rightarrow b = \top.$$

Reinsertion invariance

If for all $x \in \mathfrak{D}$, $\sigma \in \sum$ such that $\top \leftarrow$ insert(x,σ),

$$(b, \sigma') \leftarrow insert(x, \sigma; r) \Rightarrow \sigma = \sigma' \ \forall r \in \mathcal{R}.$$

Permanent disabling

If given $\sigma \in \sum$ such that there exists $x \in \mathfrak{D}$, $r \in \mathcal{R}$ where $(b, \hat{\sigma}) \leftarrow$ insert(x,σ;r) and $b = \perp$, then $\hat{\sigma} = \sigma$ and for any $x' \in \mathfrak{D}$, $r' \in \mathcal{R}$,

$$(b', \sigma') \leftarrow insert(x', \sigma; r') \Rightarrow b' = \perp \text{ and } \sigma' = \sigma.$$

Non-decreasing membership probability

If for all $\sigma \in \sum$, $x, y \in \mathfrak{D}$, $r \in \mathcal{R}$,

$$(b, \sigma') \leftarrow insert(x, \sigma; r) \Rightarrow Pr[\top \leftarrow qry(y, \sigma)] \leq Pr[\top \leftarrow qry(y, \sigma')].$$

4.2 Adversarial Correctness of insertion-only Counting filters

In this section, we introduce the simulation-based framework which we then use in order to derive bounds on the correctness of insertion-only Counting filters.

Settings. Our model considers an adversary $\mathcal{A}$ interacting with a Counting filter Π initialised empty. The adversary is given access to three oracles (see Figure 5.2). **Qry** takes as input x, an element provided by $\mathcal{A}$, and outputs if x has been inserted into Π. **Insert** inserts an element provided by $\mathcal{A}$ into Π. Finally, **Reveal** returns the current state of Π and reveals the content of the filter to the user.

The simulation-based framework introduces two worlds, a *Real* world, and *Ideal* world. In the real world, the adversary is given access to an AMQ-PDS instantiation where he is allowed to insert elements, and make membership queries, while the in the ideal world, the adversary interacts with a simulator S. The simulator S provides a non-adversarially-influenced (NAI) view of the insertion-only AMQ-PDS, as in Figure 2.2. The idea behind the simulator is that whatever the adversary can learn from interacting with the AMQ-PDS in the real world can be simulated.

The adversary plays in either the real or ideal world, and is allowed to make **Insert**, **Query**, and **Reveal** queries. At the end of the execution in which the adversary plays either in the real or ideal world, the adversary produces some output. This output is then forwarded to a distinguisher D.

The distinguisher D then utilizes the adversary's output in order to compute which world the adversary was operating in.

First, we define the Real-or-Ideal game in Figure 5.2. We denote the real (d = 0) and ideal (d = 1) versions of Real-or-Ideal as Real and Ideal, respectively. The distinguisher D generates the game's output $d' \in \{0,1\}$.

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

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

Oracle **Insert**(x)

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

Oracle **Qry**(x)

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

Oracle **Reveal**()

1 : **return** σ

Figure 4.2: Correctness game for the insertion-only Counting filter Π using the simulation paradigm.

We want to assess the advantage of an adversary in the real world compared to the ideal world. As Filić et. al. [21], we use the advantage of a distinguisher in the Real-or-Ideal game to define adversarial correctness.

Definition 4.1 (Adversarial correctness) *[21]. Let Π be an insertion-only Counting filter, with public parameters pp, and let R_K be a keyed function family from $\mathfrak{D}$ to $\mathfrak{R}$. We say Π is $(q_{in}, q_{qry}, q_{rvl}, t_a, t_s, t_d, \epsilon)$-* ***adversarially correct*** *if, for all adversaries $\mathcal{A}$ running in time at most ta and making at most q_{in}, q_{qry}, q_{rvl} queries to oracles* ***Insert***, ***Qry***, ***Reveal*** *respectively in the Real-or-Ideal game (Figure 5.2) with a simulator $\mathcal{S}$ that provides an NAI view of Π to $\mathcal{A}$ and runs in time at most t_s, and for all distinguishers D running in time at most t_d, we have:*

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

Relying on the consistency rules (Section 4.1) and the F-decomposability (Definition 2.3.1), we are able to use the simulator from the existing framework.

We give in Figure 4.3 a simulator $\mathcal{S}$ that replicates the behaviour of an insertion-only Counting filter and satisfies the consistency rules (Section 4.1) and is by definition function-decomposable (section 2.3.1).

We observe that our simulator is equivalent to the simulator described in [21]. Indeed, the **Up** algorithm is simply renamed **Insert**, and we discard the **Rep** algorithm.

Next, we state Theorem 1 from [21], which we show can be applied to Counting filters.

Theorem 4.2 ***[21].*** *Let q_{in}, q_{qry}, q_{rvl} be non-negative integers, and let t_a, $t_d > 0$. Let $F : \mathfrak{D} \to \mathfrak{R} \equiv [m]^k$. Let Π be an insertion-only Counting filter with public parameters pp and oracle access to F, such that Π satisfies the consistency rules from section 4.1 and F-decomposability (Definition 2.3.1). Let α (resp. β) be the number of calls to F required to insert (resp. query) an element in Π using its insert (resp. qry) algorithm.*
If $R_k : \mathfrak{D} \to \mathfrak{R}$ is an $(\alpha q_{in} + \beta q_{qry}, t_a + t_d, \epsilon)$-secure pseudorandom function with key $K \leftarrow\$ K$, then Π is $(q_{in}, q_{qry}, q_{rvl}, t_a, t_s, t_d, \epsilon')$-adversarially correct with respect to the simulator in Figure 4.3, where $\epsilon' = \epsilon + 2q_{qry} \cdot \mathsf{Pr}_{\Pi,pp}(FP|q_{in})$ and $t_s \approx t_a$.

We show that we can apply this Theorem to our insertion-only Counting filters defined in Figure 4.1.

Lemma 4.3 *Insertion-only Counting filters with public parameters pp and oracle access to F satisfy the consistency rules from Section 4.1 and F-decomposability from Definition 2.3.1.*

Proof. We showed in Chapter 2 section 2.3.1 that Counting filters satisfy F-decomposability. We prove now that Counting filters satisfy the consistency rules in Definition 4.1.

Element permanence. As we consider insertion-only Counting filter, deletions are not allowed and once an element is inserted inside the filter, the element remains there indefinitely. Additionally, if an element is a false positive, since deletions are not allowed, no counters are decremented, and the element also remains a false positive indefinitely.

Reinsertion invariance. The syntax in Figure 4.1 shows that by definition the filter will not be affected by reinsertion and therefore, this rule is satisfied.

Permanent disabling. In our Counting filter syntax, we observe that the filter is never disabled. Consequently, this property is trivially satisfied.

Non-decreasing membership probability. The more insert queries we perform, the higher the chance that the element was inserted. Additionally, looking at the false positive probability in equation 2.1, it is clear that the more elements in the filter, the bigger or equal the false positive probability.
□

We proved that our insertion-only Counting filters satisfy the consistency rules and F-decomposability. Therefore we can apply Theorem 4.2.

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

$F \leftarrow\$ \; Funcs[\mathfrak{D}, \mathfrak{R}]$
$\sigma \leftarrow \text{setup}(pp)$
inserted, FPlist, CALQ $\leftarrow$ {}, {}, {}
i $\leftarrow 0$ // Qry counter
$ctr \leftarrow 0$ // Insertions counter
return $out \leftarrow\$ \; A^{\textbf{InsertSim, QrySim, RevealSim}}$

Oracle **QrySim**(V)

$i \leftarrow i + 1$
// Element was inserted or determined a **false** positive
if inserted$[x] = \top$or FPlist$[x] = \top$
  return $\top$
// Element was not inserted and not **false** positive
if CALQ$[x] \leftarrow ctr$
  // If no changes/insertions since last query x
  return $\perp$
// Response needs **to** be (re)computed
CALQ$[x] \leftarrow ctr$
$\mathsf{a}_\mathsf{i}^{\mathsf{ideal}} \leftarrow\$ \; \mathsf{qry}^{\mathsf{Id}_\mathfrak{R}}(Y \leftarrow\$ \; \mathfrak{R}, \sigma)$
$\mathsf{a}_\mathsf{i}^{\mathsf{G}*} \leftarrow\$ \; \mathsf{qry}^{\mathsf{F}}(x, \sigma)$
$a \leftarrow \mathsf{a}_\mathsf{i}^{\mathsf{Ideal}}\boxed{\mathsf{a} \leftarrow \mathsf{a}_\mathsf{i}^{\mathsf{G}*}}$ // Ideal G^*
if $a = \top$
  FPlist$[x] = \top$
return a

Oracle **InsertSim**(x)

if inserted$[x] = \perp$
  $(b, \sigma) \leftarrow \mathsf{insert}^{\mathsf{F}}(\mathsf{x}, \sigma)$
  if $b = \top$
    inserted[x] $\leftarrow \top$
    $ctr+ = 1$
  return b
else
  return $\top$

Oracle **RevealSim**()

return σ

Figure 4.3: Simulator used in Theorem 1.

Corollary 4.4 *Let Π be our Counting filter from Lemma 4.3. Let q_{in}, q_{qry}, q_{rvl} be queries to oracles Insert, Qry, Reveal respectively, and let t_a, $t_d > 0$. Let $F : \mathfrak{D} \to \mathfrak{R} \equiv [m]^k$.*

If $R_k : \mathfrak{D} \to \mathfrak{R}$ is an $(q_{in} + q_{qry}, t_a + t_d, \epsilon)$-secure pseudorandom function with key $K \leftarrow\$\ K$ and $F = R_K$, then Π is $(q_{in}, q_{qry}, q_{rvl}, t_a, t_s, t_d, \epsilon')$-adversarially correct, where $\epsilon' = \epsilon + 2q_{qry} \cdot \mathsf{Pr}_{\Pi,pp}(FP|q_{in})$ and $t_s \approx t_a$:

$$\mathsf{Adv}^{\Re o \Im}_{\Pi,\mathcal{A},\mathcal{S}}(\mathcal{D}) := |\mathsf{Pr}[Real(\mathcal{A},\mathcal{D}) = 1] - \mathsf{Pr}[Ideal(\mathcal{A},\mathcal{D},\mathcal{S}) = 1]| \leq \epsilon + 2q_{qry} \cdot Pr_{\Pi,pp}(FP|n + q_{in}).$$

Proof. From the instantiation of Counting filters given in Figure 4.1, we observe that each *insert* and *qry* call contains one call to the function F. Then, using Lemma 4.3, Theorem 4.2 holds with $\alpha = \beta = 1$. □

We note that our bound is equal to the bound derived in [21] for Bloom filters. This is coherent as Bloom filters correspond to insertion-only Counting filters with the counter's $maxValue$ set to 1.

Chapter 5

Security analysis of Counting filters with deletions

In this Chapter, we analyse the adversarial correctness of Counting filters with deletions, using a simulation-based approach. We extend the simulation-based framework from Chapter 4.

To begin with, we modify the Counting filter syntax defined in Figure 2.1. We redefine the insert algorithm in Figure 5.1.

We restrict now the insert algorithm such that when a counter reaches its maximum value, the filter does not allow insertion of an element which needs this counter. In the general syntax for Counting filters given in Chapter 2 and for insertion-only Counting filter in Chapter 4, we increment the counters which have not reached yet $maxValue$. However, this method has an important drawback when deletions are allowed: false negatives can arise with just deletions of inserted elements. Indeed, some counters are shared between different inserted elements, and some counters are equal to value $maxValue$ while being actually used by $maxValue + j$ elements with $j \in \mathbb{N}$. Afterward, with deletions of those inserted elements, we decrement the counters until possibly reaching value 0, while still j elements need this counter to be > 0 to be considered in the filter. By consequence, we opt for a solution where the element is not inserted at all in the filter and all its corresponding counters are not incremented. The drawback of this solution is that the filter allows the insertion of less elements. However, the probability that a counter reaches its maximum value, under random insertions, is small, and consequently the number of insertions rejected is small too [13].

Compared to insertion-only Counting filters where the restriction on reinsertion is part of the syntax, for Counting filters with deletions, we put the restriction on the adversary and assume that he never performs reinsertion of inserted elements. Additionally, in order to be the closest possible to the

setup(pp)

```
1 : m, k, maxValue ← pp
2 : σ ← zeros[m]
3 : return σ
```

qry$^F(x, \sigma)$

```
1 : M ← F(x)
2 : for i ∈ M
3 :     if σ[i] = 0
4 :         return ⊥
5 : return ⊤
```

insert$^F(x, \sigma)$

```
1 : M ← F(x)
2 : // No corresponding counter must be full
3 : if σ[i] < maxValue, ∀i∈M
4 :     for i∈M
5 :         σ[i]+ = 1
6 :     return ⊤, σ
7 : // At least one corresponding counter is full
8 : else
9 :     return ⊥, σ
```

delete$^F(x, \sigma)$

```
1 : M ← F(x)
2 : // Element needs to be inserted or false positive
3 : if σ[i] > 0, ∀i∈M
4 :     for i∈M
5 :         if σ[i] > 0
6 :             σ[i]− = 1
7 :     return ⊤, σ
8 : else
9 :     // Element not previously inserted
10 :    return ⊥, σ
```

Figure 5.1: Syntax instantiation for Counting filter with deletions.

real world uses cases, we allow reinsertion of deleted elements. We call this adversary, a *no-reinsertion adversary*.

If reinsertions are allowed, it becomes trivial for an adversary to disable the filter by inserting the same element $maxCounterValue + 1$ times. Additionally, if reinsertion is not allowed after deletion, this might lead to blocking scenarios in real life. For example, in the use case of routing (2.5.2), we might want to remove a route temporarily (e.g. during DDoS attacks), but add it later again.

5.1 Consistency rules

In this section we extend the consistency rules defined in [21], for our Counting filter with deletions Π.

Non-decreasing membership probability for insertions

If for all $\sigma \in \sum$, $x, y \in D$, $r \in \mathcal{R}$,

$$(b, \sigma') \leftarrow insert(x, \sigma; r) \Rightarrow Pr[\top \leftarrow qry(y, \sigma)] \leq Pr[\top \leftarrow qry(y, \sigma')].$$

Similar to Chapter 4, the more **Insert** queries we perform, the higher or equal is the probability that a *query* on a random element returns $\top$.

We note that the statement is equivalent to:

"If for all $\sigma, \sigma' \in \sum, y \in D$,

$$\sigma \subseteq \sigma' \Rightarrow Pr[\top \leftarrow qry(y, \sigma)] \leq Pr[\top \leftarrow qry(y, \sigma')]."$$

Where $\sigma \subseteq \sigma'$ denotes that σ is a subset of σ'.

Non-increasing membership probability for deletions

If for all $\sigma \in \sum, x, y \in D$,

$$(b, \sigma') \leftarrow delete(x, \sigma) \Rightarrow Pr[\top \leftarrow qry(y, \sigma)] \geq Pr[\top \leftarrow qry(y, \sigma')].$$

The more **Delete** queries we perform, the smaller or equal is the probability that a *query* on a random element returns $\top$.

Non-decreasing disabling probability for insertions

If for all $\sigma \in \sum, x, y \in D, r, r' \in \mathcal{R}$,

$$(b, \sigma') \leftarrow insert(x, \sigma; r) \Rightarrow Pr[(b, \sigma'') \leftarrow insert(y, \sigma; r') : b = \perp] \leq Pr[(b, \sigma'') \leftarrow insert(y, \sigma'; r') : b = \perp].$$

In this Chapter, we block an insertion when a counter needed for the insertion has reached its maximum. Since $(b, \sigma') \leftarrow insert(x, \sigma; r)$, $\sigma \subseteq \sigma'$ and all counters in σ' are higher or equal than in σ. Therefore, the probability that a counter reached its maximum is σ' is higher than in σ.

Compared to insertion-only Counting filters, we do not have a **permanent disabling** rule. We choose not to block the filter to be the closest to real scenarios.

5.2 Adversarial Correctness of Counting filters with deletions

As discussed previously in section 3.3, Counting filters with deletions are insecure in public settings. Clayton et. al.[8] proved that using a keyed PRF is not enough when deletions are allowed, and from a practical side, we mounted an attack against Counting filters with deletions in the public setting. By consequence, we only consider private Counting Filters with deletions in our proofs.

We consider the following scenario. The Counting filter is initialised empty. Then, the adversary has access to three oracles. **Qry** takes as input x an

element provided by A, and outputs if x has been inserted into Π. **Insert** inserts an element provided by A into Π, and **Delete** deletes an element provided by A from Π.

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

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

Oracle **Insert**(x)

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

Oracle **Delete**(x)

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

Oracle **Qry**(x)

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

Figure 5.2: Correctness game for the PDS Π using the simulation paradigm.

We define two games. First, we define the Real-or-Ideal game in Figure 5.2. We denote the real (d = 0) and ideal (d = 1) versions of Real-or-Ideal as Real and Ideal, respectively. The distinguisher D generates the game's output $d' \in \{0,1\}$.

We construct a simulator S in Figure 5.7, and used in Theorem 5.2. We explain our construction of the simulator below.

Deletions. We ensure in lines 1-2 of DeleteSim algorithm in Figure 5.3 that only inserted elements can be deleted.

We notice that, combined with the rule on counters reaching their maximum value described at the beginning of this chapter, this leads to no false negatives happening in the *Ideal* world.

Insertions. We define insertions in Ideal in line 2 of InsertSim algorithm in Figure 5.3. Insertions are defined such that the input element is not directly inserted, but instead, a uniformly random element is derived and then inserted instead of the input. Consequently, for each reinsertion of an element (i.e. same input) inside the filter, we do not reinsert the same element and we do not increment the same counters. It allows each insertion inside the filter to be independent from any previous insertions and deletions.

Queries. First, since in Ideal we take into consideration only deletions of inserted elements, we do not have false negatives and an inserted element must be inside the filter (lines 3-4 of QrySim algorithm in Figure 5.3). Then, if not inserted, an element is possibly a false positive. If the element was previously set as a false positive and no deletions happened since, the element

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

1 : $F \leftarrow\$ \mathit{Funcs}[\mathfrak{D}, \mathfrak{R}]$
2 : $\sigma \leftarrow \text{setup}(pp)$
3 : $\sigma' \leftarrow \text{setup}(pp)$
4 : $\text{inserted}_{bool} \leftarrow \{\}$
5 : // Value inserted **for** the corresponding element
6 : $\text{inserted}_{value} \leftarrow \{\}$
7 : // False positive elements set **to true**
8 : $\text{FPlist} \leftarrow \{\}$
9 : // insertions count at last query
10 : $\text{CALQ\_IN} \leftarrow \{\}$
11 : // deletions count at last query
12 : $\text{CALQ\_DEL} \leftarrow \{\}$
13 : // Insertions and deletions counter
14 : $ctr_{in} \leftarrow 0$
15 : $ctr_{del} \leftarrow 0$
16 : **return** $out \leftarrow\$ \mathcal{A}^{\textbf{InsertSim, DeleteSim, QrySim}}$

Oracle **QrySim**(x)

1 : $c^{ideal} \leftarrow\$ \mathsf{qry}^{\mathsf{Id}_{\mathfrak{R}}}(Y \leftarrow\$ \mathfrak{R}, \sigma)$
2 : $c^{G*} \leftarrow \mathsf{qry}^{\mathsf{F}}(x, \sigma')$
3 : **if** $\text{inserted}_{bool}[x] = \top$
4 : $\quad c \leftarrow \top$ // If x was a **false** a positive and no deletion happened since
5 : **else if** $\text{FPlist}[x] = \top \cap \text{CALQ\_DEL}[x] = ctr_{del}$
6 : $\quad c \leftarrow \top$
7 : // If x was not a **false** a positive and no insertion happened since
8 : **else if** $\text{FPlist}[x] = \bot \cap \text{CALQ\_IN}[x] = ctr_{in}$
9 : $\quad c \leftarrow \bot$
10 : **else**
11 : $\quad c \leftarrow c^{Ideal}$
12 : $\quad \text{FPlist}[x] = c$
13 : $\text{CALQ\_IN}[x] \leftarrow ctr_{in}$
14 : $\text{CALQ\_DEL}[x] \leftarrow ctr_{del}$
15 : **return** c

Oracle **InsertSim**(x)

1 : **if** $\text{inserted}_{bool}[x] = \bot$
2 : $\quad (a^{Ideal}, \sigma) \leftarrow \mathsf{insert}^{\mathsf{Id}_{\mathfrak{R}}}(\mathsf{y} \leftarrow\$ \mathfrak{R}, \sigma)$
3 : $\quad (a^{G*}, \sigma') \leftarrow \mathsf{insert}^{\mathsf{F}}(\mathsf{x}, \sigma')$
4 : $\quad a \leftarrow a^{Ideal}$
5 : $\quad$ **if** $a = \top$
6 : $\quad\quad ctr_{in} += 1$
7 : $\quad\quad \text{inserted}_{bool}[x] \leftarrow \top$
8 : $\quad\quad \text{inserted}_{value}[x] = y$
9 : **else**
10 : $\quad a \leftarrow \top$
11 : **return** a

Oracle **DeleteSim**(x)

1 : $(b^{G*}, \sigma') \leftarrow \mathsf{delete}^{\mathsf{F}}(\mathsf{x}, \sigma')$
2 : **if** $\text{inserted}_{bool}[x] = \top$
3 : $\quad (b^{Ideal}, \sigma) \leftarrow \mathsf{delete}^{\mathsf{Id}_{\mathfrak{R}}}(\text{inserted}_{\mathsf{value}}[\mathsf{x}], \sigma)$
4 : **else**
5 : $\quad b^{Ideal} \leftarrow \bot$
6 : $b \leftarrow b^{Ideal}$
7 : **if** $b = \top$
8 : $\quad ctr_{del} += 1$
9 : $\quad \text{inserted}_{bool}[x] = \bot$
10 : **return** b

Figure 5.3: Simulator.

must remain a false positive (lines 5-6). Indeed, since no deletions happened, no counters were decremented and consequently all counters > 0 allowing the element to be a false positive must remain > 0. In the opposite, if the element was previously not a false positive and no insertions happened since, the element must remain not a false positive (lines 8-9). Indeed, if the element was not a false positive, at least one of its corresponding counters was

equal to 0. Since no insertions happened, no counters were incremented and consequently the counter(s) equal to 0 could not become > 0. Finally, if we are not in the three previous cases, we need to recompute if the element is a false positive and derive a random element which is then queried to the filter (lines 10-11). Querying a random element instead of the input allows to generate a query response independently from the inserted elements.

5.2.1 Security bound derivation

We use the advantage of a Distinguisher in the *Real*-or-*Ideal* game described in Figure 5.2 in order to define adversarially correctness.

Definition 5.1 ***[21].*** *Let Π be a Counting filter with deletions, with public parameters pp, and let R_K be a keyed function family. We say Π is $(q_{in}, q_{del}, q_{qry}, t_a, t_s, t_d, \epsilon)$-adversarially correct under no reinsertion if, for all no-reinsertion adversaries A running in time at most t_a and making q_{in}, q_{del}, and q_{qry} queries to oracles **Insert**, **Delete**, **Qry** respectively in the Real-or-Ideal game (Figure 5.2) with the simulator S in Figure 5.3 that runs in time at most t_s, and for all distinguishers D running in time at most t_d, we have:*

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

We derive now a bound on the correctness of Counting filters when adversaries are able to insert, delete, and query elements.

Theorem 5.2 *Let q_{in}, q_{del}, and q_{qry} and $maxValue$ be non-negative integers, and let t_a, $t_d > 0$. Let $F : \mathfrak{D} \to \Re \equiv [m]^k$. Let Π be a Counting filter with public parameters pp and oracle access to F, such that Π satisfies the consistency rules from Definition 5.1 and F-decomposability (Definition 2.3.1).*
If $R_K : \mathfrak{D} \to \Re$ is an $(q_{in} + q_{del} + q_{qry}, t_a + t_d, \epsilon)$-secure pseudorandom function with key $K \leftarrow\!\!\$\ K$, then Π is $(q_{in}, q_{del}, q_{qry}, t_a, t_s, t_d, \epsilon')$-adversarially correct under no reinsertion, where

$$\epsilon' = \epsilon + 2 \cdot q_{in} \cdot m \cdot \left(\frac{e \cdot (q_{in} - 1) \cdot k}{maxValue \cdot m}\right)^{maxValue} + (2 \cdot q_{qry} + q_{del}) \cdot Pr_{NAI_{\Pi,pp}}(FP|q_{in}).$$

Proof. We define the intermediate game G in Figure 5.4. We define $d = 0$ (resp. $d = 1$) as the Real (resp. G) version of Real-or-G. We also define $d = 0$ (resp. $d = 1$) as the G (resp. Ideal) version of G-or-Ideal.

Now that we defined the intermediate games, we proceed with the proof. First, we start by defining an intermediate game G that replaces the PRF in *Real* with a random function. We bound the closeness of *Real* and G in Lemma 5.3 in terms of the PRF advantage. Then, we construct a game $G*$ (Figure 5.7) that looks identical to G, and show that $G*$ and *Ideal* are equal up until some "bad" events **E**, **E'** and **E''**. It allows us to bound the closeness

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

Figure 5.4: Intermediate game G for the proof of Theorem 5.2

of G and *Ideal* in Lemma 5.4 in terms of the probability of events **E**, **E′** and **E″**. Finally, the probabilities of these bad events are computed in Lemmas 5.6, 5.7 and 5.8 respectively.

$$Real \xrightarrow{\epsilon} G \equiv G* \xrightarrow{\Pr[\mathbf{E}],\ \Pr[\mathbf{E'}],\ \Pr[\mathbf{E''}]} Ideal$$

Figure 5.5: Sketch Theorem 5.2 proof.

Lemma 5.3 ***[21].*** *The difference in probability of an arbitrary t_d-distinguisher $\mathcal{D}$ outputting 1 in experiments of game Real-or-G (Figure 5.4) with a no-reinsertion $(q_{in}, q_{del}, q_{qry}, t_a)$-Counting filter adversary $\mathcal{A}$, is bounded by the maximal PRF advantage ϵ of an $(q_{in} + q_{del} + q_{qry}, t_a + t_d, \epsilon)$-PRF adversary attacking R_K:*

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

Proof. Consider the PRF adversary $\mathcal{B}$ in Figure 5.6, who instantiates the Counting filter that $\mathcal{A}$ queries using its **RoR** oracle, in relation to the Real-or-G game from Figure 5.4.

When b = 0, $\mathcal{B}$ is running *Real* for $\mathcal{A}$, where the PRF R_K is used to handle $\mathcal{A}$'s oracle queries to Π. When b = 1, $\mathcal{B}$ is instead running G for $\mathcal{A}$, where the truly random function F is used to handle $\mathcal{A}$'s oracle queries to Π. By inspection, the advantage of $\mathcal{B}$ is

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

By assumption in Theorem 5.2, R_K is an $(q_{in} + q_{del} + q_{qry}, t_a + t_d, \epsilon)$-secure PRF, hence no adversary $\mathcal{B}$ making at most $q_{in} + q_{del} + q_{qry}$ queries and running in time at most $t_a + t_d$ can have advantage greater than ϵ. Therefore,

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

□

PRF Adversary $\mathcal{B}^{RoR}$

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

Figure 5.6: PRF adversary $\mathfrak{B}$ for Lemma 5.3

Let us denote by G^* a modified version of *Ideal* that runs S, but with modifications marked in blue in Figure 5.7.

Lemma 5.4 *For $i \in [q_{qry}]$, let c_i^{Ideal} be the response to A's i^{th} **Qry** query in the Ideal game, and let c_i^{G*} be the response in the G* game. Let **E** be the event that the response to A's i^{th} **Qry** query differ in line 12 of QrySim algorithm in Figure 5.7 for some i:*

$$\boldsymbol{E} := [\text{The first mismatch in query responses is due to } c_i^{Ideal} \neq c_i^{G*} \text{ for some } i \in [q_{qry}]]. \tag{5.1}$$

*For $i \in [q_{del}]$, let b_i^{Ideal} be the response to A's i^{th} **Delete** query in the Ideal game, let b_i^{G*} be the response in the G* game. Let **E'** be the event that A's i^{th} **Delete** query differ in line 6 of DeleteSim algorithm in Figure 5.7 for some i:*

$$\boldsymbol{E'} := [\text{The first mismatch in query responses is due to } b_i^{Ideal} \neq b_i^{G*} \text{ for some } i \in [q_{del}]]. \tag{5.2}$$

*For $i \in [q_{in}]$, let a_i^{Ideal} be the response to A's i^{th} **Insert** query in the Ideal game, let a_i^{G*} be the response in the G* game. Let **E''** be the event that A's i^{th} **Insert** query differ in line 4 of InsertSim algorithm in Figure 5.7 for some i:*

$$\boldsymbol{E''} := [\text{The first mismatch in query responses is due to } a_i^{Ideal} \neq a_i^{G*} \text{ for some } i \in [q_{in}]]. \tag{5.3}$$

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

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

Proof. G and G^* use the same function F in all algorithms. The main difference between G and G^* is that in the latter, $\mathcal{A}$ interacts with a simulator that intercepts some of the queries to the simulated Counting filter, and does some input-output bookkeeping absent in G. We show that these extra operations run by S do not affect the return values of Π and therefore, G and G^* look the same from $\mathcal{A}$'s point of view.

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

$F \leftarrow\$ Funcs[\mathfrak{D}, \mathfrak{R}]$
$\sigma \leftarrow \text{setup}(pp)$
$\sigma' \leftarrow \text{setup}(pp)$
$\text{inserted}_{bool} \leftarrow \{\}$
// Value inserted **for** the corresponding element
$\text{inserted}_{value} \leftarrow \{\}$
// False positive elements set **to true**
$\text{FPlist} \leftarrow \{\}$
// insertions count at last query
$\text{CALQ\_IN} \leftarrow \{\}$
// deletions count at last query
$\text{CALQ\_DEL} \leftarrow \{\}$
// Insertions and deletions counter
$ctr_{in} \leftarrow 0$
$ctr_{del} \leftarrow 0$
return $out \leftarrow\$ \mathcal{A}^{\textbf{InsertSim, DeleteSim, QrySim}}$

Oracle QrySim(x) - Ideal, G*

$c^{ideal} \leftarrow\$ \mathsf{qry}^{\mathsf{Id}_{\mathfrak{R}}}(Y \leftarrow\$ \mathfrak{R}, \sigma)$
$c^{G*} \leftarrow \mathsf{qry}^{\mathsf{F}}(x, \sigma')$
if $\text{inserted}_{bool}[x] = \top$
  $c \leftarrow \top, c^{G*}$
// If x was a **false** a positive and no deletion happened since
else if $\text{FPlist}[x] = \top \cap \text{CALQ\_DEL}[x] = ctr_{del}$
  $c \leftarrow \top$
// If x was not a **false** a positive and no insertion happened since
else if $\text{FPlist}[x] = \bot \cap \text{CALQ\_IN}[x] = ctr_{in}$
  $c \leftarrow \bot$
else
  $c \leftarrow c^{Ideal}, c^{G*}$
  $\text{FPlist}[x] = c$
$\text{CALQ\_IN}[x] \leftarrow ctr_{in}$
$\text{CALQ\_DEL}[x] \leftarrow ctr_{del}$
return c

Oracle InsertSim(x) - Ideal, G*

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

Oracle DeleteSim(x) - Ideal, G*

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

Figure 5.7: Simulator with G*.

By inspection of algorithm **InsertSim**, we see that it always returns the return value of the **insert** algorithm when the element is not already inserted in the filter. Since we derive a correctness bound for a no-reinsertion adversary, **InsertSim** always returns the return value of the **insert** algorithm for every insertion. Consequently, answers obtained from **InsertSim** in G^*

always agree with what would A obtain using the **insert** algorithm in G.

By inspection of algorithm **DeleteSim** in Figure 5.7, we see that for every deletion it returns the return value of the **delete** function. Consequently, answers obtained from **DeleteSim** in G^* always agree with what $\mathcal{A}$ would obtain using the **delete** algorithm in G.

Then, inspecting **QrySim**, we show that lines 6-7 in Figure 5.7 do not cause discrepancies. Lines 6-7 are executed if the current element was a false positive at the time of the last call to **QrySim** on this element, and no deletions were made since. As shown in the previous paragraph, since no elements were deleted in G^* since the last call to **QrySim** on this element, no elements were deleted in G either. Therefore, no counters were decremented in both games, including the counters allowing the element to be a false positive. Since qry^F is a deterministic algorithm, **qry** in G would return $\top$, and regarding **QrySim**, the element has to remain a false positive and it needs to return $\top$ again.
Similarly, lines 9-10 (5.7) do not cause discrepancies. Lines 9-10 are executed if the element was not previously a false positive and no insertions were made since the last call to **QrySim** on the current element. As shown previously, since no elements were inserted in G^* since the last call to **QrySim** on this element, no elements were inserted in G either. Therefore, no counters were incremented in both games, including the counters needed for the element to become a false positive. Since qry^F is a deterministic algorithm, **qry** in G would return $\bot$, and regarding **QrySim**, the element has to remain not a false positive and it needs to return $\bot$ again.
Since no other operations in **QrySim** affect its return value, the game G^* is identical to G (Figure 5.4) from the point of view of $(\mathcal{A}, \mathcal{D})$.

We now look at where G^* and *Ideal* can diverge for the first time.

First, answers to **InsertSim** queries might differ. As we explained previously, if a counter reaches its maximum value in the filter and is needed during the insertion of a new element, the element is not inserted. If the element cannot be inserted, **InsertSim** return $\bot$, while if the insertion succeeds, the algorithm returns $\top$. Therefore, since elements inserted in G^* and *Ideal* are different, their corresponding counters are different, and the first mismatch in query responses can happen in **InsertSim** and corresponds to event **E''** defined in Equation 5.3.

Otherwise, answers to **DeleteSim** queries might differ. In G^*, not only inserted elements but also false positives can be deleted, while in *Ideal* only previously inserted elements can be deleted. By consequence, depending on insertions, deletions can succeeds in only one game and the first mismatch in query responses can happen in **DeleteSim** and corresponds to event **E'** defined in Equation 5.2.

Finally, otherwise answers to **QrySim** queries might differ. We observe in Figure 5.7 that when answers to the **QrySim** algorithm are not fixed, we do not check the same counters in G^* and in $Ideal$. Therefore, these queries might output different answers and the first mismatch in query responses can happen in **QrySim**, and corresponds to event **E** from Equation 5.1

By consequence, the games $Ideal$ and G^* (and hence G) are equal from the perspective of $(\mathcal{A}, \mathcal{D})$ at least up until the events from Equations 5.1, 5.2 or 5.3 for some $i \in [q_{in}], [q_{del}]$ or $[q_{qry}]$. We denote these events by **E**, **E'**, **E''**, and we have

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

□

Definition 5.5 *Let $insert^{F}_{partial}(x, \sigma)$ be the insert algorithm from Figure 2.1. Therefore, the non-adversarially-influenced state Definition 2.5, and the false positive probability in Equation 2.3 hold for the $insert^{F}_{partial}(x, \sigma)$ algorithm.*

Lemma 5.6 ***QrySim difference.*** *Let the event* ***E*** *be defined as in Lemma 5.4. Then,*

$$\mathsf{Pr}[\boldsymbol{E}] \leq 2q_{qry} \cdot Pr_{NAI_{\Pi,pp}}(FP|q_{in}).$$

Proof. We need to compute Pr[**E**].

Let $\sigma^{(i)}$ denote the state σ and let $\sigma^{(i)'}$ denote the state σ' before A's i-th Qry query. We calculate Pr[**E**] in game *Ideal*, with outputs c_i^{Ideal} and c_i^{G*} defined as in lines 4, 7, 10 and 12 of algorithm **QrySim** in Figure 5.7 at the ith query.

We observe that event **E** can occur during each query of an element x_i. Therefore, we consider all queries $i \in [q_{qry}]$ when upper bounding Pr[**E**].

First, we note that we do not have to consider the case where a query on an *inserted element* returns different values in both games in lines 3-4 in Figure 5.7. Indeed, in this case, the element must be a false negative in game *G**, while still be positive in *Ideal*, or could have been inserted in *Ideal* but not *G**. In order for the first scenario to happen, deletion(s) of a false positive(s) must have occurred previously and by consequence the difference would have appeared first in **DeleteSim** and do not need to be considered in **QrySim**. For the second scenario, the first difference happens in **InsertSim** .Therefore, we focus on the *event for elements x_i that have not been inserted into* Π at the time they are queried to **QrySim** (and hence could return a false positive result).

Additionally, since we are looking for the first time outputs mismatch, we do not take special care of the case where the outputs are necessarily equal before the first mismatch, i.e. return value is defined in lines 7-10 in Figure 5.7. Thus, we focus on output values defined in via line 12.

We see that event **E** can be written as

$\mathbf{E}$ = "The first mismatch in query responses is due to $c_i^{Ideal} \neq c_i^{G*}$for some $i \in [q_{qry}]$

and the queried element was not inserted".

We compute

$$\begin{aligned}\mathsf{Pr}[\mathbf{E}] = \mathsf{Pr}[&\text{The first mismatch in query responses is due to } c_i^{Ideal} \neq c_i^{G*} \text{for some } i \in [q_{qry}] \\ &\text{and the queried element was not inserted}]\end{aligned}$$

$$\begin{aligned}\leq \sum_{i=1}^{q_{qry}} \mathsf{Pr}[&\text{The first mismatch in query responses is due to } \mathbf{Qry}(y_i) \\ &\text{where } y_i \text{ was not inserted and } c_i^{Ideal} \neq c_i^{G*}].\end{aligned}$$

We define the event $\mathbf{E}^{mismatch}_{qry_i}$ as

$$\mathbf{E}^{mismatch}_{qry_i} = \text{"The first mismatch in query responses is due to } \mathbf{Qry}(y_i)\text{"}.$$

$$\begin{aligned}
\mathsf{Pr}[\mathbf{E}] &\leq \sum_{i=1}^{q_{qry}} \mathsf{Pr}[\mathbf{E}^{mismatch}_{qry_i} \text{ where } y_i \text{ was not inserted and } c_i^{Ideal} = \top \cap\ c_i^{G*} = \bot] \\
&\quad +\ \mathsf{Pr}[\mathbf{E}^{mismatch}_{qry_i} \text{ where } y_i \text{ was not inserted and } c_i^{Ideal} = \bot \cap c_i^{G*} = \top] \\
&\leq \sum_{i=1}^{q_{qry}} \mathsf{Pr}[\mathbf{E}^{mismatch}_{qry_i} \text{ where } y_i \text{ was not inserted and } c_i^{Ideal} = \top] \\
&\quad + \mathsf{Pr}[\mathbf{E}^{mismatch}_{qry_i} \text{ where } y_i \text{ was not inserted and } c_i^{G*} = \top] \\
&= \sum_{i=1}^{q_{qry}} \mathsf{Pr}[\mathbf{E}^{Ideal}_{qry_i}] + \mathsf{Pr}[\mathbf{E}^{G*}_{qry_i}], \qquad (5.4)
\end{aligned}$$

where we define the event $\mathbf{E}^{Ideal}_{qry_i}$ as

$$\mathbf{E}^{Ideal}_{qry_i} = [\mathbf{E}^{mismatch}_{qry_i} \text{ where } y_i \text{ was not inserted and } c_i^{Ideal} = \top]$$

and the event $\mathbf{E}^{G*}_{qry_i}$ as

$$\mathbf{E}^{G*}_{qry_i} = [\mathbf{E}^{mismatch}_{qry_i} \text{ where } y_i \text{ was not inserted and } c_i^{G*} = \top].$$

We focus now on $\mathsf{Pr}[\mathbf{E}^{Ideal}_{qry_i}]$. From the simulator and the previous explanations, we have:

$$\mathsf{Pr}[\mathbf{E}^{Ideal}_{qry_i}] = \underset{\substack{Ideal\text{'s coins} \\ r \leftarrow\$ R \\ \mathcal{A}\text{'s coins}}}{\mathsf{Pr}}[\mathbf{E}^{mismatch}_{qry_i} \ \cap\ (c \leftarrow \mathrm{qry}^{Id_{\Re}}(Y \xleftarrow{r} \Re, \sigma^{(i)}) : c = \top)].$$

$\mathcal{A}$'s coins represents all the adversary's reactions to the responses they may observe during the game. *Ideal*'s coins represent the randomness generated in *Ideal*. Therefore, fixing $\mathcal{A}$'s and *Ideal*'s coins determines the adversary's exact behaviour. It uniquely determines each state $\sigma^{(i)}$ and the elements inserted.

$$\mathsf{Pr}[\mathbf{E}^{Ideal}_{qry_i}] = \sum_{\mathcal{A}'s\ coins} \mathsf{Pr}_{\substack{Ideal's\ coins \\ r \leftarrow \$ R}} \left[\cap \left(c \leftarrow \mathrm{qry}^{Id_{\Re}}(Y \xleftarrow{r} \Re, \sigma^{(i)}) : c = \top\right) \;\middle|\; \begin{array}{c} \mathbf{E}^{mismatch}_{qry_i} \\ \mathcal{A}'s\ coins \end{array} \right] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$= \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \mathsf{Pr}_{r \leftarrow \$ R} \left[\cap \left(c \leftarrow \mathrm{qry}^{Id_{\Re}}(Y \xleftarrow{r} \Re, \sigma^{(i)}) : c = \top\right) \;\middle|\; \begin{array}{c} \mathbf{E}^{mismatch}_{qry_i} \\ \mathcal{A}'s\ coins \\ Ideal's\ coins \end{array} \right] \cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins].$$

Let Δ_{max} represent the state σ after all q_{in} insertions, where each element was inserted using the $insert^{Id_{\Re}}_{partial}$ function in Definition 5.5. Since Δ_{max} contains all inserted elements, $\sigma^{(i)} \subseteq \Delta_{max}$, and using the **non-decreasing membership for insertion** consistency rule:

$$\mathsf{Pr}[\mathbf{E}^{Ideal}_{qry_i}] \leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \mathsf{Pr}_{r \leftarrow \$ R} \left[\cap \left(c \leftarrow \mathrm{qry}^{Id_{\Re}}(Y \xleftarrow{r} \Re, \Delta_{max}) : c = \top\right) \;\middle|\; \begin{array}{c} \mathbf{E}^{mismatch}_{qry_i} \\ \mathcal{A}'s\ coins \\ Ideal's\ coins \end{array} \right] \cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins].$$

Since in Δ_{max} all elements are inserted and mapped using counters selected uniformly and independently at random, the adversary has no power and the probability does not depend on $\mathcal{A}$'s coins anymore:

$$\mathsf{Pr}[\mathbf{E}^{Ideal}_{qry_i}] \leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \mathsf{Pr}_{r \leftarrow \$ R} \left[\cap \left(c \leftarrow \mathrm{qry}^{Id_{\Re}}(Y \xleftarrow{r} \Re, \Delta_{max}) : c = \top\right) \;\middle|\; \begin{array}{c} \mathbf{E}^{mismatch}_{qry_i} \\ Ideal's\ coins \end{array} \right] \cdot \mathsf{Pr}[Ideal's\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$\leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \mathsf{Pr}_{r \leftarrow \$ R} \left[\left(c \leftarrow \mathrm{qry}^{Id_{\Re}}(Y \xleftarrow{r} \Re, \Delta_{max}) : c = \top\right) \;\middle|\; Ideal's\ coins \right] \cdot \mathsf{Pr}[Ideal's\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins].$$

Reversing the conditional probability rule we get:

$$\mathsf{Pr}[\mathbf{E}^{Ideal}_{qry_i}] \leq \sum_{\mathcal{A}'s\ coins} \mathop{\mathsf{Pr}}_{\substack{Ideal's\ coins \\ r \leftarrow\$ R}} [c \leftarrow \mathrm{qry}^{Id_{\mathfrak{R}}}(Y \xleftarrow{r} \mathfrak{R}, \Delta_{max}) : c = \top] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$= \mathop{\mathsf{Pr}}_{\substack{Ideal's\ coins \\ r \leftarrow\$ R}} [c \leftarrow \mathrm{qry}^{Id_{\mathfrak{R}}}(Y \xleftarrow{r} \mathfrak{R}, \Delta_{max}) : c = \top] \cdot \sum_{\mathcal{A}'s\ coins} \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$= \mathop{\mathsf{Pr}}_{\substack{Ideal's\ coins \\ r \leftarrow\$ R}} [c \leftarrow \mathrm{qry}^{Id_{\mathfrak{R}}}(Y \xleftarrow{r} \mathfrak{R}, \Delta_{max}) : c = \top]$$

$$= \mathsf{Pr}\left[\begin{array}{c} Y_1, .., Y_{q_{in}} \leftarrow\$\ \mathfrak{R}, \\ \Delta \leftarrow\$\ setup(pp), \\ (\top, \Delta) = insert^{Id_{\mathfrak{R}}}_{partial}(Y_i, \Delta)\ for\ i \in [1, .., q_{in}] \\ : \top = qry^{Id_{\mathfrak{R}}}(Y \leftarrow\$\ \mathfrak{R}, \Delta) \end{array}\right]$$

Since Δ contains all q_{in} elements randomly and independently inserted in the filter, we see that Δ corresponds to the non-adversarially-influenced state defined in Definition 2.5 with q_{in} elements:

$$\mathsf{Pr}[\mathbf{E}^{Ideal}_{qry_i}] \leq \mathsf{Pr}\left[\begin{array}{c} F \leftarrow\$\ \mathrm{Funcs}[\mathfrak{D}, \mathfrak{R}], \\ [\mathsf{x}_1, ..., \mathsf{x}_{\mathsf{q_{in}}}] \leftarrow\$\ U(S \in \mathsf{P}_{\mathsf{lists}}(\mathfrak{D}) || S| = q_{in}) \\ \Delta \leftarrow\$\ \mathrm{setup}(pp) \\ (b, \Delta) \leftarrow\$\ insert^{F}_{partial}(x_k, \Delta)\ for\ k = 1, ..., q_{in} \\ : \top = qry^{F}(x \leftarrow\$\ U(\mathfrak{D} \backslash [\mathsf{x}_1, ..., \mathsf{x}_{\mathsf{q_{in}}}]), \Delta) \end{array}\right]$$

Finally, we see that this probability corresponds to the NAI false positive probability in Equation 2.3 for q_{in} elements:

$$\mathsf{Pr}[\mathbf{E}^{Ideal}_{qry_i}] \leq \mathsf{Pr}_{NAI_{\Pi,pp}}[FP|q_{in}]. \tag{5.5}$$

We focus now on $\mathsf{Pr}[\mathbf{E}^{G*}_{qry_i}]$.

We introduce $G*$'s coins. G^* coins represents the random function F sampled independently from $\mathcal{A}$'s and *Ideal*'s coins.

$$\mathsf{Pr}[\mathbf{E}^{G*}_{qry_i}] = \underset{\substack{G*'s\ coins \\ Ideal's\ coins \\ \mathcal{A}'s\ coins}}{\mathsf{Pr}} [\mathbf{E}^{mismatch}_{qry_i} \cap (c \leftarrow \mathrm{qry}^F(y_i, \sigma^{(i)'}) : c = \top) \cap (y_i \text{ not inserted})]$$

$$= \sum_{\mathcal{A}'s\ coins} \underset{\substack{G*'s\ coins \\ Ideal's\ coins}}{\mathsf{Pr}} \left[\begin{array}{c} \mathbf{E}^{mismatch}_{qry_i} \\ \cap (c \leftarrow \mathrm{qry}^F(y_i, \sigma^{(i)'}) : c = \top) \\ \cap (y_i \text{ not inserted}) \end{array} \middle| \mathcal{A}'s\ coins \right]$$
$$\cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$= \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \underset{G*'s\ coins}{\mathsf{Pr}} \left[\begin{array}{c} \mathbf{E}^{mismatch}_{qry_i} \\ \cap (c \leftarrow \mathrm{qry}^F(y_i, \sigma^{(i)'}) : c = \top) : c = \top \\ \cap (y_i \text{ not inserted}) \end{array} \middle| \begin{array}{c} \mathcal{A}'s\ coins \\ Ideal's\ coins \end{array} \right]$$
$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$\leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \underset{G*'s\ coins}{\mathsf{Pr}} \left[\begin{array}{c} (c \leftarrow \mathrm{qry}^F(y_i, \sigma^{(i)'}) : c = \top) : c = \top \\ \cap (y_i \text{ not inserted}) \end{array} \middle| \begin{array}{c} \mathcal{A}'s\ coins \\ Ideal's\ coins \end{array} \right]$$
$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins] \tag{5.6}$$

Let $\Omega_{max,i}$ represent the state σ' containing all inserted elements $\{x_1, ..., x_{q_{in}}\} \backslash \{y_i\}$ where each element was inserted using the $insert^{Id_{\Re}}_{partial}$ function. Since $\Omega_{max,i}$ contains all inserted elements except the currently queried y_i, $\sigma^{(i)'} \subseteq \Omega_{max,i}$, and using the **non-decreasing membership probability for insertion** consistency rule:

$$\mathsf{Pr}[\mathbf{E}^{G*}_{qry_i}] \leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \mathsf{Pr}_{G*'s\ coins}\left[\begin{array}{c}(c \leftarrow \mathrm{qry}^F(y_i, \Omega_{max,i}) : c = \top) : c = \top) \\ \cap\ (y_i \text{ not inserted})\end{array} \mid \begin{array}{c}\mathcal{A}'s\ coins \\ Ideal's\ coins\end{array}\right]$$
$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$= \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \cdot \mathsf{Pr}\left[\begin{array}{c} F \leftarrow\!\$\ \mathrm{Funcs}[\mathfrak{D}, \mathfrak{R}], \\ \Omega \leftarrow\!\$\ setup(pp), \\ \text{for each } x_k \in \{x_1, .., x_{q_{in}}\} \backslash \{y_i\}, \\ (\top, \Omega) = insert^F_{partial}(x_k, \Omega) \\ : \top = qry^F(y_i, \Omega) \end{array}\right]$$

$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

Since Counting filters are function-decomposable (Section 2.3.1), we rewrite:

$$\mathsf{Pr}[\mathbf{E}^{G*}_{qry_i}] \leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \cdot \mathsf{Pr}\left[\begin{array}{c} F \leftarrow\!\$\ \mathrm{Funcs}[\mathfrak{D}, \mathfrak{R}], \\ \Omega \leftarrow\!\$\ setup(pp), \\ \text{for each } x_k \in \{x_1, .., x_{q_{in}}\} \backslash \{y_i\}, \\ (\top, \Omega) = insert^{Id_{\mathfrak{R}}}_{partial}(F(x_k), \Omega) \\ : \top = qry^{Id_{\mathfrak{R}}}(F(y_i), \Omega) \end{array}\right]$$

$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

The set $\{x_1, .., x_{q_{in}}\} \backslash \{y_i\}$ is fixed due to $\mathcal{A}$'s and *Ideal*'s coins. As we consider all q_{in} insertions except the queried element y_i, the state contains $|\{x_1, .., x_{q_{in}}\} \backslash \{y_i\}|$ elements. Since F is a random function and using the **non-decreasing membership probability for insertions** consistency rule:

$$\mathsf{Pr}[\mathbf{E}^{G*}_{qry_i}] \leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins}$$

$$\cdot \mathsf{Pr}\left[\begin{array}{c} Y_1, .., Y_{|\{x_1,..,x_{q_{in}}\} \backslash \{y_i\}|} \leftarrow\$\ \Re, \\ \Omega \leftarrow\$\ setup(pp), \\ (\top, \Omega) = insert^{Id_\Re}_{partial}(Y_k, \Omega)\ for\ k \in [1, .., q_{|\{x_1,..,x_{q_{in}}\} \backslash \{y_i\}|}] \\ : \top = qry^{Id_\Re}(Y \leftarrow\$\ \Re, \Omega) \end{array}\right]$$

$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$\leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \cdot \mathsf{Pr}\left[\begin{array}{c} Y_1, .., Y_{q_{in}} \leftarrow\$\ \Re, \\ \Omega \leftarrow\$\ setup(pp), \\ (\top, \Omega) = insert^{Id_\Re}_{partial}(Y_k, \Omega)\ for\ k \in [1, .., q_{in}] \\ : \top = qry^{Id_\Re}(Y \leftarrow\$\ \Re, \Omega) \end{array}\right]$$

$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

Since all q_{in} elements are randomly and independently inserted in the filter, we observe that the last constructed state Ω corresponds to the non-adversarially-influenced state defined in Definition 2.5 with q_{in} elements:

$$\mathsf{Pr}[\mathbf{E}^{G*}_{qry_i}] \leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins}$$

$$\cdot \mathsf{Pr}\left[\begin{array}{c} F \leftarrow\$ \text{ Funcs}[\mathfrak{D}, \mathfrak{R}], \\ [\mathsf{x}_1, ..., \mathsf{x}_{\mathsf{q_{in}}}] \leftarrow\$ \mathcal{U}(S \in \mathsf{P}_{\mathsf{lists}}(\mathfrak{D}) || S| = q_{in}) \\ \Omega \leftarrow\$ \text{setup}(pp) \\ (b, \Omega) \leftarrow\$ insert^F_{partial}(x_k, \Omega)\ for\ k = 1, ..., q_{in} \\ : \top = qry^F(x \leftarrow\$ \mathcal{U}(\mathfrak{D} \backslash [\mathsf{x}_1, ..., \mathsf{x}_{\mathsf{q_{in}}}]), \Omega) \end{array}\right]$$

$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins].$$

Finally, we see that this probability corresponds to the NAI false positive probability given in Equation 2.3 for q_{in} elements:

$$\mathsf{Pr}[\mathbf{E}^{G*}_{qry_i}] \leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \mathsf{Pr}_{NAI_{\Pi,pp}}[FP|q_{in}] \cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$= \mathsf{Pr}_{NAI_{\Pi,pp}}[FP|q_{in}] \cdot \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \mathsf{Pr}[Ideal's\ coins \cap \mathcal{A}'s\ coins]$$

$$= \mathsf{Pr}_{NAI_{\Pi,pp}}[FP|q_{in}]. \tag{5.7}$$

Finally, combining Equations 5.4, 5.5 and 5.7, we get

$$\mathsf{Pr}[\mathrm{E}] \leq \sum_{i=1}^{q_{qry}} \mathsf{Pr}[\mathbf{E}^{Ideal}_{qry_i}] + \mathsf{Pr}[\mathbf{E}^{G*}_{qry_i}]$$

$$\leq \sum_{i=1}^{q_{qry}} (\mathsf{Pr}_{NAI_{\Pi,pp}}[FP|q_{in}]) + (\mathsf{Pr}_{NAI_{\Pi,pp}}[FP|q_{in}])$$

$$= 2 \cdot q_{qry} \cdot \mathsf{Pr}_{NAI_{\Pi,pp}}[FP|q_{in}].$$

□

Lemma 5.7 ***DeleteSim difference.*** *Let the event* $\boldsymbol{E'}$ *be defined as in Lemma 5.4. Then,*

$$\mathsf{Pr}[\boldsymbol{E'}] \leq q_{del} \cdot \mathsf{Pr}_{NAI_{\Pi,pp}}[FP|q_{in}].$$

Proof. Let $\sigma^{(i)'}$ denote the state of σ' before A's i-th deletion. We calculate Pr[**E′**] in game Ideal, with output b_i^{Ideal} and b_i^{G*} defined as in line 1, 3, and 5 of algorithm **DeleteSim** in Figure 5.7 at ith deletion. Let y_i be the i^{th} deleted element.

Since we are interested in the first output mismatch between games and looking at the Counting filter's syntax for the delete algorithm in Figure 5.1, we see that event **E′** can be stated as

E′ = "The first mismatch in query responses is due to $b_i^{Ideal} \neq b_i^{G*}$ for some $i \in [q_{del}]$".

Event **E′** can occur during each deletion of an element x_i. Therefore, we consider all deletions $i \in [q_{del}]$ when upper bounding $\mathsf{Pr}[\mathbf{E'}]$.

$$\mathsf{Pr}[\mathbf{E'}] = \mathsf{Pr}[\text{The first mismatch in query responses is due to } b_i^{Ideal} \neq b_i^{G*} \text{ for some } i \in [q_{del}]]$$

$$\leq \sum_{i=1}^{q_{del}} \mathsf{Pr}[\text{The first mismatch in query responses is due to } \mathbf{Delete}(y_i) \text{ and } b_i^{Ideal} \neq b_i^{G*}].$$

We define the event $\mathbf{E}_i^{mismatch}$ as

$\mathbf{E}_{delete_i}^{mismatch}$ = "The first mismatch in query responses is due to **Delete**(y_i)".

$$\mathsf{Pr}[\mathbf{E'}] \leq \sum_{i=1}^{q_{del}} \mathsf{Pr}[\mathbf{E}_{delete_i}^{mismatch} \cap b_i^{Ideal} = \top \cap b_i^{G*} = \bot] + \mathsf{Pr}[\mathbf{E}_{delete_i}^{mismatch} \cap b_i^{Ideal} = \bot \cap b_i^{G*} = \top].$$

The return values differ when we are able to delete an element only in one of the games.

We show that $\mathsf{Pr}[\mathbf{E}_{delete_i}^{mismatch} \cap b_i^{Ideal} = \top \cap b_i^{G*} = \bot] = 0$. In this case, the element is present inside the filter in the *Ideal* game, which means that the element was inserted and not deleted, but not in game G*. This situation is only possible if the element has become a false negative in game G* or was not inserted in σ' but was in σ. However, since we consider the first time outputs differ, this situation is impossible as the first difference would have happened in a previous **Delete** query, when deleting an element which is a false positive, or within some previous **Insert** query.

By consequence, we only consider

$$\mathsf{Pr}[\mathbf{E'}] \leq \sum_{i=1}^{q_{del}} \mathsf{Pr}[\mathbf{E}^{mismatch}_{delete_i} \cap b_i^{Ideal} = \perp \cap b_i^{G*} = \top].$$

We focus now on $\mathsf{Pr}[\mathbf{E'}_i] = \mathsf{Pr}[\mathbf{E}^{mismatch}_{delete_i} \cap (b_i^{Ideal} = \perp \cap b_i^{G*} = \top)]$.

From the Simulator and the previous explanations, we have:

$$\mathsf{Pr}[\mathbf{E'}_i] = \underset{\substack{G*'s\ coins \\ Ideal's\ coins \\ \mathcal{A}'s\ coins}}{\mathsf{Pr}} [\mathbf{E}_i^{mismatch} \cap ((b,\sigma') \leftarrow \text{delete}^F(y_i, \sigma^{(i)'}) : b = \top) \cap (b_i^{Ideal} = \perp)]$$

$$= \sum_{\mathcal{A}'s\ coins} \underset{\substack{G*'s\ coins \\ Ideal's\ coins}}{\mathsf{Pr}} \left[\begin{array}{c} \mathbf{E}_i^{mismatch} \cap ((b,\sigma') \leftarrow \text{delete}^F(y_i, \sigma^{(i)'}) : b = \top) \\ \cap (b_i^{Ideal} = \perp) \end{array} \mid \mathcal{A}'s\ coins \right]$$
$$\cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$= \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \underset{G*'s\ coins}{\mathsf{Pr}} \left[\begin{array}{c} \mathbf{E}_i^{mismatch} \\ \cap ((b,\sigma') \leftarrow \text{delete}^F(y_i, \sigma^{(i)'}) : b = \top) \\ \cap (b_i^{Ideal} = \perp) \end{array} \mid \begin{array}{c} \mathcal{A}'s\ coins \\ Ideal's\ coins \end{array} \right]$$
$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins].$$

From line 1 of the DeleteSim algorithm in Figure 5.7 and its corresponding line 3 of delete algorithm in Figure 5.1, we see that $((b,\sigma') \leftarrow \text{delete}^F(y_i, \sigma^{(i)'}) : b = \top)$ if and only if $(c \leftarrow \text{query}^F(y_i, \sigma^{(i)'}) : c = \top)$. Additionally, we see in lines 2-5 of the DeleteSim algorithm in Figure 5.7 that $(b_i^{Ideal} = \perp)$ if and only if y_i was not previously inserted. Therefore:

$$\mathsf{Pr}[\mathbf{E'}_i] \leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \underset{G*'s\ coins}{\mathsf{Pr}} \left[\begin{array}{c} (c \leftarrow \text{qry}^F(y_i, \sigma^{(i)'}) : c = \top) \\ \cap\ (y_i \text{ was not inserted}) \end{array} \mid \begin{array}{c} \mathcal{A}'s\ coins \\ Ideal's\ coins \end{array} \right]$$
$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins].$$

We observe that this last probability can be calculated following the derivation of Equation 5.6 of Lemma 5.6 and we omit the details. Then,

$$\mathsf{Pr}[\mathbf{E'}_i] \leq \mathsf{Pr}_{NAI_{\Pi,pp}}[FP | q_{in}].$$

Finally, we compute

$$\begin{aligned}\mathsf{Pr}[\mathbf{E}'] &\leq \sum_{i=1}^{q_{del}} \mathsf{Pr}[\mathbf{E}'_i] \\ &\leq q_{del} \cdot \mathsf{Pr}_{NAI_{\Pi,pp}}[FP|q_{in}].\end{aligned}$$

□

Lemma 5.8 ***InsertSim difference.*** *Let the event* $\boldsymbol{E''}$ *be defined as in Lemma 5.4. Then,*

$$\mathsf{Pr}[\boldsymbol{E''}] \leq 2 \cdot q_{in} \cdot m \cdot (\frac{e \cdot (q_{in} - 1) \cdot k}{maxValue \cdot m})^{maxValue}).$$

Proof.

Let $\sigma^{(i)}$ denote σ and let $\sigma^{(i)'}$ denote σ' before A's i-th insertion. We calculate Pr[**E″**] in game *Ideal*, with outputs a_i^{Ideal} and a_i^{G*} defined as in lines 2 and 3 of algorithm **InsertSim** in Figure 5.7 at the ith insertion.

Since we are interested in the first output mismatch between games and looking at the Counting filter's syntax for the insert algorithm in Figure 5.1, we see that event **E″** can be stated as

E″ = ″The first mismatch in query responses is due to $a_i^{Ideal} \neq a_i^{G*}$for some $i \in [q_{in}]$″.

Event **E″** can occur during each insertion of an element x_i. Therefore, we consider all insertions $i \in [q_{in}]$ when upper bounding $\mathsf{Pr}[\mathbf{E''}]$.

$$\mathsf{Pr}[\mathbf{E''}] = \mathsf{Pr}[\text{The first mismatch in query responses is due to } a_i^{Ideal} \neq a_i^{G*} \text{for some } i \in [q_{in}]]$$

$$\leq \sum_{i=1}^{q_{in}} \mathsf{Pr}[\text{The first mismatch in query responses is due to } \mathbf{Insert}(x_i) \text{ and } a_i^{Ideal} \neq a_i^{G*}].$$

We define the event $\mathbf{E}_{insert_i}^{mismatch}$ as

$\mathbf{E}_{insert_i}^{mismatch}$ = ″The first mismatch in query responses is due to **Insert**(x_i)″.

We have

$$\mathsf{Pr}[\mathbf{E''}] \leq \sum_{i=1}^{q_{in}} \mathsf{Pr}[\mathbf{E}_{insert_i}^{mismatch} \text{ and } a_i^{Ideal} = \top \cap\ a_i^{G*} = \perp] \ +\ \mathsf{Pr}[\mathbf{E}_{insert_i}^{mismatch} \text{ and } a_i^{Ideal} = \perp \cap\, a_i^{G*} = \top]$$

$$\leq \sum_{i=1}^{q_{in}} \mathsf{Pr}[\mathbf{E}_{insert_i}^{mismatch} \text{ and } a_i^{Ideal} = \perp] + \mathsf{Pr}[\mathbf{E}_{insert_i}^{mismatch} \text{ and } a_i^{G*} = \perp]$$

$$= \sum_{i=1}^{q_{in}} \mathsf{Pr}[\mathbf{E}_{insert_i}^{Ideal}] + \mathsf{Pr}[\mathbf{E}_{insert_i}^{G*}], \tag{5.8}$$

where we define the event $\mathbf{E}_{insert_i}^{Ideal}$ as

$$\mathbf{E}_{insert_i}^{Ideal} = [\mathbf{E}_{insert_i}^{mismatch} \text{ and } a_i^{Ideal} = \perp]$$

and the event $\mathbf{E}^{G*}_{insert_i}$ as

$$\mathbf{E}^{G*}_{insert_i} = [\mathbf{E}^{mismatch}_{insert_i} \text{ and } a_i^{G*} = \perp].$$

We focus now on $\mathbf{E}^{Ideal}_{insert_i}$. First, by definition and using the conditional probability over all $\mathcal{A}'s$ and *Ideal's coins* , we have

$$\mathsf{Pr}[\mathbf{E}^{Ideal}_{insert_i}] = \underset{\substack{Ideal's\ coins \\ r \leftarrow\$ R \\ \mathcal{A}'s\ coins}}{\mathsf{Pr}}[\mathbf{E}^{mismatch}_{insert_i} \cap ((a,\sigma) \leftarrow \text{Insert}^{Id_\Re}(Y \xleftarrow{r} \Re, \sigma^{(i)}) : a = \perp)]$$

$$= \sum_{\mathcal{A}'s\ coins} \underset{\substack{Ideal's\ coins \\ r \leftarrow\$ R}}{\mathsf{Pr}} \left[\mathbf{E}^{mismatch}_{insert_i} \cap ((a,\sigma) \leftarrow \text{Insert}^{Id_\Re}(Y \xleftarrow{r} \Re, \sigma^{(i)}) : a = \perp) \mid \mathcal{A}'s\ coins \right]$$
$$\cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$= \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \underset{r \leftarrow\$ R}{\mathsf{Pr}} \left[\mathbf{E}^{mismatch}_{insert_i} \cap ((a,\sigma) \leftarrow \text{Insert}^{Id_\Re}(Y \xleftarrow{r} \Re, \sigma^{(i)}) : a = \perp) \mid \begin{matrix} \mathcal{A}'s\ coins \\ Ideal'scoins \end{matrix} \right]$$
$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins] \tag{5.9}$$

Now from the previous line, we need to compute

$$\underset{r \leftarrow\$ R}{\mathsf{Pr}}[\mathbf{E}^{mismatch}_{insert_i} \cap ((a,\sigma) \leftarrow \text{Insert}^{Id_\Re}(Y \xleftarrow{r} \Re, \sigma^{(i)}) : a = \perp) | \mathcal{A}'s \text{ and } Ideal's\ coins].$$

Let $\Delta_{max,i}$ represent the state σ after all q_{in} insertions but without the ith insertion. Each element was inserted using the $insert^{Id_\Re}_{partial}$ function. Since $\Delta_{max,i}$ contains all insertions except the ith one, $\sigma^{(i)} \subseteq \Delta_{max,i}$ and we use the non-decreasing disabling probability for insertion:

$$\underset{r \leftarrow\$ R}{\mathsf{Pr}}[\mathbf{E}^{mismatch}_{insert_i} \cap ((a,\sigma) \leftarrow \text{Insert}^{Id_\Re}(Y \xleftarrow{r} \Re, \sigma^{(i)}) : a = \perp) | \mathcal{A}'s \text{ and } Ideal's\ coins]$$

$$\leq \underset{r \leftarrow\$ R}{\mathsf{Pr}}[\mathbf{E}^{mismatch}_{insert_i} \cap ((a,\sigma) \leftarrow \text{Insert}^{Id_\Re}(Y \xleftarrow{r} \Re, \Delta_{max,i}) : a = \perp) | \mathcal{A}'s \text{ and } Ideal's\ coins]$$

Since in $\Delta_{max,i}$ all elements are inserted and mapped to counters selected uniformly and independently at random, the adversary has no power and the probability does not depend on $\mathcal{A}'s$ coins anymore:

$$\begin{aligned}
&\Pr_{r\leftarrow\$R}[\mathbf{E}^{mismatch}_{insert_i} \cap ((a,\sigma) \leftarrow \text{Insert}^{Id_\Re}(Y \xleftarrow{r} \Re, \Delta_{max,i}) : a = \perp) | \mathcal{A}'s \text{ and } Ideal'\text{s coins}] \\
&= \Pr_{r\leftarrow\$R}[\mathbf{E}^{mismatch}_{insert_i} \cap ((a,\sigma) \leftarrow \text{Insert}^{Id_\Re}(Y \xleftarrow{r} \Re, \Delta_{max,i}) : a = \perp) | Ideal'\text{s coins}] \\
&\leq \Pr_{r\leftarrow\$R}[((a,\sigma) \leftarrow \text{Insert}^{Id_\Re}(Y \xleftarrow{r} \Re, \Delta_{max,i}) : a = \perp) | Ideal'\text{s coins}] \qquad (5.10)
\end{aligned}$$

We combine now Equations 5.9 and 5.10:

$$\begin{aligned}
\Pr[\mathbf{E}^{Ideal}_{insert_i}] \leq &\sum_{\mathcal{A}'s \text{ coins}} \sum_{Ideal'\text{s coins}} \Pr_{r\leftarrow\$R}[(a,\sigma) \leftarrow \text{Insert}^{Id_\Re}(Y \xleftarrow{r} \Re, \Delta_{max,i}) : a = \perp \, | Ideal'\text{s coins}] \\
&\cdot \Pr[Ideal'\text{s coins}] \cdot \Pr[\mathcal{A}'s \text{ coins}]
\end{aligned}$$

Reversing the conditional probability rule we get:

$$\begin{aligned}
\Pr[\mathbf{E}^{Ideal}_{insert_i}] &\leq \sum_{\mathcal{A}'s \text{ coins}} \Pr_{\substack{Ideal'\text{s coins} \\ r\leftarrow\$R}}[(a,\sigma) \leftarrow \text{Insert}^{Id_\Re}(Y \xleftarrow{r} \Re, \Delta_{max,i}) : a = \perp] \cdot \Pr[\mathcal{A}'s \text{ coins}] \\
&= \Pr_{\substack{Ideal'\text{s coins} \\ r\leftarrow\$R}}[(a,\sigma) \leftarrow \text{Insert}^{Id_\Re}(Y \xleftarrow{r} \Re, \Delta_{max,i}) : a = \perp] \cdot \sum_{\mathcal{A}'s \text{ coins}} \cdot \Pr[\mathcal{A}'s \text{ coins}] \\
&= \Pr_{\substack{Ideal'\text{s coins} \\ r\leftarrow\$R}}[(a,\sigma) \leftarrow \text{Insert}^{Id_\Re}(Y \xleftarrow{r} \Re, \Delta_{max,i}) : a = \perp] \\
&\leq \Pr\left[\begin{array}{c} Y_1,..,Y_{q_{in}-1} \leftarrow\$ \Re, \\ \Delta \leftarrow\$ setup(pp), \\ (\top,\Delta) \leftarrow insert^{Id_\Re}_{partial}(Y_i,\Delta) \; for \; i \in [1,..,q_{in}-1] \\ (a,\Delta') \leftarrow insert^{Id_\Re}(Y \leftarrow\$ \Re, \Delta) \\ : a = \perp \end{array}\right]
\end{aligned}$$

$$= \mathsf{Pr}\left[\begin{array}{c} Y_1, .., Y_{q_{in}-1} \leftarrow\!\$\ \Re, \\ \Delta \leftarrow\!\$\ setup(pp), \\ (\top, \Delta) \leftarrow insert^{Id_\Re}_{partial}(Y_i, \Delta)\ for\ i \in [1, .., q_{in}-1] \\ (a, \Delta') \leftarrow insert^{Id_\Re}(Y \leftarrow\!\$\ \Re, \Delta) \\ : \text{at least one counter needed for insertion is} \geq maxValue \end{array}\right]$$

$$\leq \mathsf{Pr}\left[\begin{array}{c} Y_1, .., Y_{q_{in}-1} \leftarrow\!\$\ \Re, \\ \Delta \leftarrow\!\$\ setup(pp), \\ (\top, \Delta) \leftarrow insert^{Id_\Re}_{partial}(Y_i, \Delta)\ for\ i \in [1, .., q_{in}-1] \\ (a, \Delta') \leftarrow insert^{Id_\Re}(Y \leftarrow\!\$\ \Re, \Delta) \\ : \text{any counter in } \Delta \text{ is } \geq maxValue \end{array}\right]$$

$$= \mathsf{Pr}\left[\begin{array}{c} Y_1, .., Y_{q_{in}-1} \leftarrow\!\$\ \Re, \\ \Delta \leftarrow\!\$\ setup(pp), \\ (\top, \Delta) \leftarrow insert^{Id_\Re}_{partial}(Y_i, \Delta)\ for\ i \in [1, .., q_{in}-1] \\ : \text{any counter in } \Delta \text{ is } \geq maxValue \end{array}\right]$$

In this last step, instead of considering only counters needed for insertion, we consider all counters in the filter.

Since we consider all q_{in} insertions but the ith one, we observe that Δ corresponds to the non-adversarially-influenced state defined in Definition 2.5 with $q_{in} - 1$ elements:

$$\mathsf{Pr}[\mathbf{E}^{Ideal}_{qry_i}] \leq \mathsf{Pr}\left[\begin{array}{c} F \leftarrow\!\$\ \mathrm{Funcs}[\mathfrak{D}, \mathfrak{R}], \\ [\mathsf{x}_1, ..., \mathsf{x}_{\mathsf{q_{in}}-1}] \leftarrow\!\$\ U(S \in \mathsf{P_{lists}}(\mathfrak{D}) || S| = q_{in} - 1) \\ \Delta \leftarrow\!\$\ \mathrm{setup}(pp) \\ (b, \Delta) \leftarrow\!\$\ insert^F_{partial}(x_k, \Delta)\ for\ k = 1, ..., q_{in} - 1 \\ : \text{any counter in } \Delta \text{ is } \geq maxValue \end{array}\right]$$

Applying Definition 2.7 which gives the probability that any counter is greater than or equal to $maxValue$ in an NAI state, we get:

$$\mathsf{Pr}[\mathbf{E}^{Ideal}_{insert_i}] \leq m \cdot (\frac{e \cdot (q_{in} - 1) \cdot k}{maxValue \cdot m})^{maxValue}. \tag{5.11}$$

We focus now on $\mathbf{E}^{G*}_{insert_i}$ at the time of the insertion of element x_i. Since the majority of the steps are similar to the computation for $\mathbf{E}^{Ideal}_{insert_i}$, we give explanations only when they differ. First, we have

$$\mathsf{Pr}[\mathbf{E}^{G*}_{insert_i}] = \underset{\substack{G*'s\ coins \\ Ideal's\ coins \\ \mathcal{A}'s\ coins}}{\mathsf{Pr}} [\mathbf{E}^{mismatch}_i \cap ((a, \sigma') \leftarrow \mathrm{Insert}^F(x_i, \sigma^{(i)'}) : a = \perp)]$$

$$= \sum_{\mathcal{A}'s\ coins} \underset{\substack{G*'s\ coins \\ Ideal's\ coins}}{\mathsf{Pr}} \left[\begin{array}{c} \mathbf{E}^{mismatch}_{insert_i} \\ \cap ((a, \sigma') \leftarrow \mathrm{Insert}^F(x_i, \sigma^{(i)'}) : a = \perp) \end{array} \middle|\ \mathcal{A}'s\ coins\right]$$
$$\cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$= \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \underset{G*'s\ coins}{\mathsf{Pr}} \left[\begin{array}{c} \mathbf{E}^{mismatch}_{insert_i} \\ \cap ((a, \sigma') \leftarrow \mathrm{Insert}^F(x_i, \sigma^{(i)'}) : a = \perp) \end{array} \middle|\ \begin{array}{c} \mathcal{A}'s\ coins \\ Ideal's coins \end{array}\right]$$
$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins].$$

Let $\Omega_{max,i}$ represent the state σ' containing all distinct inserted elements $\{x_1, ..., x_{q_{in}}\} \backslash \{x_i\}$ where each element was inserted using the $insert^{Id_{\mathfrak{R}}}_{partial}$ function. Since $\Omega_{max,i}$ contains all inserted elements except the current x_i, $\sigma^{(i)'} \subseteq \Omega_{max,i}$ and we use the **non-decreasing disabling probability for insertion** consistency rule:

$$\mathsf{Pr}[\mathbf{E}^{G*}_{insert_i}] \leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \mathsf{Pr}_{G*'s\ coins} \left[\begin{array}{c} \mathbf{E}^{mismatch}_{insert_i} \\ \cap\ ((a, \sigma') \leftarrow \mathrm{Insert}^F(x_i, \Omega_{max,i}) : a = \perp) \end{array} \mid \begin{array}{c} \mathcal{A}'s\ coins \\ Ideal'scoins \end{array} \right]$$
$$\cdot\, \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$\leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \mathsf{Pr}_{G*'s\ coins} \left[((a, \sigma') \leftarrow \mathrm{Insert}^F(x_i, \Omega_{max,i}) : a = \perp) \mid \begin{array}{c} \mathcal{A}'s\ coins \\ Ideal'scoins \end{array} \right]$$
$$\cdot\, \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$= \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \cdot \mathsf{Pr} \left[\begin{array}{c} F \leftarrow\!\$\ \mathrm{Funcs}[\mathfrak{D}, \mathfrak{R}], \\ \Omega \leftarrow\!\$\ setup(pp), \\ \text{for each } x_k \in \{x_1, .., x_{q_{in}}\} \backslash \{x_i\}, \\ (\top, \Omega) \leftarrow insert^F_{partial}(x_k, \Omega) \\ (a, \Omega) \leftarrow insert^F(x_i, \Omega) \\ : a = \perp \end{array} \right]$$
$$\cdot\, \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

Since Counting filters are function-decomposable (section 2.3.1), we can rewrite:

$$\mathsf{Pr}[\mathbf{E}^{G*}_{insert_i}] \leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \cdot \mathsf{Pr}\left[\begin{array}{c} F \leftarrow\!\!\$\ \mathrm{Funcs}[\mathfrak{D},\mathfrak{R}], \\ \Omega \leftarrow\!\!\$\ setup(pp), \\ \text{for each } x_k \in \{x_1,..,x_{q_{in}}\}\backslash\{x_i\}, \\ (\top,\Omega) \leftarrow insert^{Id_{\mathfrak{R}}}_{partial}(F(x_k),\Omega) \\ (a,\Omega) \leftarrow insert^{Id_{\mathfrak{R}}}(F(x_i),\Omega) \\ : a = \bot \end{array}\right]$$

$$\cdot\, \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

As we consider all q_{in} insertions except the current element x_i, the state contains $|\{x_1,..,x_{q_{in}}\}\backslash\{x_i\}|$ elements. Since F is a random function and using the **non-decreasing disabling probability for insertion** consistency rule:

$$\mathsf{Pr}[\mathbf{E}^{G*}_{insert_i}] \leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins}$$

$$\mathsf{Pr}\left[\begin{array}{c} Y_1,..,Y_{|\{x_1,..,x_{q_{in}}\}\backslash\{x_i\}|} \leftarrow\!\!\$\ \mathfrak{R}, \\ \Omega \leftarrow\!\!\$\ setup(pp), \\ (\top,\Omega) \leftarrow insert^{Id_{\mathfrak{R}}}_{partial}(Y_k,\Omega)\ for\ k \in [1,..,q_{|\{x_1,..,x_{q_{in}}\}\backslash\{x_i\}|}] \\ (a,\Omega') \leftarrow insert^{Id_{\mathfrak{R}}}(Y \leftarrow\!\!\$\ \mathfrak{R},\Omega) \\ : a = \bot \end{array}\right]$$

$$\cdot\, \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$\leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \cdot \mathsf{Pr}\left[\begin{array}{c} Y_1, .., Y_{q_{in}-1} \leftarrow\$\ \Re, \\ \Omega \leftarrow\$\ setup(pp), \\ (\top, \Omega) \leftarrow insert^{Id_\Re}_{partial}(Y_k, \Omega)\ for\ k \in [1, .., q_{in}-1] \\ (a, \Omega') \leftarrow insert^{Id_\Re}(Y \leftarrow\$\ \Re, \Omega) \\ : a = \perp \end{array}\right]$$

$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$= \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \cdot \mathsf{Pr}\left[\begin{array}{c} Y_1, .., Y_{q_{in}-1} \leftarrow\$\ \Re, \\ \Omega \leftarrow\$\ setup(pp), \\ (\top, \Omega) \leftarrow insert^{Id_\Re}_{partial}(Y_k, \Omega)\ for\ k \in [1, .., q_{in}-1] \\ (a, \Omega') \leftarrow insert^{Id_\Re}(Y \leftarrow\$\ \Re, \Omega) \\ \text{: at least one counters needed for insertion} \\ \text{is} \geq maxValue \end{array}\right]$$

$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

$$\leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \cdot \mathsf{Pr}\left[\begin{array}{c} Y_1, .., Y_{q_{in}-1} \leftarrow\$\ \Re, \\ \Omega \leftarrow\$\ setup(pp), \\ (\top, \Omega) \leftarrow insert^{Id_\Re}_{partial}(Y_k, \Omega)\ for\ k \in [1, .., q_{in}-1] \\ (a, \Omega') \leftarrow insert^{Id_\Re}(Y \leftarrow\$\ \Re, \Omega) \\ \text{: any counter in } \Omega \text{ is} \geq maxValue \end{array}\right]$$

$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins]$$

Since we consider all q_{in} insertions but the ith one, we observe that Ω corresponds to the non-adversarially-influenced state as in Definition 2.5 with $q_{in} - 1$ elements:

$$\mathsf{Pr}[\mathbf{E}^{G*}_{insert_i}] \leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins}$$

$$\cdot \mathsf{Pr}\left[\begin{array}{c} F \leftarrow\$ \text{ Funcs}[\mathfrak{D}, \mathfrak{R}], \\ [\mathsf{x}_1, ..., \mathsf{x}_{\mathsf{q_{in}}-1}] \leftarrow\$ U(S \in \mathsf{P}_{\mathsf{lists}}(\mathfrak{D}) || S| = q_{in} - 1) \\ \Omega \leftarrow\$ \text{setup}(pp) \\ (b, \Omega) \leftarrow\$ insert^{F}_{partial}(x_k, \Omega)\ for\ k = 1, ..., q_{in} - 1 \\ : \text{any counter in } \Omega \text{ is } \geq maxValue \end{array}\right]$$

$$\cdot \mathsf{Pr}[Ideal's\ coins | \mathcal{A}'s\ coins] \cdot \mathsf{Pr}[\mathcal{A}'s\ coins].$$

We apply Equation 2.4 from Definition 2.7 which gives the probability that any counter is greater than or equal to $maxValue$ in an NAI state:

$$\mathsf{Pr}[\mathbf{E}^{G*}_{insert_i}] \leq \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} m \cdot (\frac{e \cdot (q_{in} - 1) \cdot k}{maxValue \cdot m})^{maxValue} \cdot \mathsf{Pr}[Ideal's\ coins\ \cap\ \mathcal{A}'s\ coins]$$

$$= m \cdot (\frac{e \cdot (q_{in} - 1) \cdot k}{maxValue \cdot m})^{maxValue} \cdot \sum_{\mathcal{A}'s\ coins} \sum_{Ideal's\ coins} \mathsf{Pr}[Ideal's\ coins\ \cap\ \mathcal{A}'s\ coins]$$

$$= m \cdot (\frac{e \cdot (q_{in} - 1) \cdot k}{maxValue \cdot m})^{maxValue}. \tag{5.12}$$

Finally, we combine Equations 5.8, 5.11 and 5.12, and we have:

$$\begin{aligned}
\mathsf{Pr}[\mathbf{E''}] &\leq \sum_{i=1}^{q_{in}} \mathsf{Pr}[\mathbf{E}^{Ideal}_{insert_i}] + \mathsf{Pr}[\mathbf{E}^{G*}_{insert_i}] \\
&\leq q_{in} \cdot [m \cdot (\frac{e \cdot (q_{in}-1) \cdot k}{maxValue \cdot m})^{maxValue} + m \cdot (\frac{e \cdot (q_{in}-1) \cdot k}{maxValue \cdot m})^{maxValue}] \\
&= 2 \cdot q_{in} \cdot m \cdot (\frac{e \cdot (q_{in}-1) \cdot k}{maxValue \cdot m})^{maxValue}.
\end{aligned}$$

□

We finish the proof of Theorem 5.2. Combining Lemmas 5.3 - 5.8,

$$\begin{aligned}
\mathsf{Adv}^{\Re o \Im}_{\Pi,\mathcal{A},\mathcal{S}}(\mathcal{D}) &= |\mathsf{Pr}[Real(\mathcal{A},\mathcal{D})=1] - \mathsf{Pr}[Ideal(\mathcal{A},\mathcal{D},\mathcal{S})=1]| \\
&\leq |\mathsf{Pr}[Real(\mathcal{A},\mathcal{D})=1] - \mathsf{Pr}[G(\mathcal{A},\mathcal{D})=1]| + \\
&|\mathsf{Pr}[G(\mathcal{A},\mathcal{D})=1] - \mathsf{Pr}[Ideal(\mathcal{A},\mathcal{D},\mathcal{S})=1]| \\
&= \mathsf{Adv}^{\text{Real}-\text{or}-\text{G}}_{\Pi,A,S}(\mathcal{D}) + \mathsf{Adv}^{\text{G}-\text{or}-\text{Ideal}}_{\Pi,\mathcal{A},\mathcal{S}}(\mathcal{D}) \\
&\leq \epsilon + \mathsf{Pr}[\mathbf{E}] + \mathsf{Pr}[\mathbf{E'}] + \mathsf{Pr}[\mathbf{E''}] \\
&\leq \epsilon + 2 \cdot q_{in} \cdot m \cdot (\frac{e \cdot (q_{in}-1) \cdot k}{maxValue \cdot m})^{maxValue} + (2 \cdot q_{qry} + q_{del}) \cdot Pr_{NAI_{\Pi,pp}}(FP|q_{in}).
\end{aligned}$$

□

Chapter 6

Conclusion

In this work, we analysed Counting filters in adversarial environments. We investigated attacks with different adversarial goals on Counting filters. We defined how to achieve them, regarding the different oracles the adversary has access to. We used an existing simulation-based framework to analyse the adversarial correctness of insertion-only Counting filters, and we extended it for Counting filters with deletions. These frameworks use a simulation-based approach with the construction of a simulator enforcing constraints on the filter. We derived bounds on the correctness of insertion-only Counting filters and Counting filters with deletions in order to demonstrate the limits of the adversaries' abilities. These bound show the limits of any adversary's capability to perform pollution attacks and target-set coverage attacks on Counting filters.

We propose possible extensions of our work.

- Provide an analysis of the attacks described in Section 3.3.
- Provide an analysis of the tightness of bounds derived in Chapters 4 and 5, and investigate if they could be made tighter.
- Compare our bounds derived in the simulation-based framework with the bounds derived in the game-based setting used by Clayton et. al. [8].
- Extend our analysis to allow reinsertion in Counting filters with deletions. This requires extension of the syntax and modification of the consistency rules.
- Extend our framework to AMQ-PDS such as Cuckoo filters.
- Extend our framework to PDS such as Count-Min Sketches analyzed by Clayton et. al. in [8]. This would also require modification of the syntax and consistency rules.

Appendix A

Appendix

A.1 Showing that Ideal is not NAI

In this section we propose a manipulation that the adversary can operate on the filter in the ideal world. This shows that Ideal is not NAI.

First, the adversary computes the false positive of the current state of the filter FP. Then, the adversary inserts a random element x. Due to the definition of insertions in *Ideal* (line 4 of InsertSim algorithm in Figure 5.3), a random element $x' \leftarrow\!\!\$\ \Re$ is mapped into the filter. After insertion, and if the insertion succeeds, the adversary computes the new false positive probability FP'. By comparing both false positive values FP and FP', the adversary is able decide if he is satisfied, and if not he deletes the inserted element. For example, the adversary is able to mount an attack enforcing that each insertion satisfies $FP' - FP > \epsilon$.
We note that the adversary cannot decide later after deletion since any reinsertion of x is going to insert a new random element $x'' \leftarrow\!\!\$\ \Re$.

The distribution of the filter differs from the uniformly random distrbution. Indeed, enforcing $FP' - FP > \epsilon$ corresponds to enforcing that a minimum ϵ' of counters are incremented to 1 during the insertion, which is not the case when using uniformly random insertions. Indeed, when using uniformly random distribution each counter is chosen with probability $\frac{1}{m}$, independently of if its current value.

We showed that the filter in Ideal is not equivalent to a filter generated with uniformly random mapping. Therefore it does no corresponds to an NAI state defined in section 2.3.

A.2 Remark on attack in Section 3.4

We note that the attack performed in Section 3.4 can be simulated having access to a **Query** query instead of having access to a **Delete** query.

Indeed, before insertion the attacker can query an element. If the query returns "*not in the filter*", the adversary inserts the element in the filter and new counters are set > 0. Otherwise, he does not insert the element as it means that the element is already considered in the filter and and all its corresponding counters are already > 0. However, this method allows less freedom. Using the **Query** query, we are not able to know how many new counters are set and thus this variable cannot be controlled as in our attack.

It is trivial to see that if this attack is run using the **Query** query, this attack is feasible in both **public** and **private** setting, and without deletions. This differs from our implemented attack which needs a public setting with deletions.

This last point is taken in account in the security bound derived in Chapter 4 insertion-only public Counting filters, and in Chapter 5 private Counting filters with deletions.

A.3 Side-channel attacks on private Counting filters

Recently, some side-channel attacks were performed on private filters using PRF. Eventhough side-channel attacks are not considered in cryptographic security bounds, we mention them for awareness.

Reviriego and Rottenstreich [26] proposed two pollution attacks that do not require any knowledge of the counting filter implementation details, and where the attacker is allowed access to the insert, delete, and reveal oracles in Figure 2.3.

The first attack is called "lookup time attack". This attacks utilizes the time to complete a lookup in a sequential implementation, to identify elements that pollute the filter.

Indeed, in an implementation where counters are accessed sequentially, the lookup time depends on the number of elements accessed in the filter. If an element is in the filer, all counter's positions are accessed in the filter during the lookup. If an element is not part of the filter, the number of positions accessed during the lookup depends on when the first corresponding counter with value equal to zero is found. In order to increase the number of non-zero counters in the filter inserting only one element, an attacker can first do a lookup for a set of elements, and then give as input to the filter the one element with the lowest lookup time.

However, for this attack to be achievable, the lookup time has to depend on the number of accessed counters in the filter, such that a query with more accesses takes more time than a query with less accesses. The different counters must be accessed sequentially, which is not always the case. For example, if the filter is implemented on a system that uses a memory hierarchy, some counters might be in the cache while others are not. Similarly, if the filter is implemented in parallel where each counter is mapped to a different memory, all memories are accessed at the same time and by consequence lookup operations are always completed in a constant time.

The second attack is called ”delta on false positive probability attack”. The insertion of an element in the filter increases the false positive probability by an amount that depends on the number of bits set to one by the insertion (i.e. new non zero counters). By consequence, an attacker might be able to infer the number of counters set to one after the insertion of an element by observing the fpp and behave accordingly.

This scheme achieves a good pollution on the filter, with a deterministic execution time and number of operations on the filter.

The different evaluations of these attacks show that the ”delta fpp attack” is more effective than the ”lookup time based attack”.

These two attacks can be considered as side-channel attack and therefore, do not need to be considered in security bounds.

Bibliography

[1] Tony Allen. Counting filter implementation. https://github.com/tonya11en/bloom_filter/blob/master/counting_filter.cc, 2017. Last accessed 08 September 2022.

[2] Gianni Antichi, Domenico Ficara, Stefano Giordano, Gregorio Procissi, and Fabio Vitucci. Counting bloom filters for pattern matching and anti-evasion at the wire speed. *IEEE Network*, 23(1):30–35, 2009.

[3] Markku Antikainen, Tuomas Aura, and Mikko Särelä. Denial-of-service attacks in bloom-filter-based forwarding. *IEEE/ACM Transactions on Networking*, 22(5):1463–1476, 2014.

[4] Apache Software Foundation (ASF). Introduction to probabilistic data structures. https://hadoop.apache.org/docs/r2.7.2/api/src-html/org/apache/hadoop/util/bloom/BloomFilter.html, 2015. Last accessed 08 September 2022.

[5] Daniel Baker and Ben Langmead. Dashing: fast and accurate genomic distances with hyperloglog. *Genome Biology*, 20, 12 2019.

[6] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, jul 1970.

[7] Andrei Broder and Michael Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1, 11 2003.

[8] David Clayton, Christopher Patton, and Thomas Shrimpton. Probabilistic data structures in adversarial environments. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[9] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55:29–38, 2004.

[10] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security Symposium*, 2003.

[11] S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor. Longest prefix matching using bloom filters. *IEEE/ACM Transactions on Networking*, 14(2):397–409, 2006.

[12] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. CoNEXT '14, page 75–88, New York, NY, USA, 2014. Association for Computing Machinery.

[13] Li Fan, Pei Cao, J. Almeida, and A.Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[14] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *IN AOFA '07: PROCEEDINGS OF THE 2007 INTERNATIONAL CONFERENCE ON ANALYSIS OF ALGORITHMS*, 2007.

[15] V. Fuller, T. Li, J. Yu, and K. Varadhan. Rfc1519: Classless inter-domain routing (cidr): An address assignment and aggregation strategy, 1993.

[16] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. The power of evil choices in bloom filters. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 101–112, 2015.

[17] Deke Guo, Yunhao Liu, Xiangyang Li, and Panlong Yang. False negative problem of counting bloom filter. *IEEE Transactions on Knowledge and Data Engineering*, 22(5):651–664, 2010.

[18] Kibeom Kim, Yongjo Jeong, Youngjoo Lee, and Sunggu Lee. Analysis of counting bloom filters used for count thresholding. *Electronics*, 8:779, 07 2019.

[19] Ella Kummer. Master thesis implementation attack. https://gitlab.ethz.ch/ekummer/master-thesis, 2022.

[20] Lipton and Naughton. Clocked adversaries for hashing. *Algorithmica*, 9, 1993.

[21] Anupama Unnikrishnan Mia Filić, Kenneth G. Paterson and Fernando Virdia. Adversarial correctness and privacy for probabilistic data structures. 2022.

[22] Moni Naor and Yogev Eylon. Bloom filters in adversarial environments. *ACM Trans. Algorithms*, 15(3), jun 2019.

[23] Yin Niu. Introduction to probabilistic data structures. https://dzone.com/articles/introduction-probabilistic-0, 2015. Last accessed 01 September 2022.

[24] 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.

[25] Pedro Reviriego, Jorge Martinez, David Larrabeiti, and S. Pontarelli. Cuckoo filters and bloom filters: Comparison and application to packet classification. *IEEE Transactions on Network and Service Management*, PP:1–1, 09 2020.

[26] Pedro Reviriego and Ori Rottenstreich. Pollution attacks on counting bloom filters for black box adversaries. *2020 16th International Conference on Network and Service Management (CNSM)*, pages 1–7, 2020.

[27] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys Tutorials*, 14(1):131–155, 2012.

[28] Tyler Treat. Counting filter implementation. https://github.com/tylertreat/BoomFilters/blob/master/counting.go, 2014. Last accessed 08 September 2022.

[29] Belma Turkovic, Jorik Oostenbrink, and Fernando Kuipers. Detecting heavy hitters in the data-plane. 02 2019.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.