Solving Set Relations with Secure Bloom Filters Keeping Cardinality Private

Louis Tajan[1], Dirk Westhoff[1] and Frederik Armknecht[2]
[1]*Hochschule Offenburg, Offenburg, Germany*
[2]*University of Mannheim, Mannheim, Germany*

Keywords: Bloom Filters, Set Operations, Set Relations, Outsourced Computation.

Abstract: We propose in this work to solve privacy preserving set relations performed by a third party in an outsourced configuration. We argue that solving the disjointness relation based on Bloom filters is a new contribution in particular by having another layer of privacy on the sets cardinality. We propose to compose the set relations in a slightly different way by applying a keyed hash function. Besides discussing the correctness of the set relations, we analyze how this impacts the privacy of the sets content as well as providing privacy on the sets cardinality. We are in particular interested in how having bits overlapping in the Bloom filters impacts the privacy level of our approach. Finally, we present our results with real-world parameters in two concrete scenarios.

1 INTRODUCTION

In the work at hand, we propose a protocol to solve what we call the *private outsourced disjointness test*. We suppose two parties, let it be Alice and Bob each owning a dataset of elements, respectively $\mathcal{A}$ and $\mathcal{B}$. They would like to know if their datasets are disjoint, i.e. if $\mathcal{A} \cap \mathcal{B} = \emptyset$. To do so, they will ask a third party, *Server* to perform the verification. Alice (resp. Bob) does not want to reveal any information on her dataset including its size to any other party.

We propose a solution based on the Bloom filter (Bloom, 1970) representation along with keyed hash functions as HMAC. We also propose to apply this approach to a more straightforward set relation, the *private outsourced inclusiveness test*. Bloom filters are space-efficient data structures used to represent sets and that allow to perform set membership checks. One may argue that a simple pseudonymization could be sufficient for the above sketched scenario, as solely apply a keyed hash function on the sets (Churches and Christen, 2004). However, even if the pseudonymization function remains private to any other party than the Bloom filter owners, one may directly gain knowledge on the number of common elements of two Bloom filters. Such a naive approach will also reveal which pseudonym is present in none, one or both sets. On the contrary, Bloom filter representation has the particular feature of adding obfuscation to the sets. We argue that contrary to multi-party based solutions (Kissner and Song, 2005), it has relevance if such a protocol class is non-interactive. For instance, it could be applied to scenarios of mobile users tracking (Tajan and Westhoff, 2019) or to cloud auditing use cases (Tajan et al., 2016) where a third party auditor should perform verification on logfiles and whitelists. The concrete contributions of our work are threefold. Firstly, we allow a third party entity to compute set relations namely, *inclusiveness* and *disjointness* in an outsourced model. To do so, we tune the Bloom filter approach by enhancing its privacy with respect to the sets content. Such an approach is also providing privacy on the sets cardinality. Secondly, we present an attack to gain the sequences cardinality in the present configuration. By analyzing the behavior of *overlapping bits* in the Bloom filter environment we show what amount of information such an attack may provide. Finally, we implemented our solution and present our results obtained for the concrete cloud security audit on access control use case with real-world parameters.

2 PRELIMINARIES

In this section we introduce the type of operations we are performing on sets and how we represent them using the Bloom filters approach.

Tajan, L., Westhoff, D. and Armknecht, F.
Solving Set Relations with Secure Bloom Filters Keeping Cardinality Private.
DOI: 10.5220/0009835904430450
In *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications (ICETE 2020) - SECRYPT*, pages 443-450
ISBN: 978-989-758-446-6

2.1 Set Operations and Relations

Multiple types of operations could be performed on sets but in this work, we aim to test set relations. Some of them could be reduced to compute the cardinality of some operations. For instance, being able to compute the cardinality of the intersection of two sets indicates whether they have elements in common or if one set is included in the other one. For privacy concerns, it could be of interest to solely reveal its cardinality instead of the intersection itself. Therefore, we propose a solution to solve two kinds of set relations namely the *inclusiveness* and the *disjointness*.

2.2 Bloom Filters

A Bloom filter is a data structure introduced by Burton Howard Bloom in 1970 (Bloom, 1970). It is used to represent a set of elements. With a Bloom filter representing a certain set, one can verify whether an element is a member of this set. Such a data structure consists of a tabular of m bits which is associated to n_{key} public hash functions. At first, all the m bits are initialized to 0. Moreover two functions namely *add()* and *test()* are available. To add an element to the Bloom filter, one has to compute the hashes of this element with each of the respective n_{key} hash functions. Then, set the bit to 1 for each position corresponding to a hash value. To test whether one element is included in the Bloom filter with the *test()* function, one has, by the same manner, to compute the respective hash values of this element and verify if the respective bits are set to 1. If at least one of these bits is set to 0, then with certainty the tested element is not a member of the set represented by the Bloom filter (i.e. no false negative for *test()* could happen). On the contrary, with some probability, the *test()* function could retrieve a false positive. Indeed, even if all the bits that have been verified are set to 1, the tested element may not be part of the set represented by the Bloom filter. To express the probability of having a false positive when performing function *test()* we introduce the notion of an *overlapping bit*.

Definition 1 (Overlapping Bit)**.** *When adding an element to a Bloom filter, a certain bit has to be set to 1 but this bit is already set to 1.*

The probability of having an overlapping bit is null when the Bloom filter is still blank, and it grows along with the number of inserted elements. We express this probability as following with $X_{BF_{\mathcal{A}}}$ the amount of bits already set to 1 in $BF_{\mathcal{A}}$ at a specific point in time:

$$\mathcal{P}_{ob} = \frac{X_{BF_{\mathcal{A}}}}{m} \tag{1}$$

We could then express the average amount of different bits added to the Bloom filter when adding one new element to it:

$$X_{add} = n_{key} + \sum_{i=1}^{n_{key}} \left((-1)^i \cdot \frac{\sum_{j=i}^{n_{key}-1} \binom{j}{i}}{m^i}\right) \tag{2}$$

And we could generalize it to the average amount of bits added to the Bloom filter when adding N new elements to it:

$$X_{add}(N) = n_{key} \cdot N + \sum_{i=1}^{n_{key} \cdot N} \left((-1)^i \cdot \frac{\sum_{j=i}^{(n_{key} \cdot N)-1} \binom{j}{i}}{m^i}\right) \tag{3}$$

By observing the current state of a Bloom filter representing finite set $\mathcal{A}$ of $n_{\mathcal{A}}$ elements, one can express the exact amount of overlapping bits as the value $Y_{BF_{\mathcal{A}}}$:

$$Y_{BF_{\mathcal{A}}} = (n_{\mathcal{A}} \cdot n_{key}) - X_{BF_{\mathcal{A}}} \tag{4}$$

2.3 Adversary Models

We describe in this section how the different parties should behave regarding the security of the protocol.

2.3.1 Alice and Bob - *Honest-but-Curious Adversaries*

We consider Alice and Bob acting normally according to the protocol description. Even if knowing any information on the other party's dataset could be of interest, we consider that Alice's and Bob's main objective is to retrieve the result of the sets relations and therefore they will faithfully provide correct inputs.

2.3.2 Server - *Malicious Adversary*

On the contrary, the third party is not trustworthy and may behave arbitrarily. Slightly differently than the classic definition of a *malicious adversary* (Goldreich, 2004), we consider here solely its privacy aspect. Indeed, we do not care about any alteration of the final result by the server. We consider that the server's objective is to gain any information on the sets from Alice and Bob and not to refuse or abort the protocol prematurely. Therefore, to find information on their datasets, the server (or any other malicious adversary) could try to generate its own Bloom filter and perform the set relations on it along with another Bloom filter from one of the parties.

3 RELATED WORK

To the best of our knowledge, this work is the first to solve the set disjointness relation in an outsourced

configuration using Bloom filter construction and providing privacy on the sets' sizes. Computing the disjointness test as a two-party secure computation problem has been proposed in several papers based on homomorphic encryption and Pedersen commitments for (Freedman et al., 2004; Kiayias and Mitrofanova, 2005), on "testable and homomorphic commitments" on polynomial representations for (Hohenberger and Weis, 2006) and Sylvester matrix and Lagrange interpolation for (Ye et al., 2008) but none of these are adapted to a third party scenario or provide privacy on the sets' sizes. Using Bloom filters in the objective to process operations and sets has been already been proposed. In addition to not providing any solution to private disjointness test, (Burkhart and fontas, 2012) proposes solution for multiparty computation while (Dong et al., 2013; Pinkas et al., 2014) for a two-party protocol. In (Kerschbaum, 2012) we notice that Kerschbaum proposes an "outsourced" version of his protocol but requires homomorphic multiplications between Bloom filters of encrypted elements. Also, the protocol disables the server to learn about the intersection size. Solving the set intersection cardinality implies solving the disjointness problem as in (Egert et al., 2015) but this work relies on knowing the sizes of the sets and does not fit our outsourced configuration. Adding privacy to Bloom filters has been investigated in several works. We explain in what way these existing solutions do not fit our requirements and therefore our solution brings novelty. First of all, we highlight the fact that none of the following solutions provide privacy on the sets cardinality. In (Goh, 2003) Goh associates Bloom filters with a keyed pseudo-random function to allow a private member testing in the Bloom filter. This is in particular one aspect we do not want the third party be able to do. In (Li and Gong, 2012) the authors expose a construction of Bloom filters along with HMAC protocol in a wireless sensor aggregation scenario. Their approach is somehow similar but the base station shares HMAC keys directly with each of the nodes. Therefore, the merging of Bloom filters from different nodes does not allow any operation since different keys are used. In (Qiu et al., 2007), still by combining the Bloom filter approach with a keyed hash function, the authors propose a solution to compute the membership of elements in a set. Therefore, they manipulate Bloom filters of unique elements that leads to data leakage regarding the amount of elements and could be very costly, especially when considering thousands of them.

4 PROTOCOL

We also emphasize the fact that, to process the two set relations, the considered Bloom filters should be similarly generated, namely with the same size m, keyed hash function and set of keys K. First we recall the two privacy enhancements from tuning the classical use of Bloom filter. Then we present the two set relations before explaining how the parameters should be selected to guarantee a certain level of correctness.

4.1 Privacy Enhancements

Our approach to make such a technique fitting for privacy-sensitive use cases, is based on the use of a public keyed collision-resistant hash function (e.g. MAC) with a set of n_{key} private keys instead of the n_{key} public hash functions. Without loss of generality, we use an HMAC function to solve the two set relations. That being said, any party that does not hold the keys cannot use the *test()* function to directly verify if a specific element is included in the Bloom filter. The other security benefit when using an HMAC function is that even if the function is publicly released, any party that does not hold the keys cannot add additional elements. More formally, we define a Bloom filter of a set $\mathcal{A} = \{a_1, \ldots, a_{n_\mathcal{A}}\}$ as a tabular of m bits, with a set of n_{key} keys $K = \{k_1, \ldots, k_{n_{key}}\}$ and an HMAC function $h_{k_\kappa}\{0,1\}^* \rightarrow \{1, \ldots, m\}$ with $k_\kappa \in K$ as:

$$BF\left(\mathcal{A}, (h_{k_\kappa})_{k_\kappa \in K}\right) = bf_\mathcal{A}[j]_{1 \leqslant j \leqslant m} \tag{5}$$

$$where \;\; bf_\mathcal{A}[j] = 1 \; if \; \exists \; (i,\kappa) \; s.t. \; h_{k_\kappa}(a_i) = j$$
$$bf_\mathcal{A}[j] = 0 \; otherwise$$

In the remaining parts of this work, we use the simplified notation $BF_\mathcal{A}$ (resp. $BF_\mathcal{B}$) to represent the Bloom filter of set $\mathcal{A}$ (resp. set $\mathcal{B}$). The second privacy enhancement we add to the use of Bloom filters corresponds to keep parameter n_{key} private to avoid revealing the sets' cardinalites. Indeed, the naive technique to retrieve the cardinality of the set by looking at its respective Bloom filter would be to divide its amount of bits set to one by parameter n_{key}. There exists an optimized technique introduced by Swamidass and Baldi (Swamidass and Baldi, 2007) which computes $n^*_\mathcal{A}$ an approximation of the number of distinct elements inserted in $BF_\mathcal{A}$ with $X_{BF_\mathcal{A}}$ the amount of bits set to 1 in the Bloom filter:

$$n^*_\mathcal{A} = -\frac{m}{n_{key}} ln\left[1 - \frac{X_{BF_\mathcal{A}}}{m}\right] \tag{6}$$

Such a technique requires even more overlapping bits to mislead the attacker. We argue that by making parameter n_{key} private, one could not be able to compute

$n^*_{\mathcal{A}}$ anymore. One may argue the complexity of keeping the size of K private or the effort to store a large amount of keys. We could then slightly modify the protocol to have a unique key k. Indeed, the outcome of $h_k(x)$ will be divided in n_{key} equal size fragments and each indicates an index of the Bloom filter to increment.

4.2 Initialization

$\mathbf{h}, \mathbf{n_{key}}, \mathbf{m}, \mathbf{K} \leftarrow$**Setup:** Alice should first choose and generate the Bloom filter parameters: the dimension m, the HMAC function h, the amount of keys n_{key} and the set of keys $K = \{k_1, \ldots, k_{n_{key}}\}$. She generates parameters by performing the following protocol:

- Randomly choose $n_{key} \in [n^L_{key}; n^U_{key}]$ with integers n_{key}, n^L_{key} and n^U_{key}.
- Set m such that $X_{\cap=\emptyset} < n^L_{key}$.

Values n^L_{key} and n^U_{key} are public and represent the value space of n_{key}. We determine them later considering correctness and privacy in Sections 4.5 and 5. The restriction on parameter m corresponds to a correctness consideration on $X_{\cap=\emptyset}$ which we explain in more details in Section 4.5.2. Then Alice selects the public HMAC function h, generates its n_{key} respective keys and privately shares parameters $\{h, n_{key}, m, K\}$ with Bob.

$\mathbf{BF}_{\mathcal{A}} \leftarrow$**Create($\mathcal{A}$):** Alice (resp. Bob) generates the Bloom filter of her dataset $\mathcal{A} = \{a_1, \ldots, a_{n_{\mathcal{A}}}\}$ (resp. $\mathcal{B} = \{b_1, \ldots, b_{n_{\mathcal{B}}}\}$):

$$BF_{\mathcal{A}} = BF\left(\mathcal{A}, (h_{k_\kappa})_{k_\kappa \in K}\right) = bf_{\mathcal{A}}[j]_{1 \leqslant j \leqslant m}$$
$$(resp.\ \ BF_{\mathcal{B}} = BF\left(\mathcal{B}, (h_{k_\kappa})_{k_\kappa \in K}\right) = bf_{\mathcal{B}}[j]_{1 \leqslant j \leqslant m})$$

4.3 Inclusiveness Protocol

This operator allows to verify if one set is included in another. It performs directly on the Bloom filters of the respective sets. To determine if $\mathcal{A}$ is included in $\mathcal{B}$ we define $\mathbf{BF}_{\mathcal{A}\subseteq\mathcal{B}} \leftarrow \mathbf{INC}(\mathbf{BF}_{\mathcal{A}}, \mathbf{BF}_{\mathcal{B}})$:

$$bf_{\mathcal{A}\subseteq\mathcal{B}}[j]_{1 \leqslant j \leqslant m} \leftarrow INC(BF_{\mathcal{A}}, BF_{\mathcal{B}}) \quad (7)$$
$$where\ \ 0 \leftarrow bf_{\mathcal{A}\subseteq\mathcal{B}}[j]\ if\ (bf_{\mathcal{A}}[j] = 1 \wedge bf_{\mathcal{B}}[j] = 0)$$
$$1 \leftarrow bf_{\mathcal{A}\subseteq\mathcal{B}}[j]\ otherwise.$$

We remark that this operator is equivalent to the bitwise binary operator combination:

$$INC(BF_{\mathcal{A}}, BF_{\mathcal{B}}) \equiv \neg(BF_{\mathcal{A}})\ OR\ \ BF_{\mathcal{B}} \quad (8)$$

Server firstly computes the inclusion protocol on the two respective Bloom filters of sets $\mathcal{A}$ and $\mathcal{B}$ to test if $\mathcal{A} \subseteq \mathcal{B}$, namely if all the elements from Alice's set are included in Bob's set:

$$INC(BF_{\mathcal{A}}, BF_{\mathcal{B}}) = BF_{\mathcal{A}\subseteq\mathcal{B}} = bf_{\mathcal{A}\subseteq\mathcal{B}}[j]_{1 \leqslant j \leqslant m}$$

Then *Server* expresses $X_{\mathcal{A}\subseteq\mathcal{B}}$ which corresponds to the number of bits set to 1 in the resulting Bloom filter:

$$X_{\mathcal{A}\subseteq\mathcal{B}} = \sum_{j=1}^{m} bf_{\mathcal{A}\subseteq\mathcal{B}}[j] \quad (9)$$

Server tests if $X_{\mathcal{A}\subseteq\mathcal{B}} = m$ and can conclude that $\mathcal{A} \subseteq \mathcal{B}$ if no false positive occurred. Otherwise we have $\mathcal{A} \nsubseteq \mathcal{B}$ with certainty.

4.4 Disjointness Protocol

This set relation allows to verify that no elements from one set are included in another set. In other words, this allows to claim that two sets are disjoint. This test function is not trivial, indeed, if we use Bloom filters it is not sufficient to highlight the cases where a bit 1 has been inserted at the same index for the two respective Bloom filters. We define this operator as $\mathbf{BF}_{\mathcal{A}\cap\mathcal{B}=\emptyset} \leftarrow \mathbf{DIS}(\mathbf{BF}_{\mathcal{A}}, \mathbf{BF}_{\mathcal{B}})$:

$$bf_{\mathcal{A}\cap\mathcal{B}=\emptyset}[j]_{1 \leqslant j \leqslant m} \leftarrow DIS(BF_{\mathcal{A}}, BF_{\mathcal{B}}) \quad (10)$$
$$where\ \ 1 \leftarrow bf_{\mathcal{A}\cap\mathcal{B}=\emptyset}[j]\ if\ (bf_{\mathcal{A}}[j] = 1 \wedge bf_{\mathcal{B}}[j] = 1)$$
$$0 \leftarrow bf_{\mathcal{A}\cap\mathcal{B}=\emptyset}[j]\ otherwise.$$

We remark that this operator is equivalent to the bitwise logical-and operator:

$$DIS(BF_{\mathcal{A}}, BF_{\mathcal{B}}) \equiv BF_{\mathcal{A}}\ AND\ BF_{\mathcal{B}}. \quad (11)$$

To verify that no element from Alice's dataset are included in Bob's one, *Server* performs the disjointness relation on the respective Bloom filters of $\mathcal{A}$ and $\mathcal{B}$:

$$DIS(BF_{\mathcal{A}}, BF_{\mathcal{B}}) = BF_{\mathcal{A}\cap\mathcal{B}=\emptyset} = \ bf_{\mathcal{A}\cap\mathcal{B}=\emptyset}[j]_{1 \leqslant j \leqslant m}$$

Then *Server* expresses $X_{\mathcal{A}\cap\mathcal{B}=\emptyset}$ which corresponds to the number of bits set to 1 in the resulting Bloom filter:

$$X_{\mathcal{A}\cap\mathcal{B}=\emptyset} = \sum_{j=1}^{m} bf_{\mathcal{A}\cap\mathcal{B}=\emptyset}[j] \quad (12)$$

Server compares it such that:
if $X_{\mathcal{A}\cap\mathcal{B}=\emptyset} < n^L_{key}$ then $\mathcal{A}$ and $\mathcal{B}$ are distinct
if $X_{\mathcal{A}\cap\mathcal{B}=\emptyset} \geqslant n^L_{key}$ then $\mathcal{A}$ and $\mathcal{B}$ have at least one element in common

Indeed for each element which is included in both sets, we get n_{key} times a bit set to 1 in the resulting Bloom filter. However we could still get such a bit set to 1 due to a bit set to 1 in $BF_{\mathcal{A}}$ and $BF_{\mathcal{B}}$ stemming from different elements originally added to the Bloom filters. We call such a case a false negative for the disjointness relation since the auditor will state that the sets are not disjoint while they are. We will discuss its probability of occurrence in the following sections.

4.5 Correctness of the Set Relations

In this section we consider the correctness of our two proposed relations. We recall that the Bloom filter approach allows false positives but no false negative on the *test()* function. Nevertheless, we focus on the overlapping bits of the Bloom filters resulting from our set relations.

4.5.1 Correctness of the Inclusiveness Relation

For the *inclusiveness* relation, we notice that only false positive could happen and not false negative. Indeed, after performing $INC(BF_{\mathcal{A}}, BF_{\mathcal{B}})$, if there is an index j with $bf_{\mathcal{A}\subseteq\mathcal{B}}[j] = 0$, we have $bf_{\mathcal{A}}[j] = 1$ and $bf_{\mathcal{B}}[j] = 0$, then with certainty, at least one element from $\mathcal{A}$ does not belong to $\mathcal{B}$. Concretely, if the outcome of the auditing process states that $\mathcal{A} \nsubseteq \mathcal{B}$ then we have a probability of correctness of 1. On the other hand, if we get $\mathcal{A} \subseteq \mathcal{B}$ as result, this outcome is not necessarily correct and we get a probability of correctness equals to $1 - \mathcal{P}_{FP}$ the probability of having a false positive. $\mathcal{P}_{FP}$ could be expressed in terms of parameters n_{key}, m and $n_{\mathcal{B}}$ denoting the amount of elements inserted in $BF_{\mathcal{B}}$. The probability that our *inclusiveness* relation outcomes a false positive whereas one element a_i from $\mathcal{A}$ is not in $\mathcal{B}$ is equivalent to the one to have $test(\mathcal{B}, a_i)$ resulting true with the same parameters. We detail the value of $\mathcal{P}_{FP}$:

First, we denote the probability that after inserting $n_{\mathcal{B}}$ elements, a certain bit is equal to 1 is:

$$1 - (1 - \frac{1}{m})^{n_{key} \cdot n_{\mathcal{B}}} \tag{13}$$

If we consider that $Z_{\mathcal{A},\mathcal{B}}$ elements from $\mathcal{A}$ are not included in $\mathcal{B}$, the probability of having a false positive after computing the *inclusiveness* relation is:

$$\mathcal{P}_{FP} \geqslant \left(1 - (1 - \frac{1}{m})^{n_{key} \cdot n_{\mathcal{B}}}\right)^{n_{key} \cdot Z_{\mathcal{A},\mathcal{B}}} \tag{14}$$

$$\simeq \left(1 - (1 - \frac{1}{m})^{X_{add}(n_{\mathcal{B}})}\right)^{X_{add}(Z_{\mathcal{A},\mathcal{B}})}$$

4.5.2 Correctness of the Disjointness Relation

For the *disjointness* relation, we have on the contrary no case of false positive but a case of false negative may happen. Indeed, if we get $X_{\mathcal{A}\cap\mathcal{B}=\emptyset} < n^L_{key}$ then it means that $BF_{\mathcal{A}}$ and $BF_{\mathcal{B}}$ have less than n^L_{key} (and thus less than n_{key}) indexes i where $bf_{\mathcal{A}}[i] = 1$ and $bf_{\mathcal{B}}[i] = 1$. It is then not possible that $\mathcal{A}$ and $\mathcal{B}$ have common elements. Regarding the false negative scenario, it could happen if we get too many *resulting overlapping bits* in the resulting Bloom filter $BF_{\mathcal{A}\cap\mathcal{B}=\emptyset}$.

Definition 2 (Resulting Overlapping Bit)**.** *When there exists a specific index i, where $bf_{\mathcal{A}}[i] = bf_{\mathcal{B}}[i] = 1$ and these two bits are coming from different elements.*

Therefore, a false negative consists of a case where $\mathcal{A}$ and $\mathcal{B}$ have no element in common but *Server* gets $X_{\mathcal{A}\cap\mathcal{B}=\emptyset} \geqslant n^L_{key}$, i.e. more than n^L_{key} *resulting overlapping bits* happened. To avoid such a case, we have to accurately tune the parameters such that in a case of distinct sets $\mathcal{A}$ and $\mathcal{B}$, the respective value $X_{\mathcal{A}\cap\mathcal{B}=\emptyset}$ will never (with acceptable probability) be greater than n^L_{key}. To do so, Alice has to carefully select the parameters n_{key} and m such that $X_{\cap=\emptyset} < n^L_{key}$. Value $X_{\cap=\emptyset}$ represents the expected value of $X_{\mathcal{A}\cap\mathcal{B}=\emptyset}$ when performing the *disjointness* protocol on two distinct sets $\mathcal{A}$ and $\mathcal{B}$. To express value $X_{\cap=\emptyset}$, we first give the probability of having a bit set to 1 for any index j in both Bloom filters $BF_{\mathcal{A}}$ and $BF_{\mathcal{B}}$, knowing that $\mathcal{A}$ and $\mathcal{B}$ are distinct:

$$\begin{aligned} & p(bf_{\mathcal{A}}[j] = 1 \wedge bf_{\mathcal{B}}[j] = 1) \\ &= p(bf_{\mathcal{A}}[j] = 1) \cdot p(bf_{\mathcal{B}}[j] = 1) \\ &= (1 - (1 - \frac{1}{m})^{n_{key} \cdot n_{\mathcal{A}}}) \cdot (1 - (1 - \frac{1}{m})^{n_{key} \cdot n_{\mathcal{B}}}) \end{aligned} \tag{15}$$

Finally, the expected amount of bits set to 1 in both $BF_{\mathcal{A}}$ and $BF_{\mathcal{B}}$ at the same index resulting from distinct set elements is:

$$X_{\cap=\emptyset} = m \cdot (1 - (1 - \tfrac{1}{m})^{X_{add}(n_{\mathcal{A}})}) \cdot (1 - (1 - \tfrac{1}{m})^{X_{add}(n_{\mathcal{B}})}) \tag{16}$$

When we have $Z'_{\mathcal{A},\mathcal{B}}$ common elements inserted in both Bloom filters, we get $X_{\mathcal{A}\cap\mathcal{B}=\emptyset} \simeq Z'_{\mathcal{A},\mathcal{B}} \cdot n_{key} + X_{\cap=\emptyset}$. Therefore, if Alice takes care that $X_{\cap=\emptyset}$ never gets greater or equal to n^L_{key}, then *Server* could notice when the two sets have common elements even in the case of $Z'_{\mathcal{A},\mathcal{B}} = 1$.

4.5.3 Choosing Parameters Regarding Correctness

In the classical use of Bloom filters as presented in (Bloom, 1970), some usage recommendations are made to generate parameters n_{key} and m:

$$m = -\frac{n_{\mathcal{A}} \cdot ln\ (\mathcal{P}_{FP})}{(ln\ 2)^2} \tag{17}$$

$$n_{key} = \frac{m}{n_{\mathcal{A}}} \cdot ln\ 2 \tag{18}$$

We recall that initially Bloom filters are not supposed to hold such relations testing as *inclusiveness* or *disjointness*. Therefore, the considerations on the generation of n_{key} and m are manifold.

5 PRIVACY ANALYSIS

In this section we show how our solutions fulfill privacy in terms of content and cardinality.

5.1 Distribution of the Overlapping Bits

In this section we analyze the characteristics of *overlapping bits* occurring throughout the basic step of Bloom filters generation. We obtain such a distribution by running the generation of 10^3 Bloom filters for each parameters configuration. From these distributions we could notice several characteristics. First, the more elements we add to the Bloom filter, the larger is the *overlapping bits* range. For instance, if we follow recommendations from (17) and (18), and we insert only 10 elements, we get a range of overlapping bits to approximately 10. When we have 100 inserted elements the range increases to approximately 40. Since our protocols use an HMAC function which generates a uniform random distribution, we could consider that the overlapping bits follow a normal distribution. Setting the parameters in the objective to tune the distribution to get an acceptable overlapping bits range regarding the aiming level of privacy could be intended.

As a second characteristic, we observe that when we have two sets with highly distant cardinalities $n_{\mathcal{A}} \ll n_{\mathcal{B}}$ (or resp. $n_{\mathcal{B}} \ll n_{\mathcal{A}}$), the number of overlapping bits in the Bloom filter of the smaller set $Y_{BF_{\mathcal{A}}}$ (resp. $Y_{BF_{\mathcal{B}}}$) substantially decreases and the one of the larger set substantially increases. Having too few overlapping bits in a Bloom filter could be problematic, especially if it could even be predictable by the attacker. By running tests we notice that no matter which n_{key} is picked or how many elements are inserted in the Bloom filters, if the ratio $\frac{n_{\mathcal{A}}}{n_{\mathcal{B}}}$ remains the same, then the expected amounts of overlapping bits in $BF_{\mathcal{A}}$ and $BF_{\mathcal{B}}$ remain approximately the same. Moreover, we see that it is even worse if we keep decreasing the ratio $\frac{n_{\mathcal{A}}}{n_{\mathcal{B}}}$. One solution to keep having an acceptable range of overlapping bits in the Bloom filter representations of the smaller set, even if we have a significant difference in the cardinalities, could be to use a greater domain $[n^L_{key}; n^U_{key}]$. Indeed, for the same ratio $\frac{n_{\mathcal{A}}}{n_{\mathcal{B}}}$, we get greater *overlapping bits* ranges. In (Tajan et al., 2019), some results obtained by testing the overlapping bits distribution are presented.

5.2 Privacy on the Content

First, we claim that no attacker could determine which concrete elements from $\mathcal{A}$ is included in $\mathcal{B}$. This holds by means of the Bloom filter construction. Indeed, each element from the sets are mapped with the HMAC function constructed from a cryptographic hash function and therefore benefits from its on-wayness characteristic. The only straightforward manner to get any knowledge on the Bloom filter content would be to use the *test()* function which is only computable by Alice and Bob. More concretely, *Server* does not know the HMAC's keys and cannot generate its own Bloom filter or add any element to an existing one and perform the set relations. Indeed, they require that all the considered Bloom filters are generated with the same keys. Also, *Server* is not able to learn from $BF_{\mathcal{A}}$ or $BF_{\mathcal{B}}$ if a specific element from $\mathcal{A}$ is also included in $\mathcal{B}$. All elements inserted in a Bloom filter are mixed together and it is not possible, even from the same Bloom filter, to distinguish them.

5.3 Privacy on the Cardinality

In this section, we focus on the ability of any attacker to retrieve the cardinality of the sets from one or multiple versions of the Bloom filter's representation. The overlapping bits property of the Bloom filters allows to hide the exact number of elements in sets $\mathcal{A}$ and $\mathcal{B}$. However, *Server* is able to determine the amount of bits set to 1 in the Bloom filters. It could then deduct the following information: $n_{\mathcal{A}} \geqslant \frac{X_{BF_{\mathcal{A}}}}{n_{key}}$. By keeping parameter n_{key} secret to *Server*, we consider the cardinalities obfuscated to a certain level.

We recall that there exists an optimized manner to get the cardinality of a set from its Bloom filter representation as explained in Section 4.1, the S&B technique. Without any overlapping bit, getting the result is therefore straightforward. On the contrary, having multiple overlapping bits will lead any non-authorized party to misinterpret the cardinality. To ensure that, the ratio of the amount of overlapping bits over parameter n_{key} should be important.

We also notice that having an acceptable probability of false negative and an acceptable level of privacy for the set cardinalities are contradicting strategies. Indeed, our approach to solve the *disjointness* set relation is based on reducing the amount of overlapping bits to avoid confusion having common elements.

5.4 Sets Cardinality Attack

We present here how an attacker could aim to retrieve cardinalities $n_{\mathcal{A}}$ and $n_{\mathcal{B}}$. To do so, *Server* will firstly try to determine parameter n_{key} used by Alice and Bob. *Server* knows that $n_{key} \in [n^L_{key}; n^U_{key}]$ and that n_{key} is a factor of the amount of bits inserted in both Bloom filters. The candidates list for n_{key} is represented as $L_{n_{key}} = \{l_1, \ldots, l_{\lambda_{n_{key}}}\}$ with $\lambda_{n_{key}}$ a security parameter that represents the size of this list. We also con-

Table 1: Execution of the two set relations with n_{key} selected in $[5.10^2; 2.10^3]$.

	Parameters					Execution times in sec. for		Attack
	$n_{\mathcal{W}}$	$n_{\mathcal{L}_1}$	$n_{\mathcal{L}_2}$	m	n_{key}	$INC(\mathcal{L}_1, \mathcal{W})$	$DIS(\mathcal{L}_2, \mathcal{W})$	$\lambda_{n_{key}}$
Use Case 1	10^3	10^3	10^3	$1.18 \cdot 10^9$	733	$2.57 \cdot 10^{-1}$	$2.16 \cdot 10^{-1}$	406
	10^3	10^3	10^3	$7.62 \cdot 10^9$	1861	$2.51 \cdot 10^{-1}$	$7.44 \cdot 10^{-1}$	397
	10^4	$9 \cdot 10^3$	$2 \cdot 10^2$	$9.47 \cdot 10^9$	1468	$2.16 \cdot 10^{-1}$	$8.67 \cdot 10^{-1}$	29
Use Case 2	10^2	10^4	/	$4.40 \cdot 10^9$	1416	/	$7.41 \cdot 10^{-1}$	32
	10^2	$5 \cdot 10^4$	/	$8.13 \cdot 10^9$	861	/	$3.36 \cdot 10^1$	37
	10^2	$5 \cdot 10^5$	/	$1.27 \cdot 10^{10}$	561	/	$1.9 \cdot 10^2$	32

sider the two sub-lists $L^{\mathcal{A}}_{n_{key}} = \{l_1, \ldots, l_{\lambda_{\mathcal{A}}}\}$ and $L^{\mathcal{B}}_{n_{key}} = \{l_1, \ldots, l_{\lambda_{\mathcal{B}}}\}$ which correspond to the lists of factors regarding $BF_{\mathcal{A}}$ and $BF_{\mathcal{B}}$ before the cross-checking that leads to $L_{n_{key}}$. We set $Y_{BF_{\mathcal{A}}} \in [ob_{\mathcal{A}_1}; ob_{\mathcal{A}_2}]$ and $Y_{BF_{\mathcal{B}}} \in [ob_{\mathcal{B}_1}; ob_{\mathcal{B}_2}]$ the amounts of overlapping bits in the Bloom filters. In each Bloom filter, some overlapping bits could have occurred, therefore the attacker knows that regarding $BF_{\mathcal{A}}$, n_{key} could be a factor of $X_{BF_{\mathcal{A}}}$ or $(X_{BF_{\mathcal{A}}} + 1)$ or $(X_{BF_{\mathcal{A}}} + 2) \ldots$ Similarly holds for $BF_{\mathcal{B}}$. It means that $L^{\mathcal{A}}_{n_{key}}$ (resp. $L^{\mathcal{B}}_{n_{key}}$) is composed by elements l_j which verify the two characteristics:

$$l_j \in [n^L_{key}; n^U_{key}] \text{ and } l_j | x_{\mathcal{A}} \quad (19)$$

$$\text{with } x_{\mathcal{A}} \in [X_{BF_{\mathcal{A}}} + ob_{\mathcal{A}_1}; X_{BF_{\mathcal{A}}} + ob_{\mathcal{A}_2}]$$

$$(\text{resp. } l_j | x_{\mathcal{B}} \text{ with } x_{\mathcal{B}} \in [X_{BF_{\mathcal{B}}} + ob_{\mathcal{B}_1}; X_{BF_{\mathcal{B}}} + ob_{\mathcal{B}_2}])$$

Finally, we have $L_{n_{\mathcal{A}}} = (l_i)_{i \in [1; \lambda_{n_{\mathcal{A}}}]}$ the list of candidates for $n_{\mathcal{A}}$ with $\lambda_{n_{\mathcal{A}}}$ the amount of elements in $L_{n_{\mathcal{A}}}$. The first step of the attack consists of listing all the common factor of $\{X_{BF_{\mathcal{A}}}, (X_{BF_{\mathcal{A}}} + 1), (X_{BF_{\mathcal{A}}} + 2), \ldots\}$ and $\{X_{BF_{\mathcal{B}}}, (X_{BF_{\mathcal{B}}} + 1), (X_{BF_{\mathcal{B}}} + 2), \ldots\}$ to generate lists $L^{\mathcal{A}}_{n_{key}}$ and $L^{\mathcal{B}}_{n_{key}}$. Then, *Server* will intersect the two list to generate the candidates list $L_{n_{key}}$.

The second phase of the attack is to translate $L_{n_{key}}$ into lists $L_{n_{\mathcal{A}}}$ and $L_{n_{\mathcal{B}}}$. *Server* could use the S&B technique (6) to approximate size $n_{\mathcal{A}}$ and since parameter m is public and value $X_{BF_{\mathcal{A}}}$ is directly computable, we have the following function:

$$n^*_{\mathcal{A}}(n_{key}) = -\frac{m}{n_{key}} ln\Big[1 - \frac{X_{BF_{\mathcal{A}}}}{m}\Big] \quad (20)$$

When we look closely to $L_{n_{key}}$, we could notice that if some elements are following each others, they are translated to the same $n_{\mathcal{A}}$'s candidate. In other words, multiple elements from $L_{n_{key}}$ correspond to the same element from $L_{n_{\mathcal{A}}}$, thus $\lambda_{n_{\mathcal{A}}} \leqslant \lambda_{n_{key}}$.

6 RESULTS

The Bloom filter-based set relations have various applications in practice. We selected two of them and applied our solutions. We implemented our protocols in Java and the measurements have been made with a CPU configuration of Intel Core i5 2.40GHz x4.

6.1 Results on a Cloud Auditing Use Case

We test our solution with parameters suiting a cloud security auditing use case from (Tajan et al., 2016). A third party auditor should verify that a cloud service provider ($\mathcal{CSP}$) performed correctly an access control on data from a client stored online. The auditor thus performs the sets relations on logfiles $\mathcal{L}_1$ and $\mathcal{L}_2$ from $\mathcal{CSP}$ composed of 10^2 to 10^4 IP addresses and a whitelist $\mathcal{W}$ from the client composed of 10^3 to 10^4 IP addresses. The tested parameters configurations are presented in Table 1 where the two set relations $INC(\mathcal{L}_1, \mathcal{W})$ and $DIS(\mathcal{W}, \mathcal{L}_2)$ are tested 10^4 times. In both cases we obtain 0.00% of false positives and false negatives. We remark that the set relations are equivalent to bit-wise operations on the Bloom filters. We also notice that the performance times considering the set cardinality privacy are by far acceptable especially in an auditing use case. Finally, we express the privacy on the sets' cardinality by $\lambda_{n_{key}}$.

6.2 Results on a Mobile Devices Tracking Use Case

In this use case from (Tajan and Westhoff, 2019), there are three different sub-use cases where a third party should verify if any suspect user from a government agency's whitelist has been connected to a specific wireless access point with one of its devices. In Table 1 we give examples of relevant parameters that produced successful computations along with the running time of the disjointness function in seconds. We notice a significant decrease of the sets' cardinality privacy when the sets have different sizes as explained in Section 5.1. To overcome this privacy weakness, the parties could agree on a default size and adding dummy elements if necessary.

7 CONCLUSION

We showed how to compute two specific set relations namely *private outsourced inclusiveness test* and *private outsourced disjointness test* using the space-efficient data representation Bloom filter. In addition to fulfill privacy on the content, we provided a certain level of privacy on the cardinality of the Bloom filter's data structure. Our implementation's results validate an acceptable level of privacy, for instance when applied to a cloud security audit on access control. Such an approach based on Bloom filters could be easily adapted also to other set relations or operations like *equality* or *relative complement*.

REFERENCES

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

Burkhart, M. and fontas, X. D. (2012). Fast private set operations with sepia.

Churches, T. and Christen, P. (2004). Some methods for blindfolded record linkage. *BMC Med. Inf. & Decision Making*.

Dong, C., Chen, L., and Wen, Z. (2013). When private set intersection meets big data: an efficient and scalable protocol. In Sadeghi, A., Gligor, V. D., and Yung, M., editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 789–800. ACM.

Egert, R., Fischlin, M., Gens, D., Jacob, S., Senker, M., and Tillmanns, J. (2015). Privately computing set-union and set-intersection cardinality via bloom filters. In Foo, E. and Stebila, D., editors, *Information Security and Privacy - 20th Australasian Conference, ACISP 2015, Brisbane, QLD, Australia, June 29 - July 1, 2015, Proceedings*, volume 9144 of *Lecture Notes in Computer Science*. Springer.

Freedman, M. J., Nissim, K., and Pinkas, B. (2004). Efficient private matching and set intersection. In Cachin, C. and Camenisch, J., editors, *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, volume 3027 of *Lecture Notes in Computer Science*. Springer.

Goh, E. (2003). Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216.

Goldreich, O. (2004). *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press.

Hohenberger, S. and Weis, S. A. (2006). Honest-verifier private disjointness testing without random oracles. In Danezis, G. and Golle, P., editors, *Privacy Enhancing Technologies, 6th International Workshop, PET 2006, Cambridge, UK, June 28-30, 2006, Revised Selected Papers*, volume 4258 of *Lecture Notes in Computer Science*, pages 277–294. Springer.

Kerschbaum, F. (2012). Outsourced private set intersection using homomorphic encryption. In Youm, H. Y. and Won, Y., editors, *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012*. ACM.

Kiayias, A. and Mitrofanova, A. (2005). Testing disjointness of private datasets. In Patrick, A. S. and Yung, M., editors, *Financial Cryptography and Data Security, 9th International Conference, FC 2005, Roseau, The Commonwealth of Dominica, February 28 - March 3, 2005, Revised Papers*, Lecture Notes in Computer Science, pages 109–124. Springer.

Kissner, L. and Song, D. X. (2005). Privacy-preserving set operations. In Shoup, V., editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*. Springer.

Li, Z. and Gong, G. (2012). Efficient data aggregation with secure bloom filter in wireless sensor networks.

Pinkas, B., Schneider, T., and Zohner, M. (2014). Faster private set intersection based on OT extension. In Fu, K. and Jung, J., editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 797–812. USENIX Association.

Qiu, L., Li, Y., and Wu, X. (2007). Preserving privacy in association rule mining with bloom filters. *J. Intell. Inf. Syst.*, 29(3):253–278.

Swamidass, S. J. and Baldi, P. (2007). Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of Chemical Information and Modeling*, 47(3):952–964. PMID: 17444629.

Tajan, L. and Westhoff, D. (2019). Retrospective tracking of suspects in GDPR conform mobile access networks datasets. In *Proceedings of the Third Central European Cybersecurity Conference, CECC 2019, Munich, Germany, November 14-15, 2019*, pages 16:1–16:6. ACM.

Tajan, L., Westhoff, D., and Armknecht, F. (2019). Private set relations with bloom filters for outsourced SLA validation. *IACR Cryptology ePrint Archive*, 2019:993.

Tajan, L., Westhoff, D., Reuter, C. A., and Armknecht, F. (2016). Private information retrieval and searchable encryption for privacy-preserving multi-client cloud auditing. In *11th International Conference for Internet Technology and Secured Transactions, ICITST 2016, Barcelona, Spain, December 5-7, 2016*. IEEE.

Ye, Q., Wang, H., Pieprzyk, J., and Zhang, X. (2008). Efficient disjointness tests for private datasets. In Mu, Y., Susilo, W., and Seberry, J., editors, *Information Security and Privacy, 13th Australasian Conference, ACISP 2008, Wollongong, Australia, July 7-9, 2008, Proceedings*, volume 5107 of *Lecture Notes in Computer Science*, pages 155–169. Springer.