Trust-Preserving Set Operations

Ruggero Morselli  Bobby Bhattacharjee  Jonathan Katz  Pete Keleher
Department of Computer Science, University of Maryland, College Park, Maryland, USA
{ruggero, bobby, jkatz, keleher}@cs.umd.edu

Abstract—We describe a method for performing trust-preserving set operations by untrusted parties. Our motivation for this is the problem of securely reusing content-based search results in peer-to-peer networks. We model search results and indexes as data sets. Such sets have value for answering a new query only if they are trusted. In the absence of any system-wide security mechanism, a data set is trusted by a node \(a\) only if it was generated by some node which is trusted by \(a\).

Our main contributions are a formal definition of the problem as well as an efficient scheme that solves this problem by allowing untrusted peers to perform set operations on trusted data sets while also producing unforgeable proofs of correctness. This is accomplished by requiring trusted nodes to sign appropriately-defined digests of generated sets; each such digest consists of an RSA accumulator and a Bloom filter. The scheme is general, and has other applications as well. We give an analysis demonstrating the low overhead of the scheme, and we include experimental data which confirm the analysis.

I. INTRODUCTION

This paper describes a method for performing trust-preserving computations on sets. The party performing the computation does not need to be trusted, but the result is a set which is trusted to the same extent as the original input. These properties allow the scheme to be used in distributed environments with many untrusted hosts. Our motivation for addressing this issue is the problem of securely reusing content-based search results in peer-to-peer (P2P) networks. However, the techniques have a range of other potential applications.

For purposes of this work, a P2P network is a large, distributed, and decentralized structure that allows \(n\) autonomous peers to cooperatively provide access to a potentially large set of data. A number of approaches to providing lookup services on such data have been proposed. Distributed hash tables (DHTs) (e.g., Chord [1], CAN [2], Tapestry [3], and Pastry [4]) use cryptographic hashes to provide near-random association of objects to sites that “publish” the object to the rest of the system. Objects are “looked up” by using the hash of the object name to route to the corresponding peer, usually in \(O(\log n)\) hops. DHTs provide excellent balance of routing load because paths to a given node vary widely based on the path’s origin, and because related objects (even with only slightly different names) are randomly distributed through the system.

This random distribution is the strength of the approach, but it also destroys object locality. A set of objects chosen by common attributes or characteristics must necessarily include members mapped to peers throughout the system. This distribution, together with the sheer scale and diversity of the source data, make it impractical to consider approaches to providing content searches that (even periodically) flood the network in order to satisfy queries. Similarly, flooding the network should not be considered a practical primitive when creating indexes for satisfying queries.

Instead, searches must rely on attribute indexes that are incrementally created and maintained. Each attribute index records nodes having a particular attribute, or attribute value. Given the nature of P2P applications, the indexes should be as distributed and as decentralized as the underlying system. However, a straightforward use of distributed indexes is still potentially quite expensive. A simple conjunction of two indexes will usually require the contents of at least one of the indexes to be sent across the network. These indexes may be large, as their size potentially grows linearly with the number of system peers.

The costs of satisfying queries can be reduced by caching query results in the system. Locality in query streams and object attributes can then be exploited by using prior results to help satisfy subsequent queries. For example, conjunctions of two or more attributes can be stored. Such results are typically much smaller than the original attribute indexes, and can reduce the cost of satisfying multi-item queries by several orders of magnitude. Using such views efficiently is non-trivial, but has been addressed in other work [5].

Previous work assumes that all system peers cooperate in constructing and updating these indexes. In contrast, we would like to avoid this assumption; thus, our motivation is the problem of making such cached results secure. Our main contribution is to show an efficient scheme allowing an untrusted peer to perform set operations on trusted data sets, and to additionally supply an unforgeable proof of correctness of the result. The approach we take is to reduce the trustworthiness of the derived value (i.e., the cached result) to a function of the trustworthiness of the sources (i.e., the original attribute values). The latter are more easily secured, as they are long-lived and fairly heavyweight. Cached results, however, are light-weight, ephemeral, and hence may be stored at arbitrary (possibly untrusted) nodes.

As mentioned above, we model indexes or cached query as sets. The trusted operations we wish to perform are thus standard set operations such as union, difference, and intersection.

Our work may be viewed as extending the concept of
authenticated data structures [6], [7], [8], [9], [10]. Briefly, these allow an untrusted host to answer queries about a single trusted data set in a trustworthy way (e.g., given a set \( S \) generated by a trusted source, they allow an untrusted directory \( d \) to answer queries of the form “is \( x \) in \( S \)?”). Our work focuses on the more challenging (and more broadly-applicable) case of answering queries about multiple data sets in an efficient way (e.g., the intersection problem can be cast as a query “give me every \( x \) in \( S_1 \cap S_2 \)”).

We briefly introduce an example of the problem we are trying to solve, along with some terminology. Let \( s_1 \) and \( s_2 \) be two trusted source nodes. We assume \( s_1 \) (respectively, \( s_2 \)) stores a “set” \( S_1 \) (respectively, \( S_2 \)); cf. Fig. 1. An untrusted directory \( d \) caches the result of some set operation on \( S_1 \) and \( S_2 \); in this example, we assume \( d \) stores their intersection. Here, we also assume that the computation of \( S_1 \cap S_2 \) is performed by \( d \), although this need not necessarily be the case.

The problem we analyze is how to construct a scheme that allows an arbitrary client \( c \) to verify that \( d \) did not falsify the result of the query. If such a verification can be performed, \( c \) can obtain the cached result from \( d \) and use it during evaluation of a new query that includes the same expression.

We assume that \( s_1 \) and \( s_2 \) each possess a private/public key pair for a signature scheme, and that \( c \) knows the public keys.

The idea we exploit is the following: when \( s_1 \) sends a copy of \( S_1 \) to \( d \), it also appends a digest \( D_1 \) of \( S_1 \) together with a signature on this digest produced using \( s_1 \)'s signing key. Source \( s_2 \) supplies similar information. In the figure, \( s_1 \) sends to \( d \) the digest \( D_1 \) along with a signature \( \sigma_1 \) on this digest.

When \( d \) is queried for the intersection of the two sets, it sends the intersection \( I \), the two digests, and other information derived during the intersection evaluation. Together, this information should be sufficient to prove the result accurate (i.e., that \( I = S_1 \cap S_2 \)).

Note that a malicious directory \( d \) could insert extraneous elements in the reply but still include all the elements of the intersection; we will call this an insertion attack. Alternately, it could fail to include elements that are in the intersection without inserting any bogus elements; we will call this a deletion attack. More generally, it could both insert and delete elements.

As a trivial solution, we could set \( D_1 = S_1 \) and \( D_2 = S_2 \), so the two sources are simply signing the respective sets. The directory could then provide both of the original sets in response to the query, together with the signatures. This is clearly secure, since the client can verify the authenticity of the two original sets and compute the intersection himself. Unfortunately, since the two original sets might be much bigger than their intersection, the communication overhead for the response may be excessive. We show here schemes which perform significantly better than this trivial solution.

The rest of the paper is organized as follows. In Section II we give a formal definition for an intersection scheme that solves the sample intersection problem described above. In Section III, we summarize two existing techniques that we will use in our construction. In Section IV, we introduce an initial scheme to solve the intersection problem and discuss its limitations. In Section V, we give a secure and efficient intersection scheme that avoids these limitations. In Section VI-C, we show how to make the scheme composable: that is, how to securely compute intersections of sets that are themselves intersections performed by other untrusted nodes. In Section VI we show how our scheme can be easily extended to include the set union and set difference operations. In Section VI-C we discuss implementation details, followed by experimental results demonstrating the efficiency of our scheme in Section VII.

II. DEFINING AN INTERSECTION SCHEME

We first provide a formal definition of a secure intersection scheme. It is straightforward to extend this definition for other set operations (or to support multiple set operations simultaneously).

**Definition 1 (Intersection scheme):** An intersection scheme is a triple of efficient algorithms \((\text{Dgst, CV, Vrfy})\) (the digest, check value and verify algorithms, respectively) with the following properties:

- **Dgst** takes as input a set \( S \) of elements in some universe \( U \) and outputs a bit string \( D \leftarrow \text{Dgst}(S) \) that we will call a digest of \( S \).
- **CV** takes as input two sets \( S_1, S_2 \) and two digests \( D_1, D_2 \) and outputs a bit string \( C \leftarrow \text{CV}(S_1, S_2, D_1, D_2) \), that we will call a check value for that quadruple of inputs.
- **Vrfy** takes as input one set \( I \), two digests \( D_1 \) and \( D_2 \) and a check value \( C \); it produces a boolean value...
b = \text{Vrfy}(D_1, D_2, I, C).\footnote{When an algorithm A is randomized, we denote its output y on an input x by y \leftarrow A(x). When A is deterministic we denote the same process by y = A(x).}

- (Correctness.) For all \( S_1, S_2, I \) if:

\[
D_1 \leftarrow \text{Dgst}(S_1) \land D_2 \leftarrow \text{Dgst}(S_2) \\
\land C \leftarrow \text{CV}(S_1, S_2, D_1, D_2) \land I = S_1 \cap S_2
\]

then \text{Vrfy}(D_1, D_2, I, C) = 1.

- (Security.) It is computationally infeasible for an adversary, on input \( S_1, S_2, D_1, D_2 \), to find a set \( I' \neq I \) and a value \( C' \) such that \text{Vrfy}(D_1, D_2, I', C') = 1.

An intersection scheme is used as follows (adapting the terminology from the Introduction). Source \( s_1 \) computes \( D_1 = \text{Dgst}(S_1) \) and a signature \( \sigma_1 \) on \( D_1 \) (computed using its signing key); source \( s_2 \) computes \( D_2 = \text{Dgst}(S_2) \) and \( \sigma_2 \) analogously. Directory \( D \) has \( D_1 \) and \( D_2 \), the signatures \( \sigma_1 \) and \( \sigma_2 \), as well as \( S_1 \) and \( S_2 \) (cf. Fig. 1). When client \( c \) queries \( d \), an honest \( d \) computes \( I = S_1 \cap S_2 \) and \( C = \text{CV}(S_1, S_2, D_1, D_2) \); finally \( d \) sends the values \( (I, D_1, \sigma_1, D_2, \sigma_2, C) \).

Node \( c \), in response to its query, receives \( (I', D_1, \sigma_1, D_2, \sigma_2, C') \). Note that if \( d \) is malicious, then \( I' \) and \( C' \) might be different from \( I \) and \( C \). (The signature scheme prevents \( d \) from changing \( D_1, D_2 \).) The client first checks that the signatures are valid and then it runs \text{Vrfy}(D_1, D_2, I', C'). If \( d \) acted correctly, this will evaluate to 1 (by the correctness property); on the other hand, if \( I \neq I' \) then this should evaluate to 0 (by the security property).

III. BACKGROUND

We briefly summarize two existing techniques that we use to construct our secure intersection scheme. The first is the RSA accumulator, which allows an untrusted directory to securely prove membership of elements in a set. The second is the (counting) Bloom filter.

A. The RSA accumulator

Previous work [11], [6], [8] has shown how to use cryptographic accumulators to solve the following, related problem: A source entity \( s \) generates some set \( S \). A copy of the set \( S \) is stored in the untrusted directory entity \( d \). A client entity \( c \) queries the directory asking whether an element \( x_i \) belongs to the set \( S \) or not. If the directory replies with an affirmative answer, it must be able to prove that \( x_i \) is actually in the set. (Note that in contrast to authenticated data structures, accumulators themselves do not allow proofs of non-membership.)

To allow for this, \( s \) computes a cryptographic accumulator \( \text{Acc}(S) \) on the set \( S \), signs the result, and sends \( S \) and a copy of the signature to \( d \). When \( d \) wants to prove that \( x_i \in S \), it computes a value \( w_i \) (called the witness of \( x_i \)) from the inputs \( S \) and \( x_i \); it then sends \( \text{Acc}(S), w_i, \) and the signature on \( \text{Acc}(S) \) to the client. The client, after verifying the signature, can verify from \( \text{Acc}(S), w_i, x_i \) that the answer is correct.

The RSA accumulator has the important property that the size of \( \text{Acc}(S) \) does not depend on the size of \( S \). We now describe it in more detail. To begin, the source chooses a random RSA modulus \( N \) and a random \( a \in \mathbb{Z}_N^* \) and publishes these values.\footnote{\( \mathbb{Z}_N^* \) is the set of integers between 1 and \( N - 1 \), that are relatively prime with \( N \).} We also assume an efficient algorithm \( R \) that, on input a \( u \)-bit string \( x \), produces an odd integer \( e = R(x) \), called the representative of \( x \). We require that the \( R \) implements a division intractable function [12]. That is to say, it is infeasible for an adversary to find elements \( x_1, \ldots, x_n, x' \) such that \( R(x') \) divides the product \( R(x_1) \cdots R(x_n) \). (Constructions of such \( R \), under common cryptographic assumptions, are known.)

Let \( S = \{x_1, \ldots, x_n\} \) be a set, where each \( x_i \) is represented by \( u \) bits. In order to compute the accumulator of \( S \), the source first computes representatives \( e_1, \ldots, e_n \) of \( x_1, \ldots, x_n \), and then outputs:

\[
\text{Acc}(S) = a^{e_1 \cdots e_n} \mod N.
\] (1)

The witness \( w_i \) of \( x_i \in S \) is computed via:

\[
w_i = a^{e_1 \cdots e_{i-1} e_{i+1} \cdots e_n} \mod N.
\]

In order to verify the correctness of an answer \( \text{Acc}(S), w_i, x_i \), the client verifies that:

\[
w_i^{e_i} = \text{Acc}(S) \mod N.
\]

Note that the scheme can easily be extended to prove \( X \subseteq S \) for a set \( X \) of any size.

It can be proved ([6], [12]) that, under the strong RSA assumption, it is infeasible for an adversary to “fool” a client into believing that some \( x \) is in \( S \) when in fact it is not; i.e., it is infeasible for an adversary to find \( (x, w) \) such that \( x \notin S \) but \( w^{R(x)} = \text{Acc}(S) \mod N \).

B. Counting Bloom filters

We assume that the reader has some basic familiarity with Bloom filters [13]. Our protocol employs a generalization of Bloom filters, called counting Bloom filters [14]. A counting Bloom filter is a data structure characterized by two parameters \( k \) and \( m \). It uses \( k \) independent hash functions, denoted \( h_1, \ldots, h_k \), whose range is \( \{1, \ldots, m\} \). Given a set \( S \), the counting Bloom filter of \( S \) (denoted by \( \text{Bl}(S) \)) is a vector of \( m \) non-negative integers (counters); the \( j \)-th element of this vector (denoted \( \text{Bl}(S)_j \)) is the number of pairs \( (i, x) \) (with \( x \in S \) and \( 1 \leq i \leq k \)) such that \( h_i(x) = j \).

IV. A FIRST ATTEMPT

Suppose we use a Bloom filter as a digest for the intersection scheme. Also suppose that the directory answers the intersection query with a “claimed” intersection \( I' \) and the two Bloom filters \( B_1, B_2 \) for the two original sets \( S_1, S_2 \). The client can verify that each element of \( I' \) appears to be in the two original sets, using the Bloom filters. Note first that this does not prevent a malicious directory from returning a subset
of the real intersection. Also, the Bloom filter would need to be prohibitively large in order to ensure that an adversary would not be able to falsely insert an element.

We generalize the above approach using counting Bloom filters. As we will discuss, this scheme is neither secure nor efficient, but it will be useful in understanding the main idea of the correct secure intersection scheme given in the next section.

Consider the two sets $S_1$, $S_2$ in Fig. 2, together with the corresponding counting Bloom filters (with $k = 2$ hash functions and $m = 7$ counters). The label on the side of each Bloom filter counter is the list of items in the set that hash to the index of that counter; e.g., in $Bl(S_1)$, the element “dog” appears in the label of the fourth and sixth counters, meaning that, for example, $h_1(“dog“) = 4$ and $h_2(“dog“) = 6$.

The client receives a signed copy of these two Bloom filters, together with the directory’s signed copy of the intersection $I$. To verify correctness, the client computes the Bloom filter $\hat{Bl}$, obtained as the element-by-element minimum of $Bl(S_1)$ and $Bl(S_2)$, and the Bloom filter $Bl(I')$ of the returned intersection (Fig. 3). Note that the condition:

$$Bl(I)_j \leq \hat{Bl}_j \quad \forall j = 1, \ldots, m. \quad (2)$$

holds for the correct intersection, because each element in $I$ also belongs to both $S_1$ and $S_2$.

In the given example (but not in general), any insertion attack (that is to say, any attack that would make $I'$ a superset of $I$) would be detected, because it would make at least one of the counters of the Bloom filter of $I'$ (Fig. 3 on the right) greater than the corresponding counter of $\hat{Bl}$ (same figure, on the left).

However, this test does not prevent an attacker from performing a deletion attack (i.e., returning an $I'$ that is a subset of $I$). For example, the directory could claim that the intersection is empty because the only effect of removing elements is to decrease some of the counters. Ideally, if $Bl$ and $Bl(I')$ were equal, then we would know for certain that a deletion attack was not performed. The problem is that, even for the legitimate intersection $I$, some counters of the Bloom filter could be strictly less than the corresponding counter in $\hat{Bl}$. In our example, the counter with index 6 of $Bl(I')$ is 0 while the corresponding in $Bl$ is 1; this is due to the fact that there is at least one element (“dog”) which is only in the first set and at least one element (“whale” and “monkey”) which is only in the second set, all of which hash to the same index 6. We will say that such an index is a gap, meaning that there is a gap between the two counters in question.

**Definition 2 (Gap):** An index $j$ is called a gap if $Bl(I)_j$ is strictly less than $\hat{Bl}_j$.

A deletion attack would create more gaps, as perceived by the client. Therefore we require the directory to justify each gap. In our example, if the directory returns the correct intersection, it can justify the gap 6 by including in the answer all the elements in both sets that map to index 6 (“dog” for the first set, “whale” and “monkey” for the second set); we will say that those are the check elements of the response. If the directory tries to return an empty intersection, it must justify three gaps (2, 4 and 6), for example by finding two strings that hash to indexes 2 and 4 and adding them to the check elements. The attack is restricted but still feasible.

More formally, the untrusted directory $d$ will include in its answer to the intersection query a check value $C$ that consists of a pair of sets $(C_1, C_2)$, that are respectively a set of check elements in $S_1 \setminus I$ and $S_2 \setminus I$. Upon receiving the answer $(I', Bl(S_1), \sigma_1, Bl(S_2), \sigma_2, C_1', C_2')$, the client $c$ computes the Bloom filters $\hat{Bl}$ obtained as the element-by-element minimum of $Bl(S_1)$ and $Bl(S_2)$. The client $c$ then checks that condition (2) holds and also will check that, for each $j = 1, \ldots, m$ such that $Bl(I')_j < \hat{Bl}_j$:

$$Bl(I')_j + Bl(C'_1)_j = Bl(S_1)_j \quad \text{(3)}$$
$$Bl(I')_j + Bl(C'_2)_j = Bl(S_2)_j \quad \text{(4)}$$

and will reject if these are not satisfied. Condition (3) enforces that for each gap $j$ the check element set $C_1$ must contain elements that collectively hash $Bl(S_1) \setminus I'_j$ times to index $j$. That is to say:

$$Bl(S_1) \setminus I'_j = Bl(C_1)_j.$$

Analogously, Condition (4) states that $C_2$ must contain elements that hash $Bl(S_2) \setminus I'_j$ times to index $j$. 

---

Fig. 2. Counting Bloom filters of two sets.

Fig. 3. The elementwise minimum, the Bloom filter of the intersection and the gap for the example in Fig. 2.
We stress that the scheme as sketched above is not completely secure. See discussion below. However, it will serve as a useful building block. In the next subsection we state some results that estimate the number of check elements needed by the scheme; we use this result in Section IV-B to estimate the security and efficiency of the scheme. In Section V we will show how to extend this approach to construct a efficient and secure scheme.

A. Number of check elements

Definition 3 (Load of the filter): The load of a counting Bloom filter Bl(S) of a set S of n elements, with k hash functions and m counters, is the expected value l of each counter in the filter; i.e., l = \( \frac{kn}{m} \).

Consider the counting Bloom filter intersection scheme, where the two sets \( S_1 \) and \( S_2 \) have respectively \( n_1 \) and \( n_2 \) elements and their intersection has \( |I| = q \) elements. Let \( n \) be the maximum set size allowed by the scheme (\( n_1, n_2 \leq n \)). Let \( l_1 = \frac{kn_1}{m} \), \( l_2 = \frac{kn_2}{m} \) be the loads of Bl(\( S_1 \)), Bl(\( S_2 \)) and let \( l = \frac{kn}{m} \) be the maximum load of the scheme.

We assume that if we randomly generate \( n \) distinct objects \( x_1, \ldots, x_n \) from the universe according to any distribution of interest, the values \( \{h_i(x_a) : i = 1, \ldots, k; a = 1, \ldots, n\} \) are independent random variables uniformly distributed in \( \{1, \ldots, m\} \).

Theorem 1: The expected number of check elements for both sets satisfies:

\[
E[|C_1|] \leq m(1 - e^{-l_1})l_1 \leq ml_1 \leq ml^2 \quad (5)
\]

\[
E[|C_2|] \leq m(1 - e^{-l_2})l_2 \leq ml_2 \leq ml^2. \quad (6)
\]

A proof can be found in the appendix.

B. Security and efficiency considerations

The intersection scheme described above suffers from several kinds of attacks. For example, the attacker could insert an element in the intersection that is not in any of the original sets, as long as all the indices it maps to are gaps; it can be shown that making the probability of this attack negligible can be done only at the cost of Bloom filters with a prohibitively large number of counters (for a probability of \( 2^{-50} \) or less, we need \( m \geq 37n \)).

Independently of the security considerations, the load of the Bloom filters must be small enough to keep the number of check elements small. This implies that \( m \) must be much bigger than \( n \), making the size of the Bloom filters themselves bigger than the size of encoding the original sets. We address this problem next.

In order to improve the efficiency of the counting Bloom Filter scheme, we use compressed counting Bloom filters [15] in which the size of the counting Bloom filter is reduced by applying a compression algorithm. This is still not sufficient to make the intersection scheme secure and efficient, but it is the first step towards the solution to the problem. In this section we introduce some notation and some results on the compressed counting Bloom filters that we will need in Section V.

Let \( S \) be a set of \( n \) elements and consider its counting Bloom Filter Bl(\( S \)) with \( m \) counters and \( k \) hash functions. We can apply a data compressor to the filter, obtaining a compressed filter of some size \( z \).

Claim 1: An upper bound to the size of the compressed counting Bloom filter, assuming optimal compression, is given by

\[
z = mH(l) \quad (7)
\]

where \( H(l) \) is the entropy of a Poisson distribution with mean \( l \).

A proof appears in [16].

V. A CRYPTOGRAPHICALLY-SECURE INTERSECTION SCHEME

In this section we describe our full solution to the intersection problem. In Section V-B, we analyze the scheme to determine the optimal values of the parameters and the resulting overhead.

The scheme works as follows:

- To produce the digest of a set \( S_i \), generate:
  - the compressed counting Bloom filter Bl(\( S_i \));
  - the RSA accumulator Acc(\( S_i \));
  - a signature \( \sigma_i \) on both.
- When a directory receives a query, it returns:
  - the intersection \( I \);
  - the Bloom filters Bl(\( S_1 \)), Bl(\( S_2 \));
  - the RSA accumulators Acc(\( S_1 \)), Acc(\( S_2 \));
  - the check element sets \( C_1, C_2 \) (computed exactly as described in Section IV);
  - two RSA accumulator witnesses \( w_1, w_2 \). The two witnesses prove that \( (I \cup C_1) \subset S_1 \) and \( (I \cup C_2) \subset S_2 \) respectively.
  - the two signatures \( \sigma_1, \sigma_2 \).
- The client verifies an answer by:
  - checking that the signatures are correct;
  - checking that the witnesses are correct;
  - checking that \( I, C_1, C_2 \) are disjoint;
  - checking that, for each gap, there are enough check elements (cf. Conditions (3) and (4), above).

We note that any cryptographic accumulator scheme can be used; we chose the RSA scheme because \( |\text{Acc}(S)| \) is independent of \( |S| \) for any set \( S \).

The RSA accumulators prevent any attack that inserts elements in the intersection, because the adversary cannot prove that any such element belongs to both sets (cf. Section III-A). A malicious directory also cannot use something that is not in \( S_1 \) (resp., \( S_2 \)) as a check element for that set. As a consequence, the following holds [16]:

Claim 2: No attacker can return an incorrect intersection, unless it can break the security of the RSA accumulators or of the signature scheme.

\[^3\text{Since } h_i(x) \text{ is uniformly distributed in } 1, \ldots, m, \text{ each of the } m \text{ counters of the Bloom filter is a binomial random variable. In the proof of the claim, the Poisson distribution is used as an approximation of the binomial distribution.}\]
Proof: Suppose the attacker can return an incorrect intersection \( I' \neq I \), along with some (potentially incorrect) check value:

\[
(C'_1, C'_2, w'_1, w'_2)
\]

and the verification algorithm outputs 1.

The security of the RSA accumulators \( \text{Acc}(S_1), \text{Acc}(S_2) \) guarantees that \( (I' \cup C'_1) \subset S_1 \) and \( (I' \cup C'_2) \subset S_2 \). This implies \( I' \subset I \), therefore the attack is a deletion attack.

So, there exist one or more values \( x_1, \ldots, x_d \in I' \). For any such value \( x_l \), consider any counter \( j \) such that \( h_l(x_l) = j \), for some \( l = 1, \ldots, k \) (where \( h_1, \ldots, h_k \) are the \( k \) hash functions used by the Bloom filter). Then:

\[
\begin{align*}
\text{Bl}(I)_j &= \text{Bl}(I'_j) + \text{Bl}(I \setminus I')_j \quad (8) \\
\text{Bl}(I')_j &< \text{Bl}_j \quad (9) \\
\text{Bl}(I' \cup C'_1)_j &= \text{Bl}(S_1)_j \quad (10) \\
\text{Bl}(I' \cup C'_2)_j &= \text{Bl}(S_2)_j \quad (11)
\end{align*}
\]

where (8) is due to the definition of the counting Bloom filter and the fact that \( I' \subset I \); (9) is a consequence of (8) and the fact that \( \text{Bl}(I)_j \leq \text{Bl}_j \); equations (10) and (11) follow from the fact that Conditions (3) and (4) hold (since the client’s verification succeeds).

The fact that \( I' \cup C'_1 \subset S_1 \) and that \( \text{Bl}(I' \cup C'_1)_j = \text{Bl}(S_1)_j \) implies that \( C'_1 \) must contain the value \( x_1 \), that was removed from the intersection. For the same reason \( C'_2 \) must contains \( x_1 \). But this is a contradiction, because the verification algorithm checks that \( C'_1 \) and \( C'_2 \) are disjoint. 

We illustrate this scheme through the example in Figs. 2 and 3. When the client asks for the intersection, the directory returns \( I = \{\text{“cat”}\} \), the check element sets (\( \{\text{“dog”}, \text{“whale”}, \text{“monkey”}\} \)) and a proof (via RSA accumulators) that “cat”, “dog” are in \( S_1 \) and another proof that “cat”, “whale”, “monkey” are in \( S_2 \).

If the directory maliciously attempts to return an empty intersection, two extra gaps (2 and 4) are created for the Bloom filters. In order to justify gap 2, the directory has to find a check element for the first set that maps to index 2. Such a check element must belong to \( S_1 \); otherwise it won’t be possible to find a witness for the RSA accumulator. The only option, therefore, is to use “cat”. The directory also has to find a check element for \( S_2 \). Again, the only option is “cat”. Therefore the client will detect the attack by noticing that the same check element is returned for both sets.

A. Hashing the check elements

We here show how to reduce the size of the check elements using hashing.

In general the set elements may belong to an arbitrary domain and, therefore, be arbitrary-length bit strings. In the intersection scheme we propose, the directory must include in the reply to the query several check elements, each of which can be significantly large. The client does not really care about what those check elements are, as long as they exist.

With this in mind, we choose a cryptographic hash function (e.g., SHA-1) and then run the intersection scheme on these hashed values (and not the original elements). When a Bloom filter or an RSA accumulator is computed for set \( S \), it is actually computed using the set of the hashes of the elements of \( S \), rather than the elements themselves. The directory replies to the client with the real elements of the intersection, but it provides only the hashes of the individual check elements. This is still secure as long as the hash function is collision resistant (if not, an attacker may find two elements \( x_1, x_2 \) that map to the same value, and can thus substitute \( x_1 \) with \( x_2 \) in any set containing \( x_1 \)).

Letting \( u \) be the output length of the hash function, then we can think of the scheme as operating on \( u \)-bit elements. The collision resistance property requires \( u \) to be large enough; on the other hand the cost of the check elements is proportional to \( u \). We believe that using \( u = 160 \) (which is the output length of SHA-1) is reasonable, and we use this value in our numerical examples and experiments. Of course, our analysis holds for any value of \( u \).

B. Choosing the parameters of the Bloom filters

We consider the overhead in terms of the size of the response to the query. Recall that the response, in addition to the intersection, contains the following:

\[
(D_1, D_2, CV, \sigma_1, \sigma_2) = \text{Acc}(S_1), \text{Acc}(S_2), \text{Bl}(S_1), \text{Bl}(S_2), \text{Bl}(S_1)_j, \text{Bl}(S_2)_j, \sigma_1, \sigma_2).
\]

The elements \( \text{Acc}(S_1), \text{Acc}(S_2), \text{Bl}(S_1)_j, \text{Bl}(S_2)_j \) are 4 RSA values. The size of the two signatures \( \sigma_1, \sigma_2 \) depends on the signature scheme considered; if we assume an RSA-based scheme (using compression) then these are also RSA values. If, for example, we employ 1024-bit RSA, then the total overhead for these six values is 768 bytes. Note that this is independent of the size of the original sets or their intersection.

From the analysis in Section IV-B, the size of the compressed Bloom filter is \( z \approx mH(l) \). The encoded size of the check element set \( C_1 \) (or \( C_2 \)) is \( u \) bits for each element; if we use the upper bound of Theorem 1 as an estimate for the number of check elements, \( |C_1| = |C_2| = ml^2 = knl \). Therefore the total overhead of the protocol (in the size of the response) is given by:

\[
\frac{z}{n} + \frac{u|C_1|}{n} = k(\frac{H(l)}{l} + ul) = kf(l)
\]

(12)

(where we leave off \( o(1) \) terms). The overhead is measured as a fraction of the worst-case total number of elements (i.e., \( 2n \)) in the two original sets.

The optimal parameters are \( k = 1 \) and the choice \( l_{opt} \) of \( l \) that minimizes \( f(l) \). Note that, for \( k = 1 \), the counting Bloom filter is actually a “counting hash table”, where each bucket is replaced by the number of elements in that bucket.

For \( u = 160 \), plotting \( f \) numerically shows that it has a minimum at \( l_{opt} \approx 0.01 \), where it takes a value \( f(l_{opt}) \approx 9.7 \). This means that the overhead of the protocol is, in the worst case, approximately 9.7 bits per element in the original sets. A more careful analysis (see below) shows that the overhead decreases significantly when the size of the intersection is
large compared to the two original sets or when the sizes of $S_1$ and $S_2$ are very different; e.g., the overhead is below 2 bits per element if $|S_1|/|S_2| \approx 1000$. This may be compared with the overhead of the trivial scheme (in which the directory returns to the client the entire sets $S_1$ and $S_2$), which incurs an overhead of $u = 160$ bits per element.

For example, if the two original sets both contain $n = 100000$ elements and their intersection contains 100 elements (so we are in the worst-case scenario), then using our scheme the directory needs to send 2 Kbytes for the intersection and about 250 Kbytes for the accumulators and the Bloom filters; it saves a factor of 16 compared to the trivial scheme that would instead require 4 Mbytes, yet it still offers an arbitrarily-high level of security (for example 1024- or 2048-bit RSA security).

In a better scenario, for example $|S_1| = 100000$, $|S_2| = 100$, and $|S_1 \cap S_2| = 10$, our scheme requires 200 bytes for the intersection and about 27.5 Kbytes of overhead, compared to the 2 Mbytes of the trivial scheme.

For the optimal configuration, the Bloom filter uses $m = 100$ counters for each element in the original set. If the cost of that many counters, in terms of storage of the uncompressed Bloom filter and of processing time, is considered excessively high, then a suboptimal and larger value of the load can be used. For example, for $l = 0.03$, we can get away with $m = 33n$ counters and the overhead increases to only $f(l) = 11.3$ bits per element (plus the constant overhead due to the accumulators and the signatures).

C. Composability

In this section we show that the scheme is composable; i.e., an untrusted directory can perform a trust-preserving intersection of two sets, even if one or both of the two sets was obtained from another untrusted directory.

Consider the following example, involving four source nodes and three untrusted directories (cf. Fig. 4). The source nodes generate sets $S_1$, $S_2$, $S_3$, $S_4$ respectively. The first directory has a copy of $S_1$ and $S_2$, together with certificates $Cert_1$, $Cert_2$ for those two sets, respectively; the certificate for a set, as described in the previous sections, includes a digest (i.e., a counting Bloom filter plus an RSA accumulator) and a signature on the digest by the source. The second directory, analogously, has a copy of $S_3$ and $S_4$ and the corresponding certificates. The third directory ($d_3$) queries the first directory ($d_1$) for the intersection $S_{12} = S_1 \cap S_2$. The answer will contain a certificate for the set $S_{12}$, consisting of the certificates for the two base sets and a check value $CV_1$ (which in turn consists of the two check element sets and the two witnesses). Analogously, $d_3$ obtains $S_{34} = S_3 \cap S_4$ from $d_2$, also with a certificate. When a client ($c$) queries the third directory for $I = S_{12} \cap S_{34}$, the composability property of the scheme means that the directory can construct a certificate proving that $I$ was computed correctly, using as inputs the sets $S_{12}$ and $S_{34}$ and their certificates, without seeing any of the original sets and without having to send all of $S_{12}$ and $S_{34}$ to the client.

We illustrate the algorithm for producing such certificates for the second-level intersection, through the example in Fig. 4, with the specific sets and Bloom filters as in Fig. 5.

The directory $d_3$ knows, among other things, the check values $CV_1$ and $CV_2$. We have $CV_1 = \{C_1, C_2, w_1, w_2\}$, where $C_1 = \{\text{"dog"}\}$ and $C_2 = \{\text{"whale"}, \text{"monkey"}\}$ are the check elements for $S_1 \cap S_2$, and $w_1, w_2$ are the corresponding witnesses (e.g., $w_1$ is a witness that \{\text{"cat"}, \text{"mouse"}, \text{"horse"}, \text{"dog"}\} \subset S_1$). Analogously $CV_2 = \{C_3, C_4, w_3, w_4\}$, where $C_3$ and $C_4$ are empty.

The directory $d_3$ computes the Bloom filters $Bl(S_{12})$ and $Bl(S_{34})$ from the two sets, then it computes the check elements for the intersection using the same algorithm described in Section V for intersecting two basic sets; let $C_{12}$ and $C_{34}$ be the two check element sets thus obtained. In this example, $C_{12} = \{\text{"mouse"}\}$ and $C_{34} = \{\text{"sheep"}\}$. The check value returned by $d_3$ is given by:

$$CV = (C_1, C_2, C_3, C_4; C_{12}, C_{34}; w'_1, w'_2, w'_3, w'_4). \quad (13)$$

Here, $w'_1$ is a witness that $I \cup C_1 \cup C_{12} \subset S_1$. The directory can compute this, because it knows $w_1$ (which proves $S_{12} \cup C_1 \subset S_1$) and by construction $I \cup C_{12} \subset S_1$. The remaining $w'_i$ are analogously defined. (Here, we specifically use properties of the RSA-based accumulator.)

Note that the client can perform verification and that the scheme inherits the security guarantee of Claim 2.

We claim that the performance of the scheme does not degrade. Equation (13) shows that the size of a certificate for the set $I$ is the same as the sum of the sizes of the certificates for the two intermediate intersections, plus the second-level check element sets $C_{12}$ and $C_{34}$; those do not represent a problem, because the worst-case overhead happens exactly in
the case when their size is negligible. We will now justify this, with reference to $C_{12}$. If $S_1$ has a negligible size compared to $S_1$ and $S_2$, then the load of its Bloom filter will be very low and therefore $C_{12}$ will have almost no elements. In cases where $S_1$ is progressively larger compared to the two original sets, the size of $C_{12}$ increases, but the size of both $1$ and $2$ decreases accordingly, making the total size of the response from $d_3$ to $c$ smaller.

VI. FROM SECURE INTERSECTION TO SECURE FULL SET ALGEBRA

In Sections II through V-C, we have given a solution to the problem of secure intersection. In particular we have shown a recursive scheme that, from two sets, each accompanied by a suitable certificate, can generate a certificate for their intersection. In this section, we describe how to construct a suitable certificate for their union and their difference. With this extension, our proposed scheme allows untrusted directories to perform arbitrarily complex set operations on data from trusted nodes, and to prove the correctness of the result.

The described scheme satisfies the following important property:

Given a certificate for set $S$, it is always possible to construct $B(S)$, without knowing $S$. (14)

Note that, in the case of the intersection, this is true: if $S = S_1 \cap S_2$, the certificate of $S$ includes the Bloom filter of $S_1$ and $S_2$ or enough information to reconstruct them; it also includes the check element sets $C_1$, $C_2$ of the intersection. Therefore the Bloom filter of $S$ can be obtained as the elementwise minimum of the two vectors $B(S_1) - B(C_1)$ and $B(S_2) - B(C_2)$.

A. Performing set difference

We now show how to perform set difference, through a modified version of the example described in Figs. 4 and 5. Suppose that the query that $c$ asks of $d$ is, instead, $S_{12} \setminus S_{34} = (S_1 \cap S_2) \setminus (S_3 \cap S_4)$. The answer to the query is: $A = \{\text{"horse", "mouse"}\}$.

To prove the correctness of the answer, the directory needs to compute the check element sets $C_{12}$ and $C_{34}$ for the set difference.

- The check element set for the first set consists of all the elements of the intersection; $C_{12} = S_1 \cap S_2$. Note that the client, during verification, will reconstruct $B(S_{12})$ and $B(S_{34})$. It will notice that the counter at index 1 is equal to 1 for both Bloom filters (Fig. 5). By showing “cat”, which maps to index 1, and proving that “cat” is in the intersection, the directory assures the client that it did not delete an element that happened to map to that index from the difference.

- The check element set $C_{34}$ for the second set consists of all the elements of the opposite set difference ($S_{34} \setminus S_{12}$) that map to a gap. In this case $C_{34} = \{\text{"sheep"}\}$. By showing this element, we prove to the client that “mouse” is indeed in the set difference and not in the intersection. Note that no such proof is required for “horse”, since it does not map to a gap.

Once the check sets are known, $d_3$ can answer the query in the same form as in Fig. 4, using a check value of the form given by (13). This time, $w'_1$ is a witness that $C_1 \cup C_1 \cup A \subset S_1$ (and analogously for $w'_2$), while $w'_3$ is a witness that $C_3 \cup C_1 \cup C_3 \subset S_3$ (and analogously for $w'_4$).

B. Performing set union

Suppose the directory is queried for $S = S_1 \cup S_2$; the directory has a certificate for both $S_1$ and $S_2$; it can simply provide the two sets along with their certificates.

Note that if the result $S$ of the union is used as an operand in a subsequent operation, then Property (14) no longer holds. To fix this, we add the following rules: the certificate for $S_1 \cup S_2$ includes the check element sets $C_1$ and $C_2$ computed exactly as in the intersection case, in addition to the certificates of $S_1$ and $S_2$.

Additionally, when $S$ is later used as an operand in an intersection or in a difference, each element of $S$ that appears in the final result, or in a check element set, should carry the information about whether it belongs to $S_1$ only, or to $S_2$ only, or to both $S_1$ and $S_2$.

C. Standard Bloom filter sizes

When a source computes the digest of its set $S$, it does not know the size of the sets $S$ will be intersected with. As a practical matter, then, we introduce a set of standard
set sizes \{\tilde{n}_1, \tilde{n}_2, \ldots\} and the corresponding standard Bloom filter sizes \{\tilde{m}_1, \tilde{m}_2, \ldots\}, with the property that \(\tilde{n}_i = \tilde{m}_i / l\), and \(l\) is the desired value of the load (for example, the optimal \(l = 0.01\)). For example, we suggest the following standard set sizes \(\{\tilde{n}_1 = 10^3, \tilde{n}_2 = 10^5, \tilde{n}_3 = 10^7\}\) and a maximum set size of \(4 \cdot 10^7\). When the source generates the digest for \(S\), it generates one different Bloom filter for each standard size. In the suggested configuration, it would generate three Bloom filters with \(10^5\), \(10^7\), and \(10^9\) counters. The source signs each of these Bloom filters individually and provides all the signatures to the directory.

When the directory computes the intersection of two sets, it will choose the standard Bloom filter size that minimizes the communication overhead with the client. For example, if the two sets contain \(n = 8 \cdot 10^3\) elements each, then the choice is between a Bloom filter with \(10^5\) counters, which corresponds to a load of \(l = 0.08\) and therefore to an overhead of \(f(l) \approx 18\) or a filter with \(10^7\) counters, which corresponds to a load of \(l = 8 \cdot 10^{-4}\) and therefore to an overhead of \(f(l) \approx 12\); this second choice is obviously better.

We can extend the analysis of the overhead \(O_v\) per element (cf. Section V-B) to the case when the sizes of the two original sets are different. For \(k = 1\), we obtain:

\[
O_v = \frac{u(|C_1| + |C_2|) + z_1 + z_2}{n_1 + n_2} = \frac{2u l_2 + H(l_1) + H(l_2)}{l_1 + l_2}.
\]

Interestingly, if we are intersecting a set of size \(n_1 = 10^5\), with a set of size \(n_2 = 100\) (i.e., a much smaller set), then choosing the standard Bloom filter size of \(10^7\) leads to \(O_v = 8.1\) bits per element, while the filter size of \(10^5\) leads to the much better \(O_v = 2.2\) bits per element. The maximum overhead of the scheme (found using numerical unconstrained optimization in Matlab) happens for \(|S_1| = |S_2| \approx 4.2 \cdot \tilde{n}_i\) (for \(\tilde{n}_i = 10^3, 10^5\)) with a value of \(O_v = 12.7\) bits per element.

### VIII. EXPERIMENTAL RESULTS

We implemented a prototype of the scheme algorithms (digest, check value and verification) and used them to test the performance of the scheme. For simplicity, we tested separately the RSA accumulator computation and the remaining components of the scheme. Our experiments concentrate mostly on the intersection of two base sets, because we believe intersection to be the most challenging of the operations.

**a) Cost of RSA accumulator:** We implemented a C program that takes as input an RSA modulus \(N\), a base \(a\), and a set \(S\), and outputs the corresponding RSA accumulator. We use the PARI library [17] for big number computation.

Observe that all the principals in this scheme need this computation: the source has to compute the accumulator of the whole original set \(S\); the directory has to compute two accumulators (witnesses), for the sets \(S_1 \setminus (I \cup C_1)\) and \(S_2 \setminus (I \cup C_2)\); the client has to compute also an accumulator operation, for sets \(I \cup C_1\) and \(I \cup C_2\) (using the witnesses as a base). Note that, in practice, the source (who may know the prime factors of \(N\)) can use Chinese remaindering to speed up its computation.

We computed the accumulators for sets of different sizes. We employed a 512-bit RSA modulus and 962-bit representatives (for use in the accumulator scheme), generated using SHA-1 (see [16] for details). We ran the experiments on a 3 GHz Intel P4 with 2 GBytes RAM. Table I shows the effective CPU time required for the computation of an accumulator of one set as a function of the set size. For each set size, we repeated the experiment for 10 randomly-generated sets of strings; times were averaged. There are two columns: the first one for the source (which can employ the optimization mentioned above), the second for the directory and the client (which cannot).

As expected, the cost grows linearly with the set size. Note that the computation for the source is very efficient. Also note that, in practice, the costs at the client would be much lower because the client performs its computations on sets that are usually much smaller than the source sets.

**b) Cost of Bloom filters and check elements:** We created an unoptimized Perl implementation of the three algorithms that comprise the intersection scheme (digest, check value and verification) and used it to measure the communication overhead (encoded size of Bloom filters and check elements) and the computation overhead (CPU time to perform the algorithm). Our Perl program calls arith_coder [18] to perform Bloom filter compression.

We ran the set of experiments on the host described in the previous paragraph. Each experiment runs the four steps of the scheme (two digests, one intersection with check value computation, one verification) on a pair of random sets \(S_1, S_2\), of sizes \(n_1, n_2\), with an intersection of size \(q\). The experiment is run ten times on different pairs of sets and the results are averaged. For each experiment the number of counters \(m\) of the Bloom filters is chosen in order to obtain a value \(l_1\) of the load of \(B(S_1)\) according to Table II; this is the value that minimizes (15), for the given value of \(n_2/n_1\). In all runs, the size of the intersection is \(q = 0.01n_2\).

Table III shows the values of the running time for the computation, not including the time to compute the RSA accumulators, of (column 3 to 6): the digest of \(S_1\), the digest of \(S_2\), the operations performed by the directory (intersection and check elements) and verification.

In the third column of Table IV, we report the absolute overhead of the scheme (also accumulators excluded), i.e. the total number of bytes required to encode the two compressed

### TABLE I

<table>
<thead>
<tr>
<th>Set Size</th>
<th>Source time (sec)</th>
<th>Directory/Client time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.021</td>
<td>0.79</td>
</tr>
<tr>
<td>1000</td>
<td>0.033</td>
<td>7.8</td>
</tr>
<tr>
<td>10000</td>
<td>0.17</td>
<td>79</td>
</tr>
<tr>
<td>100000</td>
<td>1.5</td>
<td>780</td>
</tr>
</tbody>
</table>
Bloom filters of the source sets and the hashes of the check elements ($u = 160$). In the fourth column, we show the relative overhead $Ov$ as defined in Section V-B (absolute overhead divided by $n_1 + n_2$) expressed in bits per element.

Note that if the bigger set is large enough ($n_2 \geq 10000$), the measured overhead is essentially what is yielded by the analysis (15). The slightly larger overhead for $n_1 = 1000$ is due to the non-optimality of the compression algorithm, which is more evident for small input sizes.

We also ran experiments (not reported in the tables) for different values of $q$ and we noticed that this parameter has little influence on the computation times and the scheme overhead.

c) Overhead of complex queries: Finally, we ran a simple experiment to test the complete scheme, including the extensions in Sections V-C through VI-C. We generated six random sets ($S_1, \ldots, S_6$), of sizes resp. 1000, 2000, 5000, 10000, 50000, 100000, from a universe of 500000 elements (small integers); then we executed our scheme for five sample queries, using our Perl implementation. Results are shown in Table V: the second column shows the measured absolute overhead of our scheme, while the third column shows the ratio between the absolute overhead of the trivial scheme and the value in the second column. We note that, in all cases, our scheme offers significant savings.

VIII. CONCLUSIONS

In this paper we have formally defined the notion of secure set operations. Secure set operations allow any principal to perform a set operation on a pair of trusted sets, and to provide a proof of the result’s validity. This work extends work on authenticated data structures, which are limited to considering secure operations on isolated sets. We then show an efficient construction of a set operation scheme that, recursively, allows secure operations on certified results. We demonstrate, through analysis and experiments, that our scheme produces certificates that are a factor of 9 to 100 smaller than the trivial scheme in which a signed copy of all the original sets is used as a certificate. To the best of our knowledge, no other scheme that solves this problem is known. A more exhaustive description of our algorithms can be found in [16].

We believe that this scheme has an important application in the context of efficient searches in P2P systems. Clients in such systems can profit from reusing the results of previous queries, which are cached at untrusted peers. With a secure set operation scheme, a client can retrieve such a result from an untrusted peer, and use the corresponding certificate to verify that the data was not polluted during the computation. Therefore, if the client trusts the sources, then it can also trust the retrieved data.

Finally, we note that our results are not specific to result caching. The results hold for any type of set, and therefore may have other applications.

REFERENCES


APPENDIX

PROOF OF THEOREM 1

Fix a counter index $j$. What is the probability that it contains a gap? Assume that the sets $S_1$ and $S_2$ are randomly generated sets of $n_1, n_2$ elements respectively and that $|I| = q$. If we call $n$ the maximum set size that our scheme supports, then $n_1, n_2 \leq n$.

\[ Pr[\text{gap at } j] = Pr[\text{Bl}(S_1 \setminus I)_j > 0 \wedge \text{Bl}(S_2 \setminus I)_j > 0] \]

\[ = Pr[\text{Bl}(S_1 \setminus I)_j > 0] \cdot Pr[\text{Bl}(S_2 \setminus I)_j > 0] \]

where we used the fact that the elements of $S_1 \setminus I$ and $S_2 \setminus I$ are distinct, therefore the two events are independent. Following [14], we have

\[ Pr[\text{Bl}(S_1 \setminus I)_j > 0] = \left[ 1 - \left( 1 - \frac{1}{m} \right)^{k(n_1 - q)} \right] \]

\[ \approx \left[ 1 - e^{-\frac{k(n_1 - q)}{m}} \right] \leq \left[ 1 - e^{-\frac{k_1}{m}} \right] \]

Any gap $j$ requires $\text{Bl}(S_1 \setminus I)_j$ check elements for $S_1$ and $\text{Bl}(S_2 \setminus I)_j$ check elements for $S_2$. In the worst case, no check element will be useful to cover more than one gap. Therefore, the expected number of check elements in $C_1$ needed for an index $j$ is given by:

\[ Pr[\text{gap in } j] E[\text{Bl}(S_1 \setminus I)_j | \text{gap at index } j] = \]

\[ Pr[\text{Bl}(S_1 \setminus I)_j > 0] Pr[\text{Bl}(S_2 \setminus I)_j > 0] \]

\[ E[\text{Bl}(S_1 \setminus I)_j] E[\text{Bl}(S_2 \setminus I)_j] = \]

\[ Pr[\text{Bl}(S_1 \setminus I)_j > 0] \leq (1 - e^{-l_2}) l_1 \]

\[ \leq l_1 l_2. \]

The claim follows from linearity of expectation.