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 of 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 trusted by a.**

Our main contributions are a formal definition of the problem, and an efficient scheme that solves this problem by allowing untrusted peers to perform set operations on trusted data sets, and to produce 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 can be applied to other applications as well. We give an analysis that demonstrates the low overhead of the scheme and we include experimental data which confirm the analysis.

***Index Terms*— Security, peer-to-peer, trusted computation, simulations, system design.**

I. Introduction

This paper describes a method of 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 result is more general than result caching, and can potentially be used in other areas.

For purposes of this work, we define a P2P network as a large, distributed, and decentralized structure that allows 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) (Chord [1], CAN [2], Tapestry [3], Pastry [4] etc.), for example, 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 $O(\log n)$ or so overlay 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, makes 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 incrementally created and maintained indexes. Each "index" (or *attribute index*) records nodes that have 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 requires 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. The key feature is that only conjunctions of two or more attributes are 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].

Our motivation is the problem of making such cached results *secure*. Our main contribution is to show an efficient scheme that allows an untrusted peer to perform set

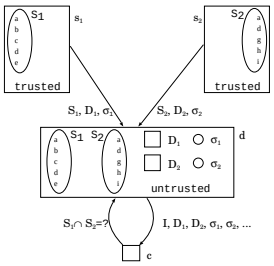

Fig. 1: The problem: an untrusted node computes intersection of two sets obtained from two trusted nodes. D_1 and D_2 are the digests of S_1 and S_2 respectively. σ_1 (resp. σ_2) is s_1's (resp. s_2's) signature on D_1 (resp. D_2). I is the intersection of S_1 and S_2.
figure

operations on trusted data sets, and to supply an unforgeable proof to the client of the correctness of the result. The approach we take is to reduce the trustworthiness of the derived value (the cached result) to a function of the trustworthiness of the sources (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.

We briefly introduce an example of the problem we are trying to solve, along with some terminology. We model each index or cached query as a set of elements. The operations we wish to perform are standard set operations, such as union, difference, and intersection.

Two trusted *source* nodes, s_1 and s_2, each store an index in the form of a set S_1 and S_2 (stored by s_1 and s_2, respectively, cf. Fig. 1). An untrusted *directory* d caches the result of some set operation on S_1 and S_2; for this example, we assume it stores their intersection. The following also assumes that the computation of $S_1 \cap S_2$ is performed by d, although this need not necessarily be the case.

The problem that we analyze in this paper 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 these public keys. We do not necessarily require the existence of a public key infrastructure; the public key will be tied to the concept of identity in the system. The fact that c knows and trusts s_1 and s_2 implies that c also knows their 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* of S_1, which contains authenticity information, together with a signature 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, of S_1, and the digest signature, σ_1, on D_1.

When d is queried for the intersection of the two sets, it provides 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 (at least at the time it was created).

The dangers are that a malicious directory could insert extraneous elements in the reply, (but still include all the elements of the intersection; we will call this an *insertion attack*), or it could fail to include elements that really are in the intersection (without inserting; we will call this a *deletion attack*). More generally, it could both insert and delete elements.

As a trivial solution, the two sources could sign the respective sets. The directory could then provide both of the original sets in response to the query, together with the signatures. (This is equivalent to choosing $D_1 = S_1$ and $D_2 = S_2$.) This is secure, because the client would be able to verify the authenticity of the two original sets and compute the intersection himself. Unfortunately, this solution is very expensive, because the two original sets might be much bigger than their intersection, so the communication overhead for the response may be excessive.

Our detailed study shows that, of the three basic set operations, intersection is indeed the most challenging to compute in provably correct fashion. We therefore first solve the intersection problem, and then show how to construct solutions for the case of union and set difference.

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 a first scheme to solve the intersection problem and we discuss its limitations. In Section V, we give a secure and efficient intersection scheme that solves the problems of the previous one. In Section V-C, we show how to make the scheme composable, that is to say, how to securely compute intersections of sets that are themselves intersections performed by other untrusted nodes. In Section VI we show how the scheme can be easily extended to include the other two sets operations, union and difference. In Section VI-C we discuss implementation details. Finally, in Section VII, we give experimental results showing the efficiency of our approach and in Section IX we summarize and conclude.

II. Defining an intersection scheme

We define an intersection scheme formally in order to make it usable as a cryptographic primitive, like signature schemes or encryption schemes. As mentioned above, the secure computation of intersection is the most challenging of the secure set operations; it is easy to extend this definition to include union and difference.

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

- Dgst takes as input a set S of elements in some universe $\mathcal{U}$ and outputs a bit string $D \leftarrow \mathsf{Dgst}(S)$ 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 $C \leftarrow \mathsf{CV}(S_1, S_2, D_1, D_2)$, 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 $b = \mathsf{Vrfy}(D_1, D_2, I, C)$.
- The following correctness property holds. For all S_1, S_2, I if:

$$D_1 \leftarrow \mathsf{Dgst}(S_1) \wedge D_2 \leftarrow \mathsf{Dgst}(S_2)$$
$$\wedge\, C \leftarrow \mathsf{CV}(S_1, S_2, D_1, D_2) \wedge I = S_1 \cap S_2$$

  then $\mathsf{Vrfy}(D_1, D_2, I, C) = 1$.
- The following security property holds. It is computationally infeasible for an adversary, on input S_1, S_2, D_1, D_2 to find a set I' and a value C', such that $I' \neq I$ and $\mathsf{Vrfy}(D_1, D_2, I', C') = 1$.

In words an intersection scheme is used as follows. In the setting of the introduction, s_1 computes $D_1 \leftarrow \mathsf{Dgst}(S_1)$ and a signature σ_1 on D_1 (computed using its private key); s_2 computes $D_2 \leftarrow \mathsf{Dgst}(S_2)$ and σ_2 analogously. d has a copy of D_1 and D_2, together with signatures σ_1, σ_2 plus a copy of S_1 and S_2 (Fig. 1). When c queries d, then a honest d computes $I = S_1 \cap S_2$ and $C \leftarrow \mathsf{CV}(S_1, S_2, D_1, D_2)$; finally d sends to c the values $(I, D_1, \sigma_1, D_2, \sigma_2, C)$.

Node c, in response to its query, receives $(I', D_1, \sigma_1, D_2, \sigma_2, C')$. If d is malicious, then I' and C' might be different from I and C. The signature scheme prevents the attacker d from lying on the remaining values. The client first checks that the signatures are valid and then it runs $\mathsf{Vrfy}(D_1, D_2, I', C')$; this will output 1, if the answer to the query is correct, by the definition above; we would like this to return 0 if $I' \neq I$. Indeed the security property states that it is hard for the directory to find an incorrect response that passes the verification test.

III. Background

In this section we briefly summarize two existing techniques that we will use in the 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 [6], [7], [8] has shown how to use *cryptographic accumulators* to solve a 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.

To allow for this, s computes the value of a cryptographic accumulator $\mathsf{Acc}(S)$ from the set S, signs it and sends 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. Then it shows $\mathsf{Acc}(S)$, w_i and the signature on $\mathsf{Acc}(S)$. The client, after verifying the signature, can verify from $\mathsf{Acc}(S), w_i, x_i$ that the answer is correct.

The work cited above pertains to the RSA accumulator, which has the important property that its size does not depend on the size of the set S.

Let $S = \{x_1, \ldots, x_n\}$ be the set stored at a source node. Each x_i is represented by u bits. The source chooses a random RSA modulus N and a random $a \in \mathbb{Z}_N^*$, then it makes those public information [1].

Suppose there exists an efficient algorithm R that, on input of a u-bit element x produces an odd integer $e = R(x)$, called a *representative* of x. We require that the algorithm R implements a *division intractable function* [9]. 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 $R(x_1) \cdot \cdots \cdot R(x_n)$. Appendix I discusses two possible constructions for R.

In order to compute the accumulator of S, compute representatives $e_1, \ldots, e_n$ of $x_1, \ldots, x_n$, then output:

$$\mathsf{Acc}(S) = a^{e_1 e_2 \ldots e_n} \mod N. \tag{1}$$

In order to compute the witness w_i of x_i in S, output:

$$w_i = a^{e_1 e_2 \ldots e_{i-1} e_{i+1} \ldots e_n} \mod N.$$

[1] $\mathbb{Z}_N^*$ is the set of integers between 1 and $N-1$, that are relatively prime with N

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

$$w_i^{e_i} = \mathsf{Acc}(S) \mod N.$$

It can be proved ([7], [9]) 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 $w^{R(x)} = \mathsf{Acc}(S) \bmod N$.

B. Counting Bloom filters

We assume that the reader has some basic familiarity with Bloom filters [10]. 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. This does not prevent the attacker from returning a subset of the real intersection, so Bloom filters are not the whole solution. Also, the Bloom filter would need to be prohibitively large in order to ensure that an adversary would not be able to find a false positive in either of the two original sets.

Our protocol employs a generalization of Bloom filters, called *counting Bloom filters* [11], for a different purpose. A counting Bloom filter is a data structure characterized by two parameters k and m. It requires k independent hash functions, which we denote $h_1, \ldots, h_k$, with range in $\{1, \ldots, m\}$. In particular, given a set S, the counting Bloom filter of S (denoted by $\mathsf{BI}(S)$) is a vector of m non-negative integers (counters). It can be incrementally constructed, starting with a vector of all zeros and then, for each element x in S, by incrementing the counters with indices $h_1(x), \ldots, h_k(x)$. We denote by $\mathsf{BI}(S)_j$ the j-th counter of the filter; it represents the number of elements that hash to that index.

IV. A first attempt

In this section we show an intersection scheme based on counting Bloom filters. This scheme, as we will discuss, is neither secure nor efficient, but it will be useful in understanding the main idea of the correct secure intersection scheme.

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 $\mathsf{BI}(S_1)$, the element "dog" appears in the label of the fourth and sixth

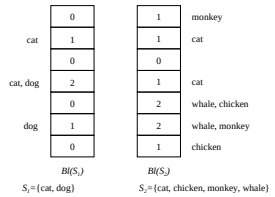

Fig. 2: Counting Bloom filters of two sets
figure

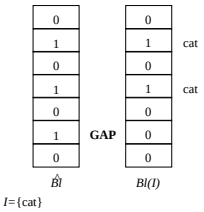

Fig. 3: The elementwise minimum, the Bloom filter of the intersection and the gap for the example in Fig. 2
figure

counters, meaning that, for example, $h_1(\text{"dog"}) = 4$ and $h_2(\text{"dog"}) = 6$.

The client receives a signed copy of these two Bloom filters, together with the supposed intersection I' (for a malicious directory, this could be different from the real intersection I). To verify correctness, the client computes the Bloom filter $\widehat{\mathsf{BI}}$, obtained as the element-by-element minimum of $\mathsf{BI}(S_1)$ and $\mathsf{BI}(S_2)$, and the Bloom filter $\mathsf{BI}(I')$ of the returned intersection (Fig. 3). The condition:

$$\mathsf{BI}(I)_j \leq \widehat{\mathsf{BI}}_j \quad \forall j = 1, \ldots, m. \tag{2}$$

holds for the correct intersection, because each element in I also belongs to both S_1 and S_2; so the client can check this condition using I' instead of I.

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 $\widehat{\mathsf{BI}}$ (same figure, on the left).

This test does not prevent an attacker from performing a deletion attack, i.e. returning an I' that is a subset of I (in the 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 $\widehat{\mathsf{BI}}$ and $\mathsf{BI}(I')$ were equal, then we would know for certain that a deletion attack was not performed (otherwise, without the attack the Bloom filter of the intersection would have some counters strictly greater than the corresponding in $\widehat{\mathsf{BI}}$, which is a contradiction). 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 $\widehat{\mathsf{Bl}}$. In our example, the counter with index 6 of $\mathsf{Bl}(I)$ is 0 while the corresponding in $\widehat{\mathsf{Bl}}$ is 1; this is due to the fact that there are at least one element ("dog") which are only in the first set and at least one element ("whale" and "monkey") which are only in the second set, all of which hash to the same index 6. We will say that index 6 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 $\mathsf{Bl}(I)_j$ is strictly less than $\widehat{\mathsf{Bl}}_j$.

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 the 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 $(I', \mathsf{Bl}(S_1), \sigma_1, \mathsf{Bl}(S_2), \sigma_2, C_1', C_2')$, the client c computes the Bloom filter $\widehat{\mathsf{Bl}}$ obtained as the element-by-element minimum of $\mathsf{Bl}(S_1)$ and $\mathsf{Bl}(S_2)$. Here we denoted with I', C_1', C_2' the values that d has returned for the intersection and the check elements; if d is malicious, those can be different from the I, C_1, C_2 an honest node would return.

The client c will check that condition (2) holds and also will check that for each $j = 1, \ldots, m$ such that $\mathsf{Bl}(I)_j < \widehat{\mathsf{Bl}}_j$:

$$\mathsf{Bl}(I')_j + \mathsf{Bl}(C_1')_j = \mathsf{Bl}(S_1)_j \tag{3}$$

$$\mathsf{Bl}(I')_j + \mathsf{Bl}(C_2')_j = \mathsf{Bl}(S_2)_j \tag{4}$$

and will reject if any of these is not satisfied. Condition (3) states that for each gap j the check element set C_1 must contain some elements $\{y_1, \ldots, y_s\}$ from $S_1 \setminus I$ that collectively hash $\mathsf{Bl}(S_1 \setminus I)_j$ times to index j. That is to say:

$$\mathsf{Bl}(S_1 \setminus I)_j = \mathsf{Bl}(\{y_1, \ldots, y_s\})_j.$$

Analogously (4) states that C_2 must contain some elements from $S_2 \setminus I$ that hash $\mathsf{Bl}(S_2 \setminus I)_j$ times to index j.

In the next subsection we state some results that estimate the number of gaps and the number of check elements needed by the scheme. In Section IV-B we use this result to estimate the level of security and efficiency of the scheme and in Section V, we will show how to make an efficient scheme out of this.

A. Number of gaps and check elements

Definition 3—Load of the filter: The *load* of a counting Bloom filter $\mathsf{Bl}(S)$ 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.

$$l = \frac{kn}{m}. \tag{5}$$

Consider the counting Bloom filter intersection scheme, where the two sets S_1 and S_2 have respectively n_1 and n_2 elements each 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 m be the number of counters and k the number of hash functions. Let $l_1 = \frac{kn_1}{m}$, $l_2 = \frac{kn_2}{m}$ be the loads of $\mathsf{Bl}(S_1)$, $\mathsf{Bl}(S_2)$ 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 $\{h_i(x_a) : i = 1, \ldots, k \wedge a = 1, \ldots, n\}$ are independent random variables uniformly distributed in $\{1, \ldots, m\}$.

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

$$E[|C_1|] \leq m(1 - e^{-l_2})l_1 \leq ml_1l_2 \leq ml^2 \tag{6}$$

$$E[|C_2|] \leq m(1 - e^{-l_1})l_2 \leq ml_1l_2 \leq ml^2. \tag{7}$$

A proof can be found in Appendix II.

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 [12]. We can reduce the size of the counting Bloom filters by applying a compression algorithm. This

is still not sufficient to make the intersection scheme secure and efficient, but 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 $\mathsf{Bl}(S)$ 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

$$z = mH(l) \tag{8}$$

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

We believe the bound is tight. Proof available in Appendix IV.

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:

- Source s_i, to produce the digest of a set S_i, generates:
  - the compressed counting Bloom filter $\mathsf{Bl}(S_i)$;
  - the RSA accumulator $\mathsf{Acc}(S_i)$;
  - a signature σ_i on both.
- When a directory receives a query, it returns:
  - the intersection I;
  - the Bloom filters $\mathsf{Bl}(S_1)$, $Bl(S_2)$;
  - the RSA accumulators $\mathsf{Acc}(S_1), \mathsf{Acc}(S_2)$;
  - 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 σ_1, σ_2.
- The client verifies an answer by:
  - checking that the signatures are correct;
  - checking that the witnesses are correct;
  - checking that, for each gap, there are enough check elements (3, 4).

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 (Section III-A). Also the malicious directory cannot use something that is not in S_1 (S_2) as a check element for that set. As a consequence, the following holds:

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

Proof in Appendix V.

We illustrate this scheme through the example in Fig. 2 and 3. When the client asks for the intersection, the directory returns $I = \{$"cat"$\}$, the check element sets ($\{$"dog"$\}$, $\{$"whale", "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, and we introduce the parameter u (the size of an element).

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 hash function (for example, SHA-1), then we map each element in the universe to its hash and we run the intersection scheme, not on the original elements, but on their hashes. When a Bloom filter or an RSA accumulator is computed for set S, it is actually computed for the set of the hashes of the elements of S. The directory replies to the client with the *real elements* of the intersection, but it provides only the hashes of the individual check elements. The hash function must be collision resistant, because, if the attacker could find two elements X_1, X_2 that map to the same value, it could substitute X_1 with X_2 in any set that contains X_1.

If we call u the length of the output of the hash function, then we can think of the scheme as operating on elements of u bits. 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 $u = 160$ (SHA-1 output size) be a reasonable compromise; we will use this value in numerical examples and experiments, but our analysis holds for any value of u.

B. Choosing the parameters of the Bloom filters

Let us consider the overhead of the protocol in the size of the response to the query. Recall that the response, in addition to the intersection, contains the following parts:

$$(D_1, D_2, CV, \sigma_1, \sigma_2) = \\ (\mathsf{Acc}(S_1), \mathsf{Bl}(S_1), \mathsf{Acc}(S_2), \mathsf{Bl}(S_2), C_1, C_2, w_1, w_2, \sigma_1, \sigma_2)$$

The elements $\mathsf{Acc}(S_1), \mathsf{Acc}(S_2), w_1, w_2$ are 4 RSA values. If, for example, we employ 1024 bit RSA, then the total overhead for those values is 512 bytes. The two signatures σ_1, σ_2 have a size that depends on the signature scheme considered. For example, they could be 1024 bits each for RSA signatures, bringing the total of these 6 components to 768 bytes.

From the analysis in Section IV-B, the size of the compressed Bloom filter is $z = mH(l)$. 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, $|C_1| = ml^2 = knl$. Therefore the total overhead of the protocol (in the size of the response) is given by:

$$Ov = \frac{z}{n} + \frac{u|C_1|}{n} = k(\frac{H(l)}{l} + ul) = kf(l) \qquad (9)$$

The overhead of the protocol is evaluated as a fraction of the total number of elements $2n$ 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 hash table, where each bucket is replaced by the number of elements in the bucket.

For $u = 160$, plotting f numerically shows that it has a minimum approximately at $l_{opt} = 0.01$, for which it takes a value $f(l_{opt}) \approx 9.7$. This means that the overhead of the protocol is, *in the worst case*, approximately of 9.7 bits per element in the original set. In practice, a more careful analysis (12) shows that the overhead decreases significantly whenever the size of the intersection is non-negligible compared to the two original sets or when the sizes of S_1 and S_2 are quite different from each other; e.g. it's below 2 bits per element if $n_1/n_2 = 1000$. Compare this with the overhead of the trivial protocol, in which the directory returns to the client the entire sets S_1 and S_2 (signed by the sources), which incurs in a overhead of $u = 160$ bits per element.

For example, if the two original sets contain both $n = 100000$ elements, the intersection contains 100 elements (so we are in the worst-case scenario), in 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 and it still offer an arbitrarily high level of security (for example 1024 or 2048 bit strong RSA security). In a better scenario, for $|S_1| = 100000$, $|S_2| = 100$ and $|I| = 10$, 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 $\frac{m}{n} = 100$ 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 only to $f(l) = 11.3$ 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 by another untrusted directory, as a different trust-preserving intersection.

Consider the following example, involving four source nodes and three untrusted directories (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 consists, as described in the previous sections, in the digest of the set (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 of 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

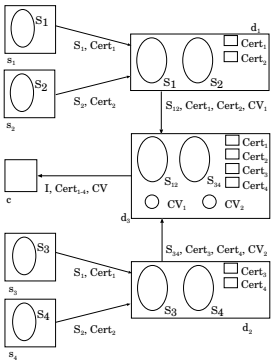

Fig. 4: Four trusted nodes (s_1 through s_4) and three untrusted directories (d_1 through d_3) involved in a two-level intersection computation.
figure

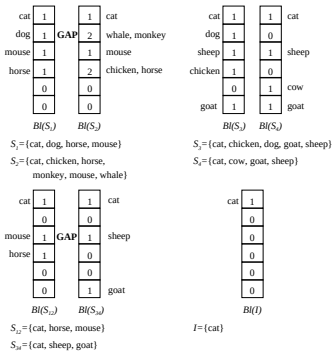

Fig. 5: An example of the scheme of an intersection of intersections. The figure shows the four original sets, the two intermediate intersections and the final intersection, together with the corresponding Bloom filters
figure

as inputs the sets S_{12} and S_{34} and their certificates, without seeing any of the original sets and without having to send the whole S_{12} and S_{34} to the client.

We now illustrate the algorithm to produce such certificate 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 witnesses 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 $\mathsf{Bl}(S_{12})$ and $\mathsf{Bl}(S_{34})$ 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:

$$CV = (C_1, C_2, C_3, C_4; C_{12}, C_{34}; w'_1, w'_2, w'_3, w'_4). \quad (10)$$

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 $I \cup C_{12} \subset S_{12}$. The remaining w'_i are analogously defined.

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 (10) 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_{12} 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_{12} is progressively larger compared to the two original sets, the size of C_{12} increases, *but the size of both C_1 and C_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 the set S, it is always possible to construct $\mathsf{Bl}(S)$, without knowing S (11)

Note that, in the case of the intersection, this is true: if $S = S_1 \cap S_2$, the certificate of S includes either 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 $\mathsf{Bl}(S_1) - \mathsf{Bl}(C_1)$ and $\mathsf{Bl}(S_2) - \mathsf{Bl}(C_2)$.

A. Performing set difference

We now show how to perform set difference, through a modified version of the example described in Fig. 4 and 5. Suppose that the query that c asks of d is, instead, $S_{12} \backslash S_{34} = (S_1 \cap S_2) \backslash (S_3 \cap S_4)$. The answer to the query is: $A = \{$"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*: $C_{12} = S_{12} \cap S_{34}$. In this case $C_{12} = \{$"cat"$\}$. Note that the client, during verification, will reconstruct $\mathsf{BI}(S_{12})$ and $\mathsf{BI}(S_{34})$. 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 ($S_{34} \backslash S_{12}$) that map to a gap. In this case $C_{34} = \{$"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 from c in the same form as in Fig. 4, with a check value of the form (10), but this time w'_1 is a witness that $C_1 \cup C_{12} \cup A \subset S_1$ and analogously w'_2, while w'_3 is a witness that $C_3 \cup C_{12} \cup C_{34} \subset S_3$ and analogously 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.

The catch is that, if the result S of the union is used as operand in a subsequent operations, then Property (11) does not hold anymore. To fix this, we add the following rule: 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 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* $\{\hat{n}_1, \hat{n}_2, \dots\}$ and the corresponding *standard Bloom filter sizes* $\{\hat{m}_1, \hat{m}_2, \dots\}$, with the property that $\hat{m}_i = \hat{n}_i / l$, 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 $\{\hat{n}_1 = 10^3, \hat{n}_2 = 10^5, \hat{n}_3 = 10^7\}$ 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 respectively 10^5, 10^7, 10^9 counters. The source will then sign each of these Bloom filters individually and provide all of 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 per element, (Section V-B) to handle the case when the size of the two original sets are different. We obtain, for $k = 1$:

$$Ov = \frac{u(|C_1| + |C_2|) + z_1 + z_2}{n_1 + n_2} = \\ = \frac{2ul_1l_2 + H(l_1) + H(l_2)}{l_1 + l_2}. \quad (12)$$

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

VII. Experimental results

We implemented a prototype of the scheme algorithms (digest, check value and verification) and we used them to test the performance of the scheme. For simplicity, we tested separately the RSA accumulator computation and

TABLE I: Time required to compute an RSA accumulator
table

Set Size	Source time (sec)	Directory/Client time (sec)
100	0.021s	0.79
1000	0.033s	7.8
10000	0.17s	79
100000	1.5s	780

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 [13] 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). Analogously to what is done for RSA signatures, note that the source can store the prime factors p_1, p_2 of the modulus N, taking care to keep them secret. Instead of using (1), it can, then, compute:

$$\mathsf{Acc}(S) \bmod p_i = a^{e_1 e_2 \ldots e_n \bmod (p_i - 1)} \bmod p_i$$

for $i = 1, 2$, and reconstruct $\mathsf{Acc}(S)$ from $\mathsf{Acc}(S) \bmod p_1$ and $\mathsf{Acc}(S) \bmod p_2$ using Chinese remaindering.

We computed the accumulators for sets of different sizes. We employed a 512-bit RSA modulus and 962-bit representatives, generated with SHA-1 (Appendix I). 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 representing small integers; 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` [14] to perform Bloom filter compression.

TABLE II: Optimal Bloom filter load for different ratios of the sizes of the two source sets
table

n_2/n_1	l_1
1	0.010
0.1	0.065
0.01	0.51
0.001	2.9

TABLE III: Execution times of the scheme algorithms (accumulator excluded) in seconds
table

n_1	n_2/n_1	dig. S_1	dig. S_2	inters.	verif.
1000	1.000	0.449	0.446	0.137	0.413
	0.100	0.169	0.151	0.112	0.168
	0.010	0.126	0.105	0.125	0.126
	0.001	0.122	0.099	0.107	0.111
10000	1.000	3.400	3.630	0.681	2.720
	0.100	0.881	0.651	0.421	0.671
	0.010	0.389	0.167	0.346	0.275
	0.001	0.304	0.098	0.312	0.159
100000	1.000	34.10	35.20	6.520	24.10
	0.100	7.570	4.940	3.470	5.180
	0.010	3.300	0.736	3.130	1.900
	0.001	2.430	0.201	2.710	0.705

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 $\mathsf{Bl}(S_1)$ according to Table II; this is the value that minimizes (12), for the given value of n_2/n_1. In all runs, the size of the intersection is $q = 0.01 n_2$.

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

TABLE IV: Communication overhead (accumulator excluded)
table

n_1	n_2/n_1	Absolute overhead (bytes)	Relative overhead (bits)
1000	1.000	3431	13.7
	0.100	1725	12.5
	0.010	1042	8.26
	0.001	478	3.82
10000	1.000	27533	11.0
	0.100	12450	9.06
	0.010	7025	5.56
	0.001	3158	2.52
100000	1.000	269572	10.8
	0.100	118924	8.65
	0.010	64699	5.12
	0.001	27334	2.18

overhead of the scheme (also accumulators excluded), i.e. the total number of bytes required to encode the two compressed 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 ($n_1 \geq 10000$), the measured overhead is essentially what is yielded by the analysis (12). 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.

TABLE V: Overhead of complex queries
table

Query	Absolute overhead (bytes)	Saving factor
$S_1 \setminus S_2$	4457	13.5
$(S_5 \cap S_6) \setminus (S_1 \cup S_2)$	209633	109.0
$(S_1 \cap S_4) \cap S_3$	17880	17.9
$(S_5 \setminus S_4) \cap S_2$	105062	11.8
$(S_3 \cup (S_5 \cap S_2)) \setminus S_5$	122927	9.3

VIII. Related work

Our work may be viewed as extending the concept of *authenticated data structures* [7], [15], [8], [16], [17]. 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 $S_1 \cap S_2$").

Reference [18]

IX. 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. 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 Appendix **??**.

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

[1] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[2] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, "A scalable content addressable network," in *Proceedings of ACM SIGCOMM 2001*, 2001.

[3] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Tech. Rep. UCB/CSD-01-1141, UC Berkeley, Apr. 2001.

[4] Antony Rowstron and Peter Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001, pp. 329–350.

[5] Bobby Bhattacharjee, Sudarshan Chawathe, Vijay Gopalakrishnan, Pete Keleher, and Bujor Silaghi, "Efficient peer-to-peer searches using result-caching," in *The 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, February 2003.

[6] Jan Camenisch and Anna Lysyanskaya, "Dynamic accumulators and application to efficient revocation of anonymous credentials," in *CRYPTO*, 2002.

[7] Niko Baric and Birgit Pfitzmann, "Collision-free accumulators and fail-stop signature schemes without trees," in *EUROCRYPT*, 1997, pp. 480–494.

[8] M. Goodrich, A. Schwerin, and R. Tamassia, "An efficient dynamic and distributed cryptographic accumulator," Tech. Rep., Johns Hopkins Information Security Institute, 2000.

[9] Rosario Gennaro, Shai Halevi, and Tal Rabin, "Secure hash-and-sign signatures without the random oracle," in *EUROCRYPT*, 1999, pp. 123–139.

[10] Burton H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[11] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.

[12] M. Mitzenmacher, "Compressed bloom filters," in *20th Annual ACM Symposium on Principles of Distributed Computing*, 2001, pp. 144–150, To appear in IEEE/ACM Trans. on Networking.

[13] The PARI Group, Bordeaux, *PARI/GP, Version 2.1.5*, 2000, available from `http://www.parigp-home.de/`.

[14] John Carpinelli et al., "arith_coder: Word, character, integer, and bit based compression using arithmetic coding," `http://www.cs.mu.oz.au/~alistair/arith_coder/`, 1999.

[15] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine, "Authentic third-party data publication," in *IFIP Conf. Database Security*, 2000.

[16] M.T. Goodrich and R. Tamassia, "Efficient authenticated dictionaries with skip lists and commutative hashing," Tech. Rep., Johns Hopkins Information Security Institute, 2000.

[17] M.T. Goodrich, R. Tamassia, N. Triandopoulos, and R. Cohen, "Authenticated data structures for graph and geometric searching," in *CT-RSA*, 2003.

[18] Premkumar Devanbu, Michael Gertz, Charles Martel, and Stuart G. Stubblebine, "Authentic data publication over the internet," *Journal of Computer Security*, vol. 11, pp. 291–314, 2003.

[19] Jean-Sbastien Coron and David Naccache, "Security analysis of the gennaro-halevi-rabin signature scheme," in *EUROCRYPT*, 2000, pp. 91–101.

[20] Mihir Bellare and Phillip Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *ACM Conference on Computer and Communications Security*, 1993, pp. 62–73.

[21] R. J. Evans, "The entropy of a poisson distribution," *SIAM Review*, vol. 30, pp. 314–315, 1988.

APPENDIX I
REPRESENTATIVE GENERATION ALGORITHMS

We now mention two efficient algorithms for generation of representatives.

The first algorithm is suggested in [9] and consists in repeated applications of SHA-1.

$$R(x) = 1|SHA1(x|1)|\dots|SHA1(x|r)|1. \qquad (13)$$

Reference [19] analyzes this representative algorithm, when SHA-1 is replaced by a random oracle [20]; their conclusion is that, in order to obtain division intractability with a level of security comparable with 1024 bit RSA, the length of the output should be approximately 1000 bits long or more. So we can set $r = 6$, which implies 962 bit representatives.

The second algorithm follows the idea suggested in [9] of choosing a representative algorithm that only outputs primes and is collision-resistant, but with a different construction. To obtain $R(x)$, compute:

$$\begin{aligned} y_1 &= SHA1(x|1)|1 \\ y_2 &= SHA1(x|2)|1 \\ y_3 &= SHA1(x|3)|1 \\ &\dots \end{aligned}$$

until you find a value y_i, which is prime; then set $R(x) = y_i$.

If we assume that SHA-1 is a random oracle, taking into account that the number of 161-bit primes is approximately

$$\frac{2^{161}}{\ln(2^{161}) - 1} \approx \frac{2^{161}}{110}$$

then computing a representative takes, on the average, $\frac{110}{2} = 55$ SHA-1 computations and primality tests on 161 bit inputs.

In the setting of Section III-A, note that only the source needs to pay this cost. When the source sends a copy of the set to the directory, it can include, for each element x the corresponding value of i, for which $R(x) = SHA1(x|i)$ is prime; therefore the directory can compute the representatives paying only one SHA-1 computation. Similarly the directory can provide to the client the value of i, when she is required to prove that x is in the set; the client can, then, compute the representative with one

SHA-1 computation and one primality test (note that the directory can maliciously lie on the value of i, so it's important that the client checks that $SHA1(x|i)$ is indeed prime). The value of i can be encoded in, for example, one byte, so it introduces a small communication overhead.

APPENDIX II
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 $n_1, n_2 \leq n$.

$$\begin{aligned}\Pr[\text{gap at } j] = \Pr[\mathsf{Bl}(S_1 \backslash I)_j > 0 \wedge \mathsf{Bl}(S_2 \backslash I)_j > 0] = \\ = \Pr[\mathsf{Bl}(S_1 \backslash I)_j > 0] \cdot \Pr[\mathsf{Bl}(S_2 \backslash I)_j > 0]\end{aligned}$$

where we used the fact that the elements of $S_1 \setminus I$ and $S_2 \backslash I$ are distinct, therefore the two events are independent. Remembering the derivation in [11]

$$\begin{aligned}\Pr[\mathsf{Bl}(S_1 \backslash I)_j > 0] = [1 - (1 - \frac{1}{m})^{k(n_1 - q)}] \\ \simeq [1 - e^{-\frac{k(n_1-q)}{m}}] \leq [1 - e^{-\frac{kn_1}{m}}] \leq [1 - e^{-\frac{kn}{m}}].\end{aligned}$$

Any gap j requires $\mathsf{Bl}(S_1 \setminus I)_j$ check elements for S_1 and $\mathsf{Bl}(S_2 \backslash I)_j$ 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:

$$\begin{aligned}\Pr[\text{gap in } j]\mathrm{E}[\mathsf{Bl}(S_1 \backslash I)_j | \text{gap at index } j] = \\ \Pr[\mathsf{Bl}(S_1 \backslash I)_j > 0] \Pr[\mathsf{Bl}(S_2 \backslash I)_j > 0] \\ \mathrm{E}[\mathsf{Bl}(S_1 \backslash I)_j | \mathsf{Bl}(S_1 \backslash I)_j > 0] = \\ \Pr[\mathsf{Bl}(S_1 \backslash I)_j > 0] \Pr[\mathsf{Bl}(S_2 \backslash I)_j > 0] \\ \frac{\mathrm{E}[\mathsf{Bl}(S_1 \backslash I)_j]}{\Pr[\mathsf{Bl}(S_1 \backslash I)_j > 0]} = \\ \Pr[\mathsf{Bl}(S_2 \backslash I)_j > 0]\mathrm{E}[\mathsf{Bl}(S_1 \backslash I)_j] \leq (1 - e^{-l_2})l_1 \\ \leq l_1 l_2.\end{aligned}$$

The claim follows from linearity of expectation.

APPENDIX III
ASYMPTOTIC BEHAVIOR OF THE ENTROPY OF POISSON DISTRIBUTION

If we expand the definition of $H(l)$ (15)

$$\begin{aligned}H(l) = -\sum_{i=1}^{+\infty} \frac{l^i}{i!} e^{-l} i \log l \\ + \sum_{i=1}^{+\infty} \frac{l^i}{i!} e^{-l} i \log(i!) + \sum_{i=1}^{+\infty} \frac{l^i}{i!} e^{-l} l \log(e)\end{aligned}$$

and then we consider $l \to 0^+$, the order of the three terms is:

$$\begin{aligned}H(l) = \left[-le^{-l} \log l + O(l^2 \log l)\right] \\ + \left[\frac{1}{2} l^2 e^{-l} + O(l^3)\right] + \left[le^{-l} \log e + O(l^2)\right] \\ = -l \log l + O(l). \quad (14)\end{aligned}$$

[21] analyzes the behavior of $H(l)$ for $l \to +\infty$:

$$H(l) = \frac{1}{2} \log(2\pi e l) + O(\frac{1}{l}).$$

APPENDIX IV
PROOF OF CLAIM 1

If the compressor is optimal, z will be equal to the entropy of $\mathsf{Bl}(S)$; an upper bound to that entropy is given by $m \cdot H(\chi)$, where χ is the random variable representing any of the counters and H is the entropy of a random variable.

χ turns out to be a binomial random variable with kn trials and probability $\frac{1}{m}$ for each trial; its mean value is therefore :

$$l = \frac{kn}{m}$$

which is the *load* of the filter.

If $kn \gg l$, then the probability distribution of χ can be approximated by the Poisson distribution with mean l:

$$\Pr[\chi = i] = \binom{kn}{i} \left(\frac{1}{m}\right)^i \left(1 - \frac{1}{m}\right)^{kn-i} \simeq p_i \equiv \frac{l^i}{i!} e^{-l}.$$

So, we can express the entropy of a counter as the entropy $H(l)$ of the Poisson distribution with mean l:

$$H(\chi) \simeq H(l) \equiv -\sum_{i=0}^{+\infty} p_i \log(p_i) \quad (15)$$

APPENDIX V
PROOF OF CLAIM 2

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 element RSA accumulators $\mathsf{Acc}(S_1), \mathsf{Acc}(S_2)$ 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 $x_1, \ldots, x_d \in I \setminus I'$. For any such value x_i, consider any counter j such that $h_l(x_i) = j$, for some $l = 1, \ldots, k$ ($h_1, \ldots, h_k$ are the k hash functions employed by the Bloom filter).

$$\mathsf{Bl}(I)_j = \mathsf{Bl}(I')_j + \mathsf{Bl}(I \setminus I')_j \tag{16}$$

$$\mathsf{Bl}(I')_j < \widehat{\mathsf{Bl}}_j \tag{17}$$

$$\mathsf{Bl}(I' \cup C'_1)_j = (B_1)_j \tag{18}$$

$$\mathsf{Bl}(I' \cup C'_2)_j = (B_2)_j \tag{19}$$

where the first equality (16) is due to the additive definition of the Bloom filter; the second is a direct consequence; (18), (19) are due to (17) and to the fact that we assumed the incorrect query answer is accepted by the Vrfy algorithm.

The fact that $I' \cup C'_1 \subset S_1$ and that $\mathsf{Bl}(I' \cup C'_1)_j = (B_1)_j$ implies that C'_1 must contain the value x_i that was removed from the intersection. For the same reason C'_2 must contains x_i. But this is a contradiction, because one of the test that the verification algorithm performs is that C'_1 and C'_2 are disjoint. So, no attack is possible.

APPENDIX VI
DESCRIPTION OF THE SECURE OPERATION SCHEME ALGORITHMS

```
type BaseCertificate
begin
  sign; {for each standard Bloom filter size m, sign[m] is the signature
         of the source on the pair (Bl, acc) where Bl is the Bloom filter
         of the set with m counters.}
  acc;  {the RSA accumulator of the set}
  witn;    {an witness for the RSA accumulator}
  modulus; {the modulus for the RSA accumulator}
  bf;      {a Bloom filter of this set}
end
```

Fig. 6: An object of this type is a certificate for some base set S, or a component for some `CompoundCertificate`
figure

```
function CS = digest(S)
begin
  CS := new BaseCertificate();
  CS.modulus := the public RSA accumulator modulus of this node;
  CS.witn    := the public RSA accumulator base of this node;
  compute representatives y_1,...,y_c of the elements of S;
  CS.acc := (CS.witn)^(y_1 ····· y_c) mod CS.modulus;
  foreach m in {standard Bloom filter sizes}
  begin
    auxbf := Bl(S, m);
    CS.sign[m] := signature on (CS.acc, auxbf);
  end
end
```

Fig. 7: Returns a `BaseCertificate` for the set `S`
figure

```
type CompoundCertificate
begin
  comp;   {comp[1] is the certificate for S1 and
           comp[2] is the certificate for S2}
  operation; {the operation that generated the set S from S1 and S2.
              Can be union, intersection or difference}
  refsize; {the reference size of the Bloom filter. With this certificate,
            it is possible to reconstruct the Bloom filter of S of size
            refsize, without knowing S}
  check;   {check[1] and check[2] are the two check element sets for the
            operation}
  tag;     {Valid only if operation=union. For each element x in S,
            tag[x] can be either left, right or both. It means, respectively,
            that x is in S1-S2, x is in S2-S1 and x is in both S1 and S2}
end
```

Fig. 8: An instance of this type represents the certificate for some set `S` obtained applying a set operation on two sets `S1` and `S2`.
figure

```
function (S,CS) = secureOperation(S1,S2,CS1,CS2,operation)
begin
 I := S1 ∩ S2;
 S := operation(S1, S2);
 CS := new CompoundCertificate();
 CS.comp[1] := CS1;
 CS.comp[2] := CS2;
 CS.operation := operation;

 if CS1 and CS2 {are both} BaseCertificate
 begin
   CS.refsize:=bestRefSize(S1.size, S2.size);
   CS.comp[1].bf := Bl(S1, CS.refsize);
   CS.comp[1].bf := Bl(S2, CS.refsize);
 end
 else if CS1 is Compound and CS2 is Base
 begin
   CS.refsize := CS1.refsize;
   CS.comp[2].bf := Bl(S2.set, CS.refsize);
 end
 else if CS1 is Base and CS2 is Compound
 begin
   CS.refsize := CS2.refsize;
   CS.comp[1].bf := Bl(S2.set, CS.refsize);
 end
 else
 begin
   CS.refsize := min(CS1.refsize, CS2.refsize);
   (CS.check[1], CS.check[2]) := checkElements(S1-I, S2-I, CS.refsize);
 end

 if operation = union
 foreach x in S
     if x in S1 - S2
     then
       CS.tag[x] := left;
     else if x in S2 - S1
       CS.tag[x] := right;
     else if x in I
       CS.tag[x] := both;

 if operation = difference
 begin
   CS.check[1] := I;
   updateWitnesses(CS.comp[2], S2-CS.check[2]);
 end
 else if operation = intersection
 begin
   updateWitnesses(CS.comp[1], S1-I-CS.check[1]);
   updateWitnesses(CS.comp[2], S2-I-CS.check[2]);
 end
```

Fig. 9: This function performs the specified `operation` on the two sets `S1`, `S2`. Available operations are `union`, `intersection` and `difference`. Arguments `CS1` and `CS2` are the certificates for the sets `S1` and `S2`. The function returns the set `S` that is the result of the operation and the certificate `CS` for it.
figure

```
function m = bestRefSize(n1, n2)
begin
  {choose} m {out of the standard Bloom filter sizes, such that
   } m(H(l_1) + H(l_2) + 2ul_1l_2) {is minimized,
   where} l_1 =n1/m , l_2 =n2/m;
end
```

Fig. 10: Returns the best Bloom filter size to be used when performing an operation between two sets of size `n1` and `n2`, respectively.
figure

```
procedure updateWitnesses(CSi, Si)
begin
 {compute representatives y_1,...,y_c of} Si;
 foreach BC BaseCertificate {in subtree of} CSi
     BC.witn := (BC.witn)^{y_1·····y_c} mod BC.modulus;
end
```

Fig. 11: Modifies all the witnesses in the certificate `CSi`, in order to exclude the elements in set `Si`. That is, for each witness w in `CSi`, if before this call w was the witness for $A \subset B$, for some sets A, B, then, after this call, w is a witness for $(A \backslash \texttt{Si}) \subset B$
figure

```
function (C1, C2) = checkElements(S1, S2, refsize)
begin
  C1 := empty;
  C2 := empty;
  {build hashtables} T1 {of} S1 {and} T2 {of} S2 {with} refsize {buckets};
  for b = 1 to refsize
    if T1[b] <> empty and T2[b] <> empty
    begin
      C1 := C1 ∪ T1[b];
      C2 := C2 ∪ T2[b];
    end
end
```

Fig. 12: Computes the check elements of pair of disjoint sets (`S1`, `S2`), assuming Bloom filters of size `refsize`. `T1[b]` represents the set of elements in bucket `b` of hashtable `T1`
figure

```
function auxbf = recursiveVerify(CS, Set0)
begin
  if CS is BaseCertificate
  begin
    {verify that }CS.sign[CS.bf.size]{ is correct signature for }(CS.acc, CS.bf);
    {compute representatives y_1,...,y_c of the elements in} Set0;
    {verify that} (CS.witn)^({y_1 ····· y_c})mod CS.modulus = CS.acc;
    auxbf := CS.bf;
  end
  else if CS.operation = intersection or CS.operation = union
  begin
    if CS.operation = intersection
    begin
      Set1 := Set0;
      Set2 := Set0;
    end
    else
    begin
      Set1 := {x ∈Set0: CS.tag[x]=left or CS.tag[x]=both };
      Set2 := {x ∈Set0: CS.tag[x]=right or CS.tag[x]=both };
    end
    auxbf1 := recursiveVerify(CS.comp[1], Set1 ∪ CS.check[1]);
    auxbf2 := recursiveVerify(CS.comp[2], Set2 ∪ CS.check[2]);
    {shrink} auxbf1 {and} auxbf2 {to size} CS.refsize;
    auxbf := new {Bloom filter of size} CS.refsize;
    for j=1 to auxbf.size
    begin
      c1[j] = {number of elements of} CS.check[1] {that map to index} j;
      c2[j] = {number of elements of} CS.check[2] {that map to index} j;
      if c1[j] = c2[j] = 0
        auxbf[j] := min(auxbf1[j], auxbf2[j])
      else
      begin
        {verify that} auxbf1[j] - c1[j] = auxbf2[j] - c2[j];
        auxbf[j] := auxbf1[j] - c1[j];
      end
    end
  end
  else if CS.operation = difference
  begin
    auxbf1 := recursiveVerify(CS.comp[1], Set0 ∪ CS.check[1]);
    auxbf2 := recursiveVerify(CS.comp[2], CS.check[1] ∪ CS.check[2]);
    {shrink} auxbf1 {and} auxbf2 {to size} CS.refsize;
    auxbf := new {Bloom filter of size} CS.refsize;
    for j=1 to auxbf.size
    begin
     c1[j] = {number of elements of} CS.check[1] {that map to index} j;
     c2[j] = {number of elements of} CS.check[2] {that map to index} j;
     if c1[j] < auxbf1[j]
     begin
       {verify that} auxbf2[j] = c1[j] + c2[j];
       auxbf[j] := auxbf1[j] - c1[j];
     end
     else
       auxbf[j] := 0;
  end
end
```

Fig. 13: Helper function for `verify`

figure

```
procedure verify(S, CS)
begin
  auxbf := recursiveVerify(CS, S);
  {verify that} auxbf = Bl(S, auxbf.size);
end
```

Fig. 14: Verifies that the set S is authentic, given the certificate CS. Returns in case S is authentic, generates an error otherwise.
figure