A Distributed Implementation of the Certified Information Access Service*

Carlo Blundo[1], Emiliano De Cristofaro[1], Aniello Del Sorbo[1], Clemente Galdi[2], and Giuseppe Persiano[1]

[1] Dipartimento di Informatica ed Applicazioni "R.M. Capocelli"
Università degli Studi di Salerno
Via Ponte Don Melillo - 84084 Fisciano (SA) - Italy
{carblu,emidec,anidel,giuper}@dia.unisa.it
[2] Dipartimento di Scienze Fisiche
Università degli Studi di Napoli "Federico II"
Complesso Univ. Monte S. Angelo - Via Cinthia
80126 Napoli - Italy
galdi@na.infn.it

Abstract. In this paper we consider the problem of securely outsourcing computation on private data. We present a protocol for securely distributing the computation of the data structures used by current implementations of the *Certified Information Access* primitive. To this aim, we introduce the concept of a *Verifiable Deterministic Envelope* - that may be of independent interest - and we provide practical implementations, based on standard cryptographic assumptions.

We also discuss a prototype implementation of our proposal based on the PUB-WCL framework developed at the University of Paderborn for sharing computation across a network.

1 Introduction

The advances in communication technology of the last decades have made it possible the emergence of global-scale computer networks. Networks of such a large scale are currently used to share information for fast dissemination and efficient access. It has been observed that most of the nodes in a global size network are idle most of the time, thus it is very tempting to use the idle cycles of this huge computer to our advantage. We envision a scenario where a user *outsources* its computation to the network. Researchers have considered challenging issues in contexts such as node/service/data discovery, resource scheduling, and load balancing. Moreover, several security issues must be addresses, such as authentication of entities collaborating in the computation, confidentiality of communication, correctness of the computation in presence of malicious adversaries, etc.

* This work has been partially supported by the European Commission through the IST program under Contract FP6-1596 AEOLUS and by the MIUR Project PRIN 2006 *Basi di dati crittografate* (prot. 2006099978).

S. Jajodia, and J. Lopez (Eds.): ESORICS 2008, LNCS 5283, pp. 396–410, 2008.

This paper addresses some of the security issues related on outsourcing computation on private data. Sharing computation across the network has been successfully achieved for several computation-intensive tasks like number-theoretic computation [1] or search for signs of extraterrestrial intelligence [2]. We notice that in both cases the input to the shared computation are public and security issues are trivial or non-existing. In other applications in which the outsourced computation is to be carried out on public data, most of the security issues are directly related to the problem of authentication: is the entity/user/service who required the computation entitled to use the network?

However, in some cases one wants to share computation to be performed on private data. A typical example is given by a user that wants to share the computation associated with a cryptographic algorithm. For instance, authors in [8] showed how a computationally weak device like a mobile device could be helped in computing the encryption of a message by faster devices. In this case the input of the computation, that is the cleartext, is private, otherwise there would be no need for encryption in the first place. Distributing computation on private data is thus more problematic. On one side, the user wants to harness the computational power of network to speed-up his computation; on the other side, he does not want to trust the network with his private data.

In this paper, we show that it is possible to securely share the computation associated with the construction of a data structure to be used for Certified Information Access. To this aim, we introduce a new cryptographic primitive, which we call *Verifiable Deterministic Envelope*, and we show how to use it to securely share the construction of the data structures needed for Certified Information Access. We then give a very efficient implementation of Deterministic Envelopes. We also describe an implementation based on the PUB-WCL [5,4] framework developed at the University of Paderborn.

Certified access to a database. The Certified Information Access (CIA for short) is a cryptographic primitive introduced by [8] that provides certified access to a database. Specifically, the database owner publishes a *snapshot* of its current database, which we refer to as the *Public Information*, on a trusted entity. Once the public information is available, any user may issue queries to the database. The database owner gives the user the query result along with a short proof that the result is consistent with the published snapshot. In other words, the database owner can only reply to users' queries by sending the information actually contained in the database while a user will be able to identify wrong and/or maliciously constructed answers.

We notice that this problem has several applications in contexts in which the database owner as incentives to provide specific informations to the users. For example consider a website that provides information on the "closest" shop to a given position to its users. Furthermore, assume that the same website publishes commercials for some shops that pay some fee for such a service. In this case, the website administrator might be willing to pre-process replies in a way to favor the companies that pay for commercials w.r.t. the ones that do not pay any fee.

One trivial way of implementing CIA is to publish the whole content of the database on a trusted server. A user can thus compare the received reply with the one contained in the public copy of the database (or actually directly query the trusted server). This solution is, of course, neither secure nor efficient. Since the database has to publish its whole contents, the confidentiality of the information therein contained is compromised. Furthermore, since all the elements in the database need to be transmitted and stored, the communication and space complexity of this solution is linear in the size of the database.

For these reasons the public information should satisfy the following properties:

- *Compactness:* If s is the number of entries in the database, then the size of the public information should be at most $O(polylog(s))$.
- *Confidentiality:* The public information should not reveal anything about the actual content of the database.
- *Correctness:* A correct answer to a query should be consistent with the public information with probability one.
- *Soundness:* Any wrong answer to a query will be detected with high probability.

Currently, a way of implementing CIA primitives is by means of a new cryptographic primitive, namely *mercurial commitments* introduced and studied in [10,7,6,9]. Unfortunately, the implementation of such primitive are computationally intensive. On one hand, the generation of the public information is time consuming also on current servers. On the other hand, although the verification procedure can be easily executed on a PC in few seconds, it still requires a not neglibigle time on tiny mobile devices.

In [8], the authors provide a distributed implementation of a CIA service. In such implementation, the database owner builds a tree representing the database. Each node in the database is associated to the computation of modular exponentiations. The system presented in the above paper, by using "pre-computation" peers, securely distributes, for each node in the tree, the computation of the modular exponentiations. Each pre-computation peer receives a base and a modulus and she is required to randomly select a number of exponents and to compute the corresponding modular exponentiation. The database owner is simply required to locally combine partial results obtained by the peers.

Our Contribution. In this paper we present an efficient distributed implementation of a CIA service. Our implementation allows the database owner to completely outsource the tree computation. Clearly, the information in the database need to be hidden before they are transferred. Unfortunately, as it will be clear in the next sections, it is not possible to use "simple" (neither randomized nor deterministic) encryption schemes. To this aim, we first present a new primitive, which we call Verifiable Deterministic Envelope (or VDE for short), that allows the "encryption" of all the elements in the database. The "encrypted" version of the database is then transferred to a web-cluster that computes the tree associated to the database and sends it back to the database owner.

In this scenario, the database owner still needs to locally compute an "encryption" of each element in the databases.

As in [8], we describe a solution for *static* databases, i.e., databases in which the content does not change. This is however without loss of generality as our main target will be slowly-changing databases for which every time the database is updated a new snapshot is published. For rapidly changing databases, one could use the construction of [9] which is however not very practical. Indeed, the latter solution requires that, for each database update, the database owner has to publish some information whose size is poly-logarithmic in the size of the key space.

Notice that, since our solution works only for static databases, it is important to speed up the construction of the tree of commitments since CIA for slowly-changing databases can be implemented by reconstructing from scratch the whole tree.

We assume that there exists a trusted entity that does not collude with the entity holding the database. The trusted party is only required to generate some of the public parameters for the scheme and to store the database's snapshot. Furthermore, we assume the possibility of accessing a cluster of untrusted machines whose role is to construct the tree of commitments.

This paper is organized as follows: In Section 2, we describe the cryptographic primitives used to implement our prototype. In Section 3 we show how to modify the initial database in order to securely outsource computation. In Section 4 we describe a new primitive, which we call the Verifiable Deterministic Envelope, used to secure the outsourcing of the computation. In Section 5 we report the design principles for a distributed architecture implementing a CIA service. Due to space limitations, an extended version of this paper can be found at [3].

2 Cryptographic Background

In this section we describe the cryptographic primitives we will use in this work.
One Way Trapdoor Permutations. A trapdoor permutation family Π over D consists of the following triple of algorithms: ($\texttt{PermGen}$, $\texttt{Permute}$, $\texttt{Invert}$). The randomized algorithm $\texttt{PermGen}$, on input a security parameter, outputs the description f of a permutation along with the corresponding trapdoor t_f. The algorithm $\texttt{Permute}$, given the permutation description f and a value $x \in D$, outputs $y \in D$, the image of x under the permutation f. The inversion algorithm $\texttt{Invert}$, given the permutation description f, the trapdoor t_f, and a value $y \in D$, outputs the pre-image of y under the permutation f. When it engenders no ambiguity, we will write $f(x)$ instead of $\texttt{Permute}(f, x)$ and $f^{-1}(y)$ instead of $\texttt{Invert}(f, t_f, y)$.

We require that $\texttt{Permute}(f, \cdot)$ be a permutation of D for all possible (f, t_f) output by $\texttt{PermGen}$, and that $\texttt{Invert}(f, t_f, \texttt{Permute}(f, x)) = x$ hold for all (f, t_f) output by $\texttt{PermGen}$ and for all $x \in D$. Let $(f, t_f) \leftarrow \texttt{PermGen}(1^k)$ and $y = f(x)$. A trapdoor permutation family Π is said to be one-way if, given y, and f, no polynomial time algorithm can compute x with non-negligible probability.

Deterministic Signatures. A deterministic signature scheme is a triple of algorithms $S = (\texttt{KeyGen}, \texttt{Sign}, \texttt{Ver})$. The algorithm $\texttt{KeyGen}$ is a randomized algorithm that, on input a security parameter, outputs a pair (sk, vk), consisting of

a signature key sk and a verification key vk. The algorithm `Sign` is a deterministic algorithm that takes as input a message x and the signing key sk and outputs a signature σ for x. The verification algorithm `Ver` is a (possibly randomized) algorithm that takes as input a message, x, a signature, σ and a verification key vk and outputs 1 if and only if $\sigma = \texttt{Sign}(x, sk)$.

A signature scheme is secure against *selective forgery* attacks if there exist no (probabilistic) polynomial time algorithm that, on input a message x and the verification key vk, outputs a valid signature σ for x with non-negligible probability.

Mercurial Commitments. In any commitment scheme, a sender and a receiver participate in a `Commit` protocol where the sender commits to some secret information m while the receiver learns nothing[1] on the value of m. Later the sender can "open" the commitment by revealing the secret information and the receiver can reject such value if it is not consistent with the information received during the `Commit`. A commitment scheme meets the *binding* property if it guarantees the receiver that the sender cannot reveal a value $m' \neq m$ that is consistent with the information exchanged during the `Commit` phase.

A Mercurial Commitment scheme, studied in [10,7,6,9], allows two possible ways of creating a commitment. A *hard commitment* h to a message m, corresponding to a "classical" commitment, meets both the *hiding* and *binding* properties. In other words, the hard commitment h for a message m does not reveal any information on m. At the same time, h can be only "opened" to the value m, i.e., the sender cannot reveal a value $m' \neq m$. A *soft commitment* does not take any message as input and cannot be "opened" as hard commitments. On the other hand, soft commitments are not binding, that is, there exists a special *tease* operation that allow to "open" a soft commitment to any message m. A hard commitment can be *opened* and *teased* only to the original message m.

A feature of Mercurial commitments schemes is that hard and soft commitments are *indistinguishable*. Thus, given a mercurial commitment, it is not possible to determine *a priori* whether it is a hard or a soft one. More formally, a Mercurial Commitment scheme consists of the following 6-tuple

$$MC = (\texttt{Setup}, \texttt{Commit}, \texttt{VerifyOpen}, \texttt{SoftCommit}, \texttt{Tease}, \texttt{VerifyTease})$$

of possibly randomized algorithms. In this section we show how we implement Mercurial Commitment schemes. Our implementation is based on the hardness of the discrete logarithm in cyclic groups and is based on [7].

The `Setup` procedure consists in randomly picking a random prime p and two generators g, h of the cyclic group $Z_p^\star$. All operations are to be considered in the group $Z_p^\star$ unless otherwise specified.

[1] Commitment schemes may be computationally or unconditionally hiding. In the first case, the receiver is restricted to some PPT and, with high probability, she cannot obtain information on the committed value. In the latter case, the receiver has unbounded computational power and, with probability 1, she cannot gain any information on the value of m in the information-theoretic sense.

The `Commit` procedure takes as input the string m and public parameters (p, g, h) and computes `com` and `dec` as follows: randomly pick $r_0, r_1 \in Z_p^\star$ and set $\texttt{com} = (g^m \cdot (h^{r_1})^{r_0}, h^{r_1})$ and $\texttt{dec} = (r_0, r_1)$.

The `VerifyOpen` procedure takes as input the public parameters (p, g, h), a message m, a commitment $\texttt{com} = (C_0, C_1)$ and decommitment key $\texttt{dec} = (r_0, r_1)$ and consists in checking whether $C_0 = g^m \cdot C_1^{r_0}$ and $C_1 = h^{r_1}$.

The `SoftCommit` procedure takes as input the public parameters (p, g, h) and computes `Scom` and `Sdec` as follows: randomly pick $r_0, r_1 \in Z_p^\star$ and set $\texttt{Scom} = (g^{r_0}, g^{r_1})$ and $\texttt{Sdec} = (r_0, r_1)$.

As stated above, the `Tease` procedure can be applied both on hard and soft commitments. In case of hard commitment, the `Tease` procedure takes as input a hard commitment $\texttt{com} = (g^m \cdot (h^{r_1})^{r_0}, h^{r_1})$, the decommitment key $\texttt{dec} = (r_0, r_1)$, and the string m and simply returns $\tau = r_0$. In case of soft commitment, the procedure takes as input a soft commitment $\texttt{Scom} = (g^{r_0}, g^{r_1})$, the teasing key $\texttt{Sdec} = (r_0, r_1)$, and the string m and returns $\tau = (r_0 - m)/r_1 \pmod{p-1}$.

The `VerifyTease` procedure takes as input public parameters (p, g, h) and teasing τ of commitment (C_0, C_1) to string m and consists in checking whether $C_0 = g^m \cdot C_1^\tau$. Correctness and security of this scheme have been shown in [7].

Certified Information Access. In the context of secure databases, an implementation of a certified information access has to provide the users with a database service in which each answer to a query consists of the actual query results and a proof that such information is indeed the actual content of the database. The verification of the proof can be accomplished by using some public information that the database provided before the query was issued. Such public information should not reveal anything about the actual content of the database. In a CIA system we identify three parties, the CertifiedDBOwner the User and the PubInfoStorage.

In a setup phase, the PubInfoStorage generates the public parameters that will be used for the CIA service. The PubInfoStorage is assumed not to collude with the CertifiedDBOwner.

The CertifiedDBOwner, based on public parameters and the content of the database, produces the public information that is then sent to the PubInfoStorage. Whenever a User makes a query to the CertifiedDBOwner, he obtains an object that contains the answer to the query and some information that can be used, along with the information held by the PubInfoStorage, to prove that the answer is indeed correct and that the CertifiedDBOwner has not cheated.

Certified Information Access via Mercurial Commitments. In the following, we describe how to implement the CIA functionality based on Mercurial Commitments. This description resembles the one in [7]. We consider a simple database D associating to a key x a value $D(x) = v$. Let us assume that all keys have the same length ℓ, i.e., $x \in \{0,1\}^\ell$. A reasonable choice is $\ell = 60$ since on one hand it allows to have a large key space and, at the same time, it is possible to use hash functions for reducing the key size to this small value. The database D can thus be represented by a height-ℓ binary tree where leaf identified by the binary

representation of x contains the value $v = D(x)$. If no value is associated by the database to key x, the leaf numbered x contains the special value $\perp$.

A first solution to the CIA problem could be to constructs a binary tree as follows: the leaves of the tree contain "classical" commitments of elements of the database. Each internal node of the tree contains the "classical" commitment of the hash of the concatenation of the contents of its two children. The "classical" commitment contained in the root of such a tree constitutes the public information that is sent to the PubInfoStorage.

To answer to a query about x, the CertifiedDBOwner simply de-commits the corresponding leaf and provides the authenticating path (along with all the decommitments) to the root. The problem with this approach is that it requires time exponential in the height of the tree: if we choose $\ell = 60$, then $2^{61} - 1$ commitments need to be computed.

This is where Mercurial Commitment helps. Observe that the exponential-size tree might have large empty subtrees (that is, subtrees where each leaf is a commitment to $\perp$). Hence, instead of actually computing such a subtree ahead of time, the CertifiedDBOwner forms the root of this subtree as a soft commitment and does not compute anything for the rest of the tree. Thus, the size of the tree is reduced from $2^{\ell+1} - 1$ to at most $2\ell|D|$, where $|D|$ represents the number of elements in the database. Answering to a query about x such that $D(x) \neq \perp$ is still done in the same way, i.e., the CertifiedDBOwner de-commits the leaf corresponding to x along with all the commitments on the path from the root of the tree to the leaf. If instead $D(x) = \perp$, the CertifiedDBOwner teases the path from the root to x. More precisely, the path from the root to x will consists of hard commitments until the root R of the empty subtree containing x is encountered. All hard commitments from the root of the tree to R are teased to their real values (recall that hard commitments can be teased only to their real value). Then, the CertifiedDBOwner generates a path of soft commitments from R to (the leaf with number) x ending with the commitment of $\perp$. Each soft commitment corresponding to a node along the path is teased to the soft commitments corresponding to its two children. The User simply needs to verify that each teasing has been correctly computed. We stress that for positive queries (that is, queries for x such that $D(x) \neq \perp$) the User expects to see *opening* of hard commitments whereas for negative queries (that is, queries for x such that $D(x) = \perp$) the User expects to see *teasing* of commitments; some of them will be hard commitments and some will be soft commitments but the user cannot say which ones are which.

3 Securely Outsourcing CIA

As stated in the previous section, a CIA service can be implemented by using mercurial commitments. An implicit assumption is that the CertifiedDBOwner uses a *deterministic* algorithm to associate the elements in the database to the leaves of the tree, e.g., the value $v = D(x)$ is associated to the leaf identified by the binary representation of the key x. Such a deterministic approach allows the

user to *verify* that the received proof corresponds to *a specific path* that, in turn, corresponds to the queried key. We remark that the above argument does not rule out the possibility of (pseudo-)randomly assigning keys to tree leaves. However, if the CertifiedDBOwner uses a randomized strategy for such assignment, for each key in the database, there should exists a "certified" way of binding the key along with the randomness used to create the path in the tree.

Thus, a deterministic placement of database elements to tree leaves guarantees the security of the CIA service. On the other hand, if the CertifiedDBOwner is willing to outsource the tree construction, the problem of confidentiality of information contained in the database has to be taken into account. Notice, however, that there exist two conflicting requirements. On one hand, the CertifiedDBOwner need to hide the information contained in the database from the entity that constructs the tree. Specifically, it should not be possible to check whether there exists in the database a value associated to a specific key. On the other hand, the User should be able to verify that the answer received for a given query actually corresponds to the submitted key.

A first solution could be to (deterministically) encrypt the keys of the elements in the database (along with the values associated to them). If the CertifiedDBOwner uses a symmetric key encryption scheme, she can securely hide all the information from the computing entity. Unfortunately, after the tree has been constructed, she need to publish the key used to encrypt the keys in order to allow the users to properly verify the answers.

A second possible approach could be to encrypt each key in the database using a public key encryption scheme. In this case, the leaf associated to the key is identified by the binary representation of the encryption of the key itself. This approach allows the User to compute by herself the "permuted" value of the key. Unfortunately the entity that computes the tree can execute the same operation herself by obtaining information on the existence of some specific key in the database.

Another possible solution could be to place each element in the database in the leaf identified by the (binary representation of the deterministically computed) signature of the key. We remark that also this solution does not prevent the computing entity to check whether or not some key belongs to the database by simply "verifying" the existance of a valid signature for the considered key.

For the above reasons, we need a way of deterministically permuting the keys in the database in a way that the computing entity will not be able (a) to compute the key given its permuted value and (b) to compute the permuted value of any key of its choice. Furthermore, the user, by interacting with the CertifiedDBOwner, should be able (a) to compute the permuted value associated to a given key and (b) to verify that the computed value is the one used by the CertifiedDBOwner during the tree construction phase.

In order to hide the information contained in the database before outsourcing the tree computation we need a deterministic function, we will call it `EnvelopeGen`, which should guarantee the following properties: the keys contained in the database are deterministically permuted and the entity to which the tree construction

will be outsourced will have no information on the actual keys contained in the original database. At the same time, the user should be able to verify the answers generated by the CertifiedDBOwner. Furthermore, since the CertifiedDBOwner transfers the hard commitment of the values contained in the database, the confidentiality of the content of the database is guaranteed. Once we have such a function, we build a new database as follows: to each pair $(key, value)$ we associate the pair $(\texttt{EnvelopeGen}(key), \texttt{Commit}(value))$.

As we will see in the next section, such properties can be achieved by using a new primitive we introduce. This primitive will be used to permute the elements in the database. Given such a permutation, the user can, by interacting with the CertifiedDBOwner, compute the value associated to each key. On the other hand, the computing entity, given the "permuted" database, cannot tell whether or not a key is therein contained without interacting with the CertifiedDBOwner.

4 Verifiable Deterministic Envelopes (VDE)

In this section we introduce a new primitive, the *Verifiable Deterministic Envelope*, or VDE for short. We will use such a primitive to hide the information contained in the database before outsourcing the tree computation our our distributed implementation of CIA.

4.1 VDE: Definition

Definition 1. *A* Verifiable Deterministic Envelope *(VDE for short) is a triple (*`EnvKeyGen`*,* `EnvelopeGen`*,*`EnvelopeOpen`*) of algorithms, involving a sender S and a receiver R, described as follows:*

- *A randomized key generation algorithm* `EnvKeyGen` *executed by the sender. The algorithm* `EnvKeyGen`*, on input a security parameter, k, generates a pair (pk, sk). The value pk is made public while sk is kept secret by the sender.*
- *A deterministic envelope generation algorithm* `EnvelopeGen` *executed by the sender. The algorithm* `EnvelopeGen`*, on input a string x, and the pair (pk, sk) computes a VDE for x. In order to simplify the notation, we will denote by* $\texttt{EnvelopeGen}(x)$ *the VDE computed as* $\texttt{EnvelopeGen}(x, (pk, sk))$.
- *An interactive opening protocol* `EnvelopeOpen` *executed by both the sender and the receiver. The protocol is started by the receiver who provides a string x. At the end of the protocol, the receiver is able to compute the (correct) value for* $\texttt{EnvelopeGen}(x)$.

The following is a security definition for VDE:

Definition 2. *A VDE scheme* $(\texttt{EnvKeyGen}, \texttt{EnvelopeGen}, \texttt{EnvelopeOpen})$ *is OW-secure if the following holds:*

a. For any x, given pk and sk, the sender can compute $\texttt{EnvelopeGen}(x)$ *in polynomial time.*

b. For any x, given pk, the receiver R by executing the opening protocol `EnvelopeOpen` *can compute in polynomial time the (correct) value* $y = \texttt{EnvelopeGen}(x)$.
c. For any x, given pk, there exists no polynomial time algorithm that computes $y = \texttt{EnvelopeGen}(x)$ *with non-negligible probability.*
d. For any VDE $y = \texttt{EnvelopeGen}(x)$, *given pk, there exists no polynomial time algorithm that computes x with non-negligible probability.*

4.2 A General Construction

In the following we describe a general construction for a VDE scheme based on one-way trapdoor permutations.

Definition 3 (A VDE Scheme). *Let Π be a one-way trapdoor permutation family. The proposed VDE consists of the following algorithms:*

- `EnvKeyGen`$(\cdot)$*: The algorithm takes as input 1^k, where k is a security parameter and outputs the pair $(pk, sk) = ((\ell, (f, g)), (t_f, t_g))$, where ℓ is an integer, $(f, t_f) \leftarrow$* `PermGen`(1^k) *and $(g, t_g) \leftarrow$* `PermGen`(1^k)*, are one-way trapdoor permutations over $\{0,1\}^\ell$. The sender publishes the pair (f, g) while (t_f, t_g) is kept secret.*
- `EnvelopeGen`$(\cdot,\cdot)$*: The algorithm is executed by the sender. It takes as input a value $x \in \{0,1\}^\ell$, and the pair (pk, sk) and outputs $e =$* `EnvelopeGen`$(x, (pk, sk)) = f(g^{-1}(x))$*. The value e is sent to the receiver.*
- `EnvelopeOpen`*: Is an interactive protocol executed by the sender and the receiver. Let $x \in \{0,1\}^\ell$. The opening protocol works as follows:*
  - *The receiver R sends x to S.*
  - *The sender S computes $w = g^{-1}(x)$ and sends w to R.*
  - *The receiver checks whether $g(w) = x$ and computes* `EnvelopeGen`$(x) = f(w)$.

It is possible to prove the following:

Theorem 1. *If f and g are one-way trapdoor permutations, then the Verifiable Deterministic Envelope scheme described in Definition 3 is OW-secure.*

4.3 VDE: A Practical Instantiation

In this section we describe a practical instantiation of the VDE primitives we have introduced in the previous section. In our system we use the following implementation of the VDE:

- The function f is implemented as a deterministic RSA encryption.
- The function g is implemented as a deterministic RSA signature.

The different phases of the VDE are implemented as follows:

- `EnvKeyGen`$(\cdot)$: The algorithm `EnvKeyGen` outputs the pair $((n_E, e_E, d_E), (n_S, e_S, d_S))$, where
  - $n_S < n_E$;
  - (n_E, e_E, d_E) are the parameters for the RSA encryption scheme;

  - (n_S, e_S, d_S) are the parameters for the RSA signature scheme;
  - The parameters $pk = ((n_E, e_E), (n_S, e_S))$ are published while $sk = (d_E, d_S)$ is kept secret.
- `EnvelopeGen`$(x, (pk, sk))$: Given a value x, the sender computes $e = E(\mathtt{Sig}(x, d_S), e_E) = (x^{d_S} \bmod n_S)^{e_E} \bmod n_E$. The value e is sent to the receiver.
- `EnvelopeOpen`: Let x be a key. The opening protocol works as follows:

  - The Receiver R sends x to the S.
  - The Sender S computes $b = \mathtt{Sig}(x, d_s) = x^{d_S} \bmod n_S$ and sends b to R.
  - The receiver checks whether $b^{e_S} = x \bmod n_S$ and computes `EnvelopeGen`$(x, (pk, sk)) = b^{e_E} \bmod n_E$.

5 The Architectural Design Principles

In this section we describe the architectural design principles of the implemented prototype and we report the results of some experiments run in order to evaluate the performances and the scalability of our distributed certified information access service.

We implemented our prototype by using the PUB-WCL library [5,4] which has been designed to execute massively parallel algorithms in the BSP model on PCs distributed over the Internet, exploiting unused computation donated by PC owners. This goal is achieved efficiently by supporting thread migration in Java and providing a load-balanced programming interface.

The BSP (Bulk Synchronous Parallel) model provides a parallel computing scheme consisting of a set of processors with local memory, an interconnection mechanism allowing point-to-point communication, and a mechanism for barrier-style synchronizations. A BSP program consists of a set of processes and a sequence of supersteps, i.e. time intervals bounded by the barrier synchronization.

Architectural design. In this section we briefly describe the architectural design for a system implementing the primitives described above. We identify four different entities, CertifiedDBOwner, User, PubInfoStorage, and Cluster.

As described in Section 2, a CIA service consists essentially of three phases: an initialization phase in which the parameters used for the scheme are generated; a tree construction phase in which the tree of commitments is constructed; a query phase, in which the User queries the database. In our setting, the applications cooperate as follow:

- Generation of the parameters: The parameters used for the implementations are generated as follows:

  - At startup, the PubInfoStorage generates the public parameters that will be used for the Mercurial Commitment implementations. Since public parameters are used both for generating the public information and verifying all the answers to queries, such generation is carried out once, and the parameters are stored in a file.

- The CertifiedDBOwner generates the pair $((n_E, e_E, d_E), (n_S, e_S, d_S))$ that will be used for the VDE computations. The CertifiedDBOwner sends to the PubInfoStorage $pk = ((n_E, e_E), (n_S, e_S))$ while it keeps secret $sk = (d_E, d_S)$.

– Construction of the Tree of Commitments: The tree of commitments is constructed in three main steps:

  - Local VDEs Computation: For each element $(key, value)$ in the database the CertifiedDBOwner locally computes $\texttt{EnvelopeGen}(key, (pk, sk))$. Once the VDEs for all the keys have been computed, the CertifiedDBOwner holds a "new" database consisting of the pairs $(\texttt{EnvelopeGen}(key, (pk, sk)), value)$.
  - Local Computation of Tree Leaves: For each element in the database, the CertifiedDBOwner computes the hard commitment to the string $value$. After this phase, the CertifiedDBOwner holds a new database consisting the pairs $(\texttt{EnvelopeGen}(key, (pk, sk)), \texttt{com}(value))$.
  - Outsourcing the Tree Construction: The CertifiedDBOwner sends the transformed database to the Cluster by using the PUBWCL library. The Cluster computes the public information as described in the previous paragraph that is sent back to the CertifiedDBOwner who forwards it to PubInfoStorage.

– User Queries: Once the public information has been published by the PubInfoStorage, the User can query the database as follows:

  - Let key be a key. The User engages in a opening protocol with the CertifiedDBOwner.
  - The User computes $y = \texttt{EnvelopeGen}(key, (pk, sk))$.
  - The CertifiedDBOwner sends to the User the answer to the query along with the open/tease of all the nodes on the path identified by the binary representation of y.
  - The User verified the proof received by the CertifiedDBOwner and accepts it if it is consistent with the public information held by the PubInfoStorage.

When the User queries the CertifiedDBOwner, he obtains a reply that is verified against the public information held by the PubInfoStorage. We assume point-to-point secure communication among CertifiedDBOwner, User and PubInfoStorage. Such an assumption can be easily implemented, e.g., by using Diffie-Hellman key exchange for establishing once a common key to used for all the communications.

A BSP-based tree construction implementation. In the following we present a distributed algorithm to construct the Mercurial commitment tree. Such a distributed construction will be realized by the processors of a BSP-cluster. The workload assigned to each processor of the cluster will be about the same. This

is due both for efficiency reasons and because of the specific characteristics of the BSP model.

For each element $(\texttt{EnvelopeGen}(key), \texttt{Commit}(value))$ of the database, the algorithm run by the CertifiedDBOwner computes the hash value $H(\texttt{EnvelopeGen}(key))$, where $H : \{0,1\}^* \rightarrow \{0,1\}^{60}$, and stores its value in an ordered list. Then, such a list is partitioned into 2^p ordered sub-list, where p depends on the number of processors in the BSP cluster. The i-th sub-list will includes all the elements in the subtree rooted at the node identified by the binary representation of i. On the BSP cluster, a task spanning on 2^p processors is created. Finally, the 2^p ordered sub-lists are sent to the BSP cluster. In the cluster, Processor *0* will play a *special* role collecting the roots of all subtree computed by all processors and building a Mercurial commitment tree having such roots as leaves. The CertifiedDBOwner will be ready to answer to the client's queries once that it has received all the subtrees computed by the processors in the BSP cluster (including the "top" subtree computed by Processor *0*).

In order to keep the memory footprint low on each processor, we fix the maximum height h allowed for the tree reconstructed by the processor. The procedure TreeHeight(L) returns the height k of Mercurial commitment tree containing the leaves in L. If k is less than h, then the processor will compute the subtree from the ordered block of leaf nodes assigned to it (i.e. the leaves in L). Once that the computation of the subtree is completed, the processor sends the whole subtree to the CertifiedDBOwner while its root is sent to Processor 0. On the other hand, if k is bigger than h, then the processor will partition the list of received values as done by the CertifiedDBOwner. At this point, the processor, for each sub-list in the partition, will compute a Mercurial commitment tree that will be sent to the CertifiedDBOwner. The processor will cache the roots of the computed subtrees. After the computation of all subtrees has been completed, the processor will build a Mercurial commitment tree having the cached roots as leaves. Such a subtree is sent to the CertifiedDBOwner while its root will be sent to Processor *0*. The CertifiedDBOwner caches each received subtree.

Processor 0, besides computing the Mercurial commitment tree from the values received by the CertifiedDBOwner, will retrieve and sort all the cached roots of the other subtrees. From this ordered list L Processor 0 will build a Mercurial commitment tree having the values in L as its leaves.

Experimental Results. In this section, we report the results of some experiments run to evaluate the performances and the real applicability of our distributed certified information access service. Our experiments have been carried out on two different types of machines. The first, which we call "local PCs", and a set which we call the "cluster PCs", implementing the web cluster consists of 39 PCs geographically distributed in 8 European Countries. Among these computers we identify a *central core* consisting of 16 PCs located in the University of Paderborn. Due to the scheduling policy, if the number of tasks submitted to the cluster is low and if the load of the machines in the central core is low, all tasks are assigned to such PCs. We notice that, the PUBWCL library allows

the possibility to submit tasks to the cluster by assigning a *virtual* node to each task. Depending on the current load of the cluster, multiple virtual nodes may be assigned to the same machine. Furthermore, since it is not possible to gain complete control over the machines in the cluster, if the number of submitted tasks is high, very likely multiple tasks will be allocated to the same machine inducing, as a side effect of the underlying BSP model, a slowdown for the whole application.

For this reason we have first evaluated the time needed to outsource the computation of the tree of commitments on 4 and 8 nodes. Our experiments show that, although the CertifiedDBOwner needs to locally compute the VDE and the hard commitment for each element in the database, most of the computation time is required by the cluster for distributively construct the tree of commitments. Furthermore, as the number of nodes increases, the system performance becomes fluctuating because of the cluster load.

We have then compared the time required by a distributed solution against the one needed by a centralized algorithm executed on a local PC. Our experiments show a dependance on the cluster load on a solution including a bigger number of nodes. We can derive that, if we consider only the time needed to distributively compute the tree of commitments, the speed-up achieved is linear in the number of nodes used for the computation. On the other hand, in order to properly compare a centralized solution against the distributed one, we need to take into account the total time from the moment in which the CertifiedDBOwner starts the process of (locally) computing the VDE for the elements in the database to the moment in which such computation terminates, i.e., we need to include the local VDE computation and the data transfer. In this case we notice that, as the number of nodes increases, the distributed computation time decreases and, thus, the impact on the speed-up ration of local computation and data transfer increases. Nevertheless, the speed-up achieved by the distributed solution is still linear in the number of nodes used for outsourcing the computation.

6 Conclusions

In this paper we have presented a distributed architecture for a Certified Information Access system. We have shown that it is possible to securely outsource the load for the construction of the tree of commitments to a cluster of untrusted PCs. To this aim, we have introduced a new cryptographic primitive, the Verifiable Deterministic Envelope, that might be of independent interest. Our experiments have shown that, if it is possible to gain complete control over all the machines in the cluster, then it is possible to achieve a linear speed-up w.r.t. a centralized implementation. We have used the PUBWCL library as an "off-the-shelf" technology that allows the parallelization of computation. We remark that our solution for secure outsourcing does not depend on the specific technology.

Acknowledgements

The authors want to thank Joachim Gehweiler for helpful discussions on PUBWCL.

References

1. Great Internet Mersenne Prime Search (GIMPS) (Last Visit, July 2007), http://www.mersenne.org
2. Search for extraterrestril intelligence (SETI@home) (Last Visit, July 2007), http://setiathome.berkley.edu
3. Blundo, C., Cristofaro, E.D., Sorbo, A.D., Galdi, C., Persiano, G.: A distributed implementation of the certified information access service, http://people.na.infn.it/~galdi/PAPERS/esorics08.pdf
4. Bonorden, O., Gehweiler, J., auf der Heide, F.M.: Load balancing strategies in a web computing environment. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Wasniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 839–846. Springer, Heidelberg (2006)
5. Bonorden, O., Gehweiler, J., auf der Heide, F.M.: A Web Computing Environment for Parallel Algorithms in Java. In: Proceeedings of International Conference on Parallel Processing and Applied Mathematics (PPAM), pp. 801–808 (2005)
6. Catalano, D., Dodis, Y., Visconti, I.: Mercurial commitments: Minimal assumptions and efficient constructions. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 120–144. Springer, Heidelberg (2006)
7. Chase, M., Healy, A., Lysyanskaya, A., Malkin, T., Reyzin, L.: Mercurial commitments with applications to zero-knowledge sets. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 422–439. Springer, Heidelberg (2005)
8. Sorbo, A.D., Galdi, C., Persiano, G.: Distributed certified information access for mobile devices. In: Sauveron, D., Markantonakis, C., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 67–79. Springer, Heidelberg (2007)
9. Liskov, M.: Updatable zero-knowledge databases. In: Roy, B.K. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 174–198. Springer, Heidelberg (2005)
10. Micali, S., Rabin, M.O., Kilian, J.: Zero-knowledge sets. In: 44th Symposium on Foundations of Computer Science (FOCS 2003), pp. 80–91. IEEE Computer Society, Los Alamitos (2003)