Publicly Verifiable Secret Sharing for Cloud-based Key Management

Roy D'Souza[1], David Jao[2,⋆], Ilya Mironov[1,3], and Omkant Pandey[1,⋆⋆]

[1] Microsoft Corporation, Redmond WA, USA
{royd,omkantp}@microsoft.com
[2] University of Waterloo, Waterloo ON, Canada
djao@math.uwaterloo.ca
[3] Microsoft Research Silicon Valley Center, Mountain View CA, USA
mironov@microsoft.com

Abstract. Running the key-management service of cryptographic systems in the cloud is an attractive cost saving proposition. Supporting key-recovery is an essential component of every key-management service. We observe that to verifiably support key-recovery in a public cloud, it is essential to use publicly verifiable secret-sharing (PVSS) schemes. In addition, a *holistic* approach to security must be taken by requiring that running the key-management service in the (untrusted) cloud does not violate the security of the cryptographic system at hand.

This paper takes such a holistic approach for the case of public-key encryption which is one of the most basic cryptographic tasks. The approach boils down to formalizing the security of public-key encryption *in the presence of* PVSS. We present such a formalization and observe that the PVSS scheme of Stadler [29] can be shown to satisfy our definition, albeit in the Random Oracle Model.

We construct a new scheme based on pairings which is much more efficient than Stadler's scheme. Our scheme is noninteractive and can support any monotone access structure. In addition, it is proven secure in the *standard* model under the Bilinear Diffie-Hellman (BDH) assumption. Interestingly, our PVSS scheme is actually the *first* non-interactive scheme proven secure in the *standard* model; all previous non-interactive PVSS schemes assume the existence of a Random Oracle. Our scheme is simple and efficient; an implementation of our scheme demonstrates that our scheme compares well with the current fastest known PVSS schemes.

1 Introduction

Today, there is a huge emphasis on cloud computing. The "cloud" can be thought of as an infrastructure which is available to everyone at all times and provides reliable data storage and computational power at low cost. More and more applications are now moved from private machines to run in clouds to reduce capital

⋆ Partially supported by NSERC

⋆⋆ Also affiliated with Microsoft Research India, Bangalore, India

and operational expense. Leveraging the cloud infrastructure is an increasingly popular value proposition for IT companies [10, 17]. The cloud can either be operated *privately* by a trusted party, or *publicly* by an untrusted third party. In this paper, we will be concerned only with public clouds which cannot be trusted for cryptographic purposes.

Key-management is an essential component of systems that deploy cryptographic techniques. Since availability and reliability are two crucial requirements of a good key-management service, running it in the cloud is a natural proposition for cost reduction [18]. In this work, we shall focus on one of the most basic cryptographic tasks: public-key encryption. A similar formal treatment can be easily provided for other cryptographic tasks as well such as signature schemes.

Suppose that (PK, SK) is the public and secret key-pair of an employee U of a business B. The key-management service, run by the business (administrator) B, has two fundamental tasks: securely store SK from where *only* U can access SK (for example, by providing his passphrase); and support *key-recovery*. That is, if U loses SK (or his passphrase), it should be possible for some authorized party (such as U's manager) to recover SK for U. Typically, who can recover SK for U is determined by an *access policy* $\mathbb{A}$ which represents authorized sets of parties which must collude to recover SK for U. This set may or may not include the system administrator B. At the time of registration when (PK, SK) are generated for U, strict guidelines must be followed to *verifiably* ensure that SK can indeed be recovered as defined by the policy $\mathbb{A}$. Failure to recover SK in legitimate circumstances can result in data loss which can lead to severe financial damage and perhaps legal consequences.

We would like to bring attention to a subtle yet crucial point here. The verification steps to ensure that key-recovery can indeed be performed when needed, is usually performed by an automatized procedure controlled by the system administrator B. Such a process usually needs one-time access to the generated key SK to successfully complete the verification task. As a result, in principle, administrator B might be able to look at SK even if ideally he should not be allowed to do so. The usual "fix" around this problem is that B is bound by a legal contract trusted entity. Therefore, it is not considered the part of the adversary trying to break the security of the encryption scheme. That is, in the mathematical formalizations of security of PKE (such as the IND-CPA game), B is not modeled as a separate entity.

However, when the key-management service is moved to the cloud, this becomes a severe problem. The cloud, unlike B, must be treated as an untrusted entity and hence the adversary. There is really no alternative but to use cryptographic methods to ensure that key-recovery can be successfully performed when needed. Therefore, we need a cryptographic mechanism which ensures that the cloud must be able to store SK in some "encrypted" form (from where U can access it), verify that SK can be successfully recovered by authorized set of parties as defined by the policy $\mathbb{A}$, and *yet be unable to break* the IND-CPA security of messages encrypted under PK.

Securely storing SK in "encrypted" form in the cloud so that it does not compromise IND-CPA security of encrypted messages is quite easy. Simply use an appropriate Key Encapsulation Mechanism (KEM) [28] to encrypt SK under U's password (or some other appropriate key stored, e.g., in U's employee smartcard). This allows U to access SK, and security can be argued using standard techniques almost automatically (see [28]). In this paper, we focus on the key-recovery part: how to allow the cloud to (non-interactively) verify that SK can be recovered by the legitimate parties and yet ensure that allowing the cloud to do so does not compromise the IND-CPA security of the associated public-key encryption scheme. The cryptographic tool which allows such a public verification is known as *publicly verifiable secret-sharing* (PVSS) scheme. However, plain PVSS schemes do not explicitly consider supporting public-key encryption. Rather, their goal is to simply ensure that there is a *unique* and well defined secret value s that will be recovered by (all) authorized sets of parties. It is not *explicitly* required that $s = SK$ where SK is a legitimate secret-key for PK. This is because such a guarantee is not really needed in cryptographic tasks which use PVSS, e.g., secure function evaluation [9, 11], electronic voting applications [25], and so on. In our context, however, we need to ensure that $s = SK$ without compromising IND-CPA security of messages encrypted under PK.

To do this, we first present a formal security model for public-key encryption schemes that support publicly verifiable secret-sharing schemes. We then observe that even though traditional PVSS schemes do not satisfy all our requirements, the scheme of Stadler [29] can be shown to satisfy these requirements in the Random Oracle Model [4]. The public-key encryption scheme supported by Stadler's construction is the ElGamal cryptosystem [12]. The scheme, however, relies on parallel repetition of zero-knowledge proofs for proving relations about double discrete logarithms [29]. Such parallel repetitions are necessary to reduce the soundness error to an acceptable level, and makes the scheme quite inefficient in practice. Indeed, this was later addressed by Schoenmakers [25] who presented a much more efficient scheme based on simple discrete logarithms (as opposed to the double discrete logarithms). However, Schoenmakers' scheme is only a plain PVSS, i.e., it does not support a public-key encryption scheme.

To address this problem, we construct a new PVSS scheme and a corresponding PKE scheme using pairing-based techniques [16, 8]. Our construction is highly efficient: asymptotically, it is optimal just like Schoenmakers' construction; in addition, our implementation shows that even in practice, despite the use of pairings, the system performs very well when compared to Schoenmakers system (which, to the best of our knowledge, is the fastest known PVSS). The implementation details, and our test results can be found in Section 4.

Our construction has an added theoretical benefit. It is proven secure in the *standard* model, under the standard bilinear Diffie-Hellman (BDH) assumption. All previous (non-interactive) PVSS constructions assume the existence of a Random Oracle. While not our main motivation, this actually resolves an open question in the area of publicly-verifiable secret-sharing schemes: ours is the first construction of a non-interactive PVSS scheme proven secure in the standard model under standard assumptions.

Related Work. To our knowledge, a formal treatment of key-management from the point of view of running key-management services on untrusted computing facilities (such as the cloud) has not previously appeared. Indeed, our work also only focuses on one of the crucial aspects of key-recovery, and does not aim to explicitly provide a full treatment to key-management in the cloud.

Nevertheless, we have been able to focus on the aspect of key-recovery which is common to almost all good key-management services. Our work relies heavily on the techniques of secret-sharing schemes, in particular the standard extension of Blakley-Shamir secret-sharing scheme [5, 27] to access trees [14, 24]. The idea of verifiability in secret-sharing schemes (VSS) was introduced by Chor et al. [9]. Efficient non-interactive versions were presented by Feldman [11], where verifiability of the secret is information-theoretic but secrecy relies on computational assumptions, and by Pedersen [23], where verifiability is only guaranteed computationally while secrecy is unconditional. Publicly verifiable secret-sharing schemes most relevant to our work are those of Stadler [29] and Schoenmakers [25]; both schemes hide the secret computationally.

The idea of using PVSS schemes to enforce verifiability of shares for a secret key is not new in its own, and has been used in a closely related goal of verifiable *key-escrow*. Key-escrow were initially designed with the purpose of allowing the government to recover DES keys to monitor suspected activities [21]. This was followed by extensive research considering various issues to partial key-escrow [20], verifiability in key-escrow [3, 19], and so on (see [3] for a good exposure). In summary, adding verifiability to the key-escrow problem naturally brought the usage of PVSS schemes and variations of the same were developed as needed. We note that the focus of these works is different, and as such a holistic approach to security of PKE was never considered by any of these works. In addition, our scheme is the first non-interactive PVSS proven secure in the standard model.

2 Preliminaries

We assume familiarity with public key encryption scheme [13]. Unless stated otherwise, $\kappa \in \mathbb{N}$ will denote the security parameter. All parties, and mechanisms are assumed to have the security parameter as an implicit input in the form 1^κ, and run in time polynomial in κ. A function is called *negligible* if it approaches zero faster than the inverse of every polynomial.

2.1 Definitions

We first recall the definition of an Access Structure involving parties $P_1, \ldots, P_n$.

Definition 2.1 (Access Structure [2]) *Let $\{P_1, \ldots, P_n\}$ be a set of parties. A collection $\mathbb{A} \subseteq 2^{\{P_1,\ldots,P_n\}}$ is monotone if $\forall B, C$: if $B \in \mathbb{A}$ and $B \subseteq C$ then*

$C \in \mathbb{A}$. *An* access structure *(resp., monotone access structure) is a collection (resp., monotone collection)* $\mathbb{A}$ *of non-empty subsets of* $\{P_1, \ldots, P_n\}$*; i.e.,* $\mathbb{A} \subseteq 2^{\{P_1,\ldots,P_n\}} \setminus \{\emptyset\}$. *The sets in* $\mathbb{A}$ *are called* authorized sets*, and the sets not in* $\mathbb{A}$ *are called* unauthorized sets*.*

In our context, these parties will receive encrypted shares of a secret key. Each party P_i will be defined by its public parameters PP_i to be fixed by the scheme. Description of $\mathbb{A}$, defined over P_i, then includes the public parameters PP_i of relevant parties P_i. Unless stated otherwise, we shall only deal with monotone access structures in this paper.

Let PKE $= \{\mathcal{K}, \mathcal{E}, \mathcal{D}\}$ be a public key encryption scheme. We assume that there exists an efficient algorithm Valid such that $\textsc{Valid}(PK, SK) = 1$ if and only if (PK, SK) is in the range of $\mathcal{K}(1^\kappa)$ for some $\kappa \in \mathbb{N}$.

We now formally define PKE schemes that support publicly verifiable secret sharing (PVSS).

PKE Supporting Public-VSS A public-key encryption scheme supporting publicly verifiable secret sharing for an access structure $\mathbb{A}$ consists of seven algorithms $\{\mathcal{K}, \mathcal{E}, \mathcal{D}, \texttt{Setup}, \texttt{GenShare}, \texttt{Verify}, \texttt{Reconst}\}$ such that the triplet PKE $= \{\mathcal{K}, \mathcal{E}, \mathcal{D}\}$ is an (ordinary) public-key encryption scheme and:

$\texttt{Setup}(1^\kappa, n)$. This is a randomized algorithm. For every $i \in [n]$, it computes a public-value PP_i (defining the party P_i) and a corresponding secret-value SK_i. It outputs the vector of pairs $\{(PP_i, SK_i), \ldots, (PP_n, SK_n)\}$.

$\texttt{GenShare}(PK, SK, \mathbb{A})$. This is a randomized algorithm for generating encrypted shares. It takes as input a public-secret key-pair (PK, SK) and an access structure $\mathbb{A}$; it outputs a string π. Recall that the description of $\mathbb{A}$ includes public parameters PP_j of relevant parties.

$\texttt{Verify}(PK, \pi, \mathbb{A})$. This is a deterministic (verification) algorithm. On input $(PK, \pi, \mathbb{A})$, the algorithm either outputs 1 or 0. We require that for every $\kappa \in \mathbb{N}$ and for every (valid[4]) $\mathbb{A}$:

$$\begin{aligned}\Pr\left[\texttt{Verify}(PK, \pi, \mathbb{A}) = 1 : (PK, SK) \leftarrow \mathcal{K}(1^\kappa) \wedge \pi \leftarrow \texttt{GenShare}(PK, SK, \mathbb{A})\right] \\ = 1.\end{aligned}$$

This requirement is known as the *correctness* condition.

$\texttt{Reconst}(PK, \pi, \mathbb{A}, SK_S)$. This is a deterministic algorithm for reconstructing the secret key SK from (encrypted shares in) π. Formally, let $S \in \mathbb{A}$ be an authorized set, and let $SK_S = \{SK_j\}_{j:P_j \in S}$ be the set of secret keys of parties $P_j \in S$. Algorithm Reconst takes as input $(PK, \pi, \mathbb{A}, SK_S)$ and outputs a string SK'.

[4] $\mathbb{A}$ is defined over P_i, which in turn are defined over PP_i; $\mathbb{A}$ is valid if it satisfies Definition 2.1 and every PP_i is an output of $\texttt{Setup}(1^\kappa, n)$.

Informally, we require that the no polynomial time adversary can produce a (PK^*, π^*) which will be accepted by `Verify` but `Reconst` will fail to recover a valid secret key SK' for PK^*. This requirement is known as the *soundness* condition. The formal definition follows.

Soundness. Formally, we require that there exists a negligible function $\mathrm{negl}(\cdot)$ such that for every valid $\mathbb{A}$, every $S \in \mathbb{A}$, every non-uniform PPT algorithm U^*, and every sufficiently large $\kappa \in \mathbb{N}$:

$$\Pr\left[\begin{array}{c}(PK^*, \pi^*) \leftarrow U^*(\mathbb{A}); SK' \leftarrow \texttt{Reconst}(PK^*, \pi^*, \mathbb{A}, SK_S);\\ \texttt{Verify}(PK^*, \pi^*, \mathbb{A}) = 1 \bigwedge \textsc{Valid}(PK^*, SK') = 0\end{array}\right] \leq \mathrm{negl}(\kappa).$$

Security Game for PKE supporting Public-VSS The security is defined by considering a game played between the challenger and the adversary. We shall give the adversary flexibility to choose the public parameters of the parties it wishes to corrupt. However, we shall only consider *static* corruptions where the adversary chooses these parameters before the challenge phase. The game proceeds in the following phases:

Setup. The challenger runs the `Setup` algorithm to obtain system parameters $\{PP_i, SK_i\}_{i=1}^n$; it then samples ("user") keys $(PK, SK) \leftarrow \mathcal{K}(1^\kappa)$. Public parameters PK and $\{PP_i\}_{i=1}^n$ are sent to the adversary.

Corruption. The adversary "corrupts" a set of parties by sending the following to the challenger: a set $C \subset [n]$ of indices and a public parameter PP_i^* for every $i \in C$. The new public parameters for the system are: $PK, \{PP_i^*\}_{i \in C} \cup \{PP_i\}_{i \in [n] \setminus C}$.

Phase 1. The adversary sends a (valid) access structure $\mathbb{A}^*$ to the challenger such that set C of corrupted parties does not satisfy $\mathbb{A}^*$; that is, $C \notin \mathbb{A}^*$. The challenger runs the `GenShare` algorithm on inputs $(PK, SK, \mathbb{A}^*)$ and sends the resulting output to the adversary.

Challenge. The adversary sends two distinct and equal length messages m_0 and m_1. The challenger samples a random bit b and computes the challenge ciphertext $\mathrm{CT}^* \leftarrow \mathcal{E}_{PK}(m_b)$. Adversary receives CT^*.

Phase 2. Phase 1 is repeated.

Guess. Adversary outputs a guess bit b'.

The advantage of an adversary in this game is defined to be $\Pr[b' = b] - \frac{1}{2}$.

Definition 2.2 *A public-key encryption scheme supporting publicly verifiable secret-sharing is said to be secure in the (static) corruption model if all (non-uniform) polynomial time algorithms have at most a negligible advantage in the security game.*

2.2 Access Trees

We will consider access structures that are representable by a tree of threshold gates. This is a very large class of access structures and have been used in many previous works including attribute-based encryption and verifiable secret sharing. To facilitate working with them, the basic framework of access trees is recalled here.

Access Tree $\mathcal{T}$. Let $\mathcal{T}$ be a tree representing an access structure. Each non-leaf node of the tree represents a threshold gate, described by its children and a threshold value. If num_x is the number of children of a node x and k_x is its threshold value, then $0 \leq k_x \leq num_x$ When $k_x = 1$, the threshold gate is an OR gate and when $k_x = num_x$, it is an AND gate. Each leaf node x of the tree is described by a party P_i and a threshold value k_x.

To facilitate working with the access trees, we define a few functions. We denote the parent of the node x in the tree by parent(x). The access tree $\mathcal{T}$ defines an ordering between the children of every node. That is, the children of a node x are numbered from 1 to num_x. The function id(z) returns such a number associated with the node z. (We assume that there is a publicly known method to assign such index values so that they are unique for every node x). Note that by definition, the leaf nodes do not have any children; instead they are associated with a party in $\{P_1, \ldots, P_n\}$. If x is a leaf node, function id(x) returns the index $i \in [n]$ of the party associated with x.

Satisfying an Access Tree. Let $\mathcal{T}$ be an access tree with root r. Denote by $\mathcal{T}_x$ the subtree of $\mathcal{T}$ rooted at the node x. Hence $\mathcal{T}$ is the same as $\mathcal{T}_r$. If a set $\gamma \subseteq [n]$ of indices satisfies the access tree $\mathcal{T}_x$, we denote it as $\mathcal{T}_x(\gamma) = 1$. We compute $\mathcal{T}_x(\gamma)$ recursively as follows. If x is a non-leaf node, evaluate $\mathcal{T}_{x'}(\gamma)$ for all children x' of node x. $\mathcal{T}_x(\gamma)$ returns 1 if and only if at least k_x children return 1. If x is a leaf node, then $\mathcal{T}_x(\gamma)$ returns 1 if and only if id$(x) \in \gamma$.

2.3 Cryptographic Assumptions

Bilinear Diffie-Hellman (BDH) Assumption. We assume familiarity with bilinear maps (see [16, 8]). Let $\mathbb{G}_1$ be bilinear group of prime order p and generator g. In addition, let $e\colon \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_2$ be the bilinear map with the target group $\mathbb{G}_2$. Let $a, b, c, d \in \mathbb{Z}_p$ be chosen at random and g be a generator of $\mathbb{G}_1$. The BDH assumption [6] states that no (non-uniform) probabilistic polynomial time algorithm $\mathcal{B}$ can distinguish the tuple $(g^a, g^b, g^c, e(g,g)^{abc})$ from the tuple $(g^a, g^b, g^c, e(g,g)^d)$ with more than a negligible advantage. Here, the advantage of $\mathcal{B}$ is defined by:

$$\left|\Pr\left[\mathcal{B}(g^a, g^b, g^c, e(g,g)^{abc}) = 1\right] - \Pr\left[\mathcal{B}(g^a, g^b, g^c, e(g,g)^d) = 1\right]\right|.$$

3 An Efficient Scheme without Random Oracles

In this section we shall present a public-key encryption scheme which will support public-VSS for access trees. This construction is based on bilinear pairings, and is proven secure in the standard model under the (standard) BDH assumption.

As noted before, we will first present an encryption scheme, and then present a publicly verifiable secret sharing for the specific purpose of sharing the decryption keys of this encryption scheme. While our encryption schemes is new, the secret-sharing scheme will follow standard approaches. Nevertheless, since this is the first time, for completeness we will present the secret sharing part in full detail.

Let $\mathbb{G}_1$ be a bilinear group of prime order p, and let g be a randomly chosen generator of $\mathbb{G}_1$. In addition, let $e\colon \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_2$ denote the bilinear map with target group $\mathbb{G}_2$. Note that the security parameter κ determines the size of the groups, and parameters $g, \mathbb{G}_1, \mathbb{G}_2, p$ are available to all parties.

The Encryption Scheme. Our new encryption scheme, PKE $= \{\mathcal{K}, \mathcal{E}, \mathcal{D}\}$, is a variant of the ElGamal encryption scheme. It encrypts messages in $\mathbb{G}_2$. The description can be found in Figure 1.

Key Generation $\mathcal{K}$: $h \xleftarrow{\$} \mathbb{G}_1$. $SK = h$, and $PK = e(g, h)$.

Encryption $\mathcal{E}_{PK}(m \in \mathbb{G}_2)$: $R \xleftarrow{\$} \mathbb{Z}_p$, output: $\langle g^R, m \cdot PK^R \rangle$.

Decryption $\mathcal{D}(\langle C_1, C_2 \rangle, SK)$: Output $C_2/e(C_1, SK)$.

Fig. 1. Encryption scheme PKE.

Observe that for correctly generated ciphertexts $\langle C_1, C_2 \rangle$: $C_2/e(C_1, SK) = m \cdot PK^R/e(g^R, h) = m$, since the denominator $e(g^R, h) = e(g, h)^R = PK^R$. Also observe that corresponding to every public key, there is a *unique* secret key, and it is possible to efficiently test if a proposed secret key SK^* is valid for a given public PK by testing that: $e(g, SK^*) = PK$.

Supporting Public-VSS Property for Access Trees. We complete the description of the remaining four algorithms $\{\texttt{Setup}, \texttt{GenShare}, \texttt{Verify}, \texttt{Reconst}\}$ in our system. Recall that our access structure $\mathbb{A}$ is represented by an access tree $\mathcal{T}$.

For $i \in \mathbb{Z}_p$ and a set S consisting of elements in $\mathbb{Z}_p$, we define the Lagrange coefficient $\Delta_{i,S}(X) = \prod_{j \in S \setminus \{i\}} \frac{X-i}{i-j}$.

$\texttt{Setup}(1^\kappa, n)$. For every $i \in [n]$: sample $y_i \xleftarrow{\$} \mathbb{Z}_p$; output $SK_i = y_i$ and $PP_i = g^{y_i}$.

$\texttt{GenShare}(PK, SK, \mathcal{T})$. Recall that $SK = h$ and $PK = e(g, h)$. Let $s \in \mathbb{Z}_p$ such that $h = g^s$. For clarity, we break the algorithm in three steps.

1. *Define polynomials*: Choose a polynomial q_x for every node x (including the leaves) in the $\mathcal{T}$. These polynomials are chosen in the following way in a top-down manner, starting from the root node r.

For each node x in the tree, set the degree d_x of the polynomial q_x to be one less than the threshold value k_x of that node; that is, $d_x = k_x - 1$. Now, for the root node r, set $q_r(0) = s$. That is, the constant term of q_r is set to s. Choose d_r more points randomly to completely fix the polynomial q_r. For every other node x, set $q_x(0) = q_{\text{parent}(x)}(\text{id}(x))$; i.e., the constant term of q_x is set to $q_{\text{parent}(x)}(\text{id}(x))$. Choose the remaining d_x points randomly to completely define the polynomial q_x.

2. *Encapsulate shares*: For every *leaf* node x, the share of node x is defined by: $\lambda_x = g^{q_x(\text{id}(x))}$. This value can be computed by using polynomial interpolation since all points are known (recall that $\text{id}(x)$ returns the index $i \in [n]$ of the party P_i at the leaf node x). Now, choose a random value $R_x \in \mathbb{Z}_p$; the encapsulation of λ_x is $\langle B_x, C_x \rangle$, where:

$$B_x = g^{R_x}, \qquad C_x = \lambda_x \cdot PP_{\text{id}(x)}^{R_x}.$$

Observe that the encapsulation of λ_x is simply an ElGamal encryption of λ_x under the public parameter $PP_{\text{id}(x)}$.

3. *Proof*: Finally, to enable public verification, the algorithm will "commit" to polynomials of every node x in the target group $\mathbb{G}_2$. For every node x and every $0 \leq i \leq d_x$, define the following values:

$$A_{x,i} = g^{q_x(i)} \quad \text{and} \quad \widehat{A}_{x,i} = e(g, A_{x,i}) = e(g,g)^{q_x(i)}.$$

The output string π consists of the following:

1. For every node x (including the leaf nodes), the "committed polynomial": $\{\widehat{A}_{x,i}\}_{i=1}^{d_x}$;
2. For every leaf node, the encapsulations: $\langle B_x, C_x \rangle$.

`Verify`$(PK, \pi, \mathcal{T})$. The algorithms proceeds in following steps:

1. For every node x in $\mathcal{T}$, parse π to obtain the committed points $\{\widehat{A}_{x,i}\}_{i=1}^{d_x}$ of polynomial q_x. For every leaf node x in $\mathcal{T}$, parse π to obtain the encapsulations $\langle B_x, C_x \rangle$ of secrets λ_x. (Note that λ_x is not publicly known).
2. For the root node, verify that $\widehat{A}_{r,0} = PK$. For every other node x, verify that:

$$\widehat{A}_{x,0} = \prod_{i=0}^{d_z} \left(\widehat{A}_{z,i}\right)^{\Delta_{i,\gamma_z}(w)}, \tag{1}$$

where $z = \text{parent}(x)$, $w = \text{id}(x)$, and the set $\gamma_z = \{0, 1, \ldots, d_z\}$.

3. For every leaf node x, verify that:

$$\widehat{A}_{x,0} = \frac{e(g, C_x)}{e(B_x, PP_i)}, \tag{2}$$

where $i = \text{id}(x)$.

If all tests pass, output 1; otherwise output 0. We quickly note that for correctly generated values all tests do pass, because:

$$\begin{aligned}
\text{RHS of (1)} &= \prod_{i=0}^{d_z} \left(\widehat{A}_{z,i}\right)^{\Delta_{i,\gamma_z}(w)} = e(g,g)^{\sum_{i=0}^{d_z} q_z(i)\cdot\Delta_{i,\gamma_z}(w)} \\
&= e(g,g)^{q_z(w)} = e(g,g)^{q_x(0)} = \widehat{A}_{x,0}, \\
\text{RHS of (2)} &= \frac{e(g,C_x)}{e(B_x,PP_i)} = \frac{e(g,\lambda_x\cdot PP_i^{R_x})}{e(g^{R_x},PP_i)} \\
&= \frac{e(g,\lambda_x)\cdot e(g,PP_i^{R_x})}{e(g,PP_i)^{R_x}} = e(g,\lambda_x) = \widehat{A}_{x,0}.
\end{aligned}$$

$\texttt{Reconst}(PK, \pi, \mathcal{T}, SK_S)$. Informally, the reconstruction procedure works as follows. First "decrypt" the shares λ_x for relevant leaf nodes. Then, apply the standard polynomial interpolation in the exponent recursively (e.g., see [24, 14]). The formal description follows.

For every node x in $\mathcal{T}$, parse π to obtain the committed coefficients $\{\widehat{A}_{x,i}\}_{i=1}^{d_x}$ of polynomial q_x. For every leaf node x in $\mathcal{T}$, parse π to obtain the encapsulations $\langle B_x, C_x\rangle$ of secrets λ_x.

Now, we define a recursive algorithm DecryptNode(π, SK_S, x) that takes as input the string π and the secret key set SK_S (we assume that S is included in SK_S), and a node x in the tree. It outputs an element in $\mathbb{G}_1$ or $\perp$.

Let $i = \mathrm{id}(x)$. If x is a leaf node then let $y_i \in SK_S$ be the secret key corresponding to PP_i. The algorithm is defined as follows: if $i \in S$,

$$\mathrm{DecryptNode}(\pi, SK_S, x) = \frac{C_x}{B_x^{y_i}} = \frac{\lambda_x \cdot PP_i^{R_x}}{g^{R_x\cdot y_i}} = \lambda_x = g^{q_x(0)}.$$

If $i \notin S$, then we define DecryptNode$(\pi, SK_S, x) = \perp$.

We now consider the case when x is not a leaf node. In this case, the algorithm DecryptNode(π, SK_S, x) proceeds as follows: for all nodes z that are *children* of x, it calls DecryptNode(π, SK_S, z) and stores the output as F_z. Let γ_x be an arbitrary k_x-sized set of child nodes z such that $F_z \neq \perp$. If no such set exists then the node was not satisfied and the algorithm returns $\perp$.

Otherwise, compute:

$$\begin{aligned}
F_x &= \prod_{z\in\gamma_x} F_z^{\Delta_{i,\gamma_x'}(0)}, && \text{where } \begin{cases} i = \mathrm{id}(z) \\ \gamma_x' = \{\mathrm{id}(z) : z \in \gamma_x\} \end{cases} \\
&= \prod_{z\in\gamma_x} g^{q_z(0)\cdot\Delta_{i,\gamma_x'}(0)} \\
&= \prod_{z\in\gamma_x} g^{q_{\mathrm{parent}(z)}(\mathrm{id}(z))\cdot\Delta_{i,\gamma_x'}(0)} && \text{(by construction)} \\
&= \prod_{z\in\gamma_x} g^{q_x(i)\cdot\Delta_{i,\gamma_x'}(0)} \\
&= g^{q_x(0)} && \text{(using polynomial interpolation)}
\end{aligned}$$

and return the result.

Having defined the recursive algorithm DecryptNode, our reconstruction algorithm `Reconst` simply calls the function DecryptNode on the root node r of the tree with inputs (π, SK_S). Observe that: $\text{DecryptNode}(\pi, SK_S, r) = g^{q_r(0)} = g^s = SK$ if and only if $\mathcal{T}(S) = 1$ (as desired).

Efficiency. Note that for ease of exposition, we have defined the simplest form of reconstruction algorithm. There are several optimizations possible. See the discussion in [14] on how to minimize the number of exponentiations (ignoring the pairing computations). Note that the `Reconst` algorithm does not perform any pairing computations; the computation cost is thus dominated by number of exponentiations.

On supporting every LSSS-realizable $\mathbb{A}$. Our construction is only described for access trees. However, it can be easily extended to suppose every access structure $\mathbb{A}$ which can be realized by a *linear secret-sharing scheme* (LSSS, see [2]). Such access structures $\mathbb{A}$ are represented by a *monotone span program*. Our construction will commit to the randomness of such secret-sharing scheme instead of committing to the coefficients. The full construction can be obtained by following the details of construction in Section A of [14].

4 System Implementation

Recall that our access structures are composed of a tree of threshold gates. For the purposes of evaluating performance, it suffices to consider a single (k, n)-threshold gate. For such a gate, it is easy to calculate that the theoretical cost of our scheme is $O(n)(T_1 + T_3)$ for `GenShare`, $O(k^2)T_2 + O(n)T_3$ for `Verify`, and $O(k^2 + n)T_1$ for `Reconst`, where T_1, T_2, and T_3 are the costs of a $\mathbb{G}_1$-exponentiation, a $\mathbb{G}_2$-exponentiation, and a pairing respectively. Hence, in terms of asymptotic cost complexity, our scheme has performance similar to [25]. In order to compare the performance of the two schemes in more detail, we implemented the two schemes and measured their running times empirically.

Our implementation is based on Mike Scott's MIRACL library [26]. For the pairing, our protocol requires a type 1 (symmetric) cryptographic pairing. We used the Tate pairing on supersingular elliptic curves over $\mathbb{F}_p$ of embedding degree 2. Although other type 1 pairings lead to a sizable performance improvement [1], we chose the Tate pairing implementation built into MIRACL because it has the advantages of public availability and integration with the supporting MIRACL API. We evaluated the performance of both schemes at the 80, 112, 128, and 256-bit security levels. Following the guidance of [22, Table 1], we used corresponding group sizes of 160, 224, 256, and 512 bits, and field sizes of 1024, 2048, 3072, and 15360 bits respectively. For the pairing-based implementation, the field size is the size of $\mathbb{G}_2$ in bits; the size of $\mathbb{G}_1$ in bits is half that of $\mathbb{G}_2$ (since the embedding degree is 2). All tests were run on an AMD 2.4GHz Opteron in 64-bit mode.

The results of our tests are presented in Appendix B. We observe that, in general, the performance of the two schemes on `GenShare` is comparable. Our scheme is slower for `GenShare` at the 256-bit security level because pairing operations over such large curves are slow. For `Verify`, our scheme is slower than [25] for the smallest measured values of k and faster for the largest values. We expect such a performance improvement in asymptotic terms since our scheme avoids the double exponentiation step of [25, p. 154]. For `Reconst`, our scheme is slower by about a factor of 2, in this case because group operations on large elliptic curves are slow. As mentioned above, one possible strategy for improving performance would be to use pairings on supersingular curves over fields of small characteristic with larger embedding degrees [1]. We mention, however, that execution of `Reconst` is normally needed only in unforeseen circumstances such as the loss of a key, and will not be performed simultaneously for too many users.

5 Security Proof for Our Construction

In this section, we provide a full proof of security of our pairing based scheme. First note that the proof of soundness (of the `Reconst` procedure) is straightforward. Further details can be found in Appendix A. We move on to prove the security of encryption (in the presence of public-VSS). The security of our scheme is proven by reduction to the BDH assumption. We show that if an adversary can win the security game for PKE supporting Public-VSS with non-negligible advantage, then one can construct a simulator to break the BDH assumption.

Theorem 5.1. *If a polynomial time adversary $\mathcal{A}$ wins the security for PKE scheme supporting publicly verifiable secret-sharing scheme, then there exists a polynomial time simulator $\mathcal{B}$ to break the Bilinear Diffie-Hellman Assumption.*

Proof. Suppose that $\mathcal{A}$ can succeed in the security game for PKE supporting public-VSS with advantage ϵ. We construct a simulator $\mathcal{B}$ that succeeds in the decisional BDH game with advantage $\epsilon/2$ or more. The simulation proceeds as follows.

We first let the challenger set the groups $\mathbb{G}_1, \mathbb{G}_2$ of prime order p with an efficient bilinear map e and a generator g. The challenger flips a fair coin μ outside the view of $\mathcal{B}$. If $\mu = 0$ the challenger sets $(A, B, C', D) = (g^a, g^b, g^c, e(g,g)^{abc})$; otherwise, it sets $(A, B, C', D) = (g^a, g^b, g^c, e(g,g)^d)$ for random (a, b, c, d). Now, the simulator initiates the adversary $\mathcal{A}$ interacting with it through various phases as follows.

Setup. $\mathcal{B}$ prepares the following values. First it sets $PK = e(A, B) = e(g,g)^{ab}$. Next, for every $i \in [n]$, it chooses a random value $\beta_i \in \mathbb{Z}_p$ and sets $PP_i = B^{\beta_i} = g^{b\beta_i}$. Adversary receives $(PK, \{PP_i\}_{i\in[n]})$.

Corruption. The adversary corrupts a set $C \subset [n]$ of parties by fixing public parameter PP_i^* of its own choice for every $i \in C$. The new public parameters for the system are: $PK, \{PP_i^*\}_{i\in C} \cup \{PP_i\}_{i\in[n]\setminus C}$.

Phase 1. The adversary sends an access tree $\mathcal{T}$ to the simulator such that $\mathcal{T}(C) = 0$. The simulator needs to respond with a string π as its response to the public VSS query. It proceeds as follows.

Let $s = ab$ so that $PK = e(g,g)^s$ and $y_i = b\beta_i$ so that $PP_i = g^{y_i}$ for every $i \in [n] \setminus C$. The simulator first needs to define a polynomial q_x of degree d_x for every node x. We define the following two procedures to be executed by the $\mathcal{B}$ later: PolySat and PolyUnsat. These are recursive procedures, and append values to the output string π (initially empty).

$\mathrm{PolySat}(\mathcal{T}_x, C, \delta_x)$ This procedure sets up the polynomials for all nodes of an access *sub*-tree whose root node is *satisfied* by parties in C; that is $\mathcal{T}_x(C) = 1$. The inputs to the procedure are: the subtree $\mathcal{T}_x$ rooted at node x of $\mathcal{T}$, the set C, and an integer $\delta_x \in \mathbb{Z}_p$.

The procedure starts by defining a polynomial q_x for node x; it sets $q_x(0) = \delta_x$. It then sets the remaining points of q_x randomly to completely fix the polynomial q_x. For $0 \le i \le d_x$, values $\widehat{A}_{x,i} = e(g,g)^{q_x(i)}$ are then appended to π.

Now, for every child node x' of x, we call $\mathrm{PolySat}(\mathcal{T}_{x'}, C, q_x(\mathrm{id}(x')))$. This fixes the polynomials for every node z in the access sub-tree $\mathcal{T}_x$ and appends relevant values $\widehat{A}_{x,i}$ to π. Note that by construction, all nodes satisfy the constraint that: $q_z(0) = q_{\mathrm{parent}(z)}(\mathrm{id}(z))$.

$\mathrm{PolyUnsat}(\mathcal{T}_x, C, e(g,g)^{\delta_x})$ This procedure sets up the polynomials for all nodes of an unsatisfied access sub-tree $\mathcal{T}_x$; that is $\mathcal{T}_x(C) = 0$. The inputs to the procedure are: the subtree $\mathcal{T}_x$ rooted at node x of $\mathcal{T}$, the set C, and an element $e(g,g)^{\delta_x} \in \mathbb{G}_2$ where $\delta_x \in \mathbb{Z}_p$.

It first defines a polynomial q_x of degree d_x for the root node x such that $q_x(0) = \delta_x$. Since $\mathcal{T}_x(C) = 0$, at most $h_x \le d_x$ children of x are satisfied. For each satisfied child x' of x, the procedure chooses a random value $\delta_{x'}$ and sets $q_x(\mathrm{id}(x')) = \delta_{x'}$. It then fixes the remaining $d_x - h_x$ points of q_x randomly to completely fix the polynomial. Let γ_x be the set of these d_x points where the value of the polynomial is chosen. That is, except for $i = 0$, value of $q(i)$ is known to $\mathcal{B}$ for every $i \in \gamma_x$.

Now the algorithm recursively defines polynomials for the rest of the nodes in the tree as follows. For each child node x' of x, the algorithm calls:

- $\mathrm{PolySat}(\mathcal{T}_{x'}, C, \delta_{x'})$, if x' is a satisfied child node. Note that the value $\delta_{x'} = q_x(\mathrm{id}(x'))$ is chosen by $\mathcal{B}$ in this case.
- $\mathrm{PolyUnsat}(\mathcal{T}_x, C, e(g,g)^{q_x(\mathrm{id}(x'))})$, if x' is an unsatisfied child node. The unknown value $e(g,g)^{q_x(\mathrm{id}(x'))}$ is computed by polynomial interpolation. To see this, we obtain a general formula as follows. First note that:

$$\begin{aligned}
q_x(X) &= \sum_{i \in \gamma_x} q_x(i) \Delta_{i,\gamma_x}(X) \\
&= q_x(0)\Delta_{0,\gamma_x}(X) + \underbrace{\sum_{i \in \gamma_x \setminus \{0\}} q_x(i)\Delta_{i,\gamma_x}(X)}_{\xi_x(X) \quad (=\text{known})} \\
&= \delta_x \cdot \Delta_{0,\gamma_x}(X) + \xi_x(X).
\end{aligned}$$

Then, the following function is computable by $\mathcal{B}$:

$$e(g,g)^{q_x(X)} = \left(e(g,g)^{\delta_x}\right)^{\Delta_{0,\gamma_x}(X)} \cdot e(g,g)^{\xi_x(X)}. \tag{3}$$

Hence, the procedure can compute the input $e(g,g)^{q_x(\mathrm{id}(x'))}$ as needed above.

Before finishing the execution, the procedure computes the values $\widehat{A}_{x,i} = e(g,g)^{q_x(i)}$ for every $0 \le i \le d_x$ using formula (3) and appends it to the output π.

Having defined the two procedures, the simulator runs PolyUnsat($\mathcal{T}, C, PK$). The procedure returns a partially complete output π which includes the committed polynomials corresponding to every node x in $\mathcal{T}$. To complete the output, $\mathcal{B}$ needs to compute the encapsulations corresponding to every leaf node x. These are computed as follows and appended to the string π:

1. If x is a satisfied leaf node (i.e., $\mathrm{id}(x) \in C$), then value $\lambda_x = g^{q_x(0)}$ is known to $\mathcal{B}$. In this case, $\mathcal{B}$ generates the encapsulation as usual; choose $R_x \xleftarrow{\$} \mathbb{Z}_p$ and output $\langle B_x, C_x \rangle$ where $B_x = g^{R_x}$ and $C_x = \lambda_x \cdot PP_i^{R_x} = \lambda_x \cdot B^{\beta_i R_x}$.
2. If x is an unsatisfied leaf node (i.e., $\mathrm{id}(x) \notin C$), then value $\lambda_x = g^{q_x(0)}$ is not known to $\mathcal{B}$. However, by construction of PolyUnsat, we have that:

$$\lambda_x = g^{\xi_1 + ab\cdot\xi_2},$$

where both ξ_1 and ξ_2 are known values (computed recursively using interpolations and functions $\xi_x(X)$ defined above). The simulator sets $R_x = -a\xi_2/\beta_{\mathrm{id}(x)} + R'_x$ for a randomly chosen $R'_x \in \mathbb{Z}_p$. Let $i = \mathrm{id}(x)$, then the encapsulation of λ_x includes:

$$B_x = g^{R_x} = g^{-a\xi_2/\beta_i + R'_x} = g^{R'_x} \cdot A^{-\xi_2/\beta_i};$$
$$C_x = \lambda_x \cdot PP_i^{R_x} = g^{\xi_1 + ab\cdot\xi_2} \cdot \left(g^{b\beta_i}\right)^{-a\xi_2/\beta_i + R'_x} = g^{\xi_1} \cdot B^{\beta_i R'_x}.$$

Hence, $\mathcal{B}$ can compute encapsulations for all unsatisfied nodes as well.

Therefore, the simulator is able to answer the Phase 1 queries of $\mathcal{A}$. Furthermore, these queries are distributed identically to that in the original scheme.

Challenge. $\mathcal{A}$ sends two distinct equal length messages $m_0, m_1 \in \mathbb{G}_2$. The simulator chooses a random bit ν and responds by sending the following values: $\langle C', m_\nu \cdot D \rangle$.

If $\nu = 0$, then $D = e(g,g)^{abc} = PK^c$. In this case, $\langle C', m_\nu \cdot D \rangle$ is a valid encryption of m_ν and distributed identical to the original scheme. Whereas if $\nu = 1$, D is a random element of $\mathbb{G}_2$ and hence the ciphertext $\langle C', m_\nu \cdot D \rangle$ contains no information about m_ν.

Phase 2. The simulator acts exactly as it did in Phase 1.

Guess. $\mathcal{A}$ will submit a guess ν' of ν. If $\nu' = \nu$ the simulator will output $\mu' = 0$ to indicate that it was given a valid BDH-tuple otherwise it will output $\mu' = 1$ to indicate it was given a random 4-tuple.

As shown in the construction, the simulator's generation of public parameters and answers to the queries of $\mathcal{A}$ in all stages are identical to that of the actual scheme.

In the case where $\mu = 1$ the adversary gains no information about ν. Therefore, we have $\Pr[\nu \neq \nu' \mid \mu = 1] = \frac{1}{2}$. Since the simulator guesses $\mu' = 1$ when $\nu \neq \nu'$, we have $\Pr[\mu' = \mu \mid \mu = 1] = \frac{1}{2}$.

If $\mu = 0$ then the adversary sees an encryption of m_ν. The adversary's advantage in this situation is ϵ by definition. Therefore, we have $\Pr[\nu = \nu' \mid \mu = 0] \geq \frac{1}{2} + \epsilon$. Since the simulator guesses $\mu' = 0$ when $\nu = \nu'$, we have $\Pr[\mu' = \mu \mid \mu = 0] \geq \frac{1}{2} + \epsilon$.

The overall advantage of the simulator in the Decisional BDH game is equal to $\frac{1}{2}\Pr[\mu' = \mu \mid \mu = 0] + \frac{1}{2}\Pr[\mu' = \mu \mid \mu = 1] - \frac{1}{2} \geq \frac{1}{2}(\frac{1}{2} + \epsilon) + \frac{1}{2} \cdot \frac{1}{2} - \frac{1}{2} = \frac{1}{2}\epsilon$.

References

1. Diego F. Aranha, Julio López, and Darrel Hankerson. High-speed parallel software implementation of the η_t pairing. In Josef Pieprzyk, editor, *CT-RSA*, volume 5985 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 2010.
2. Amos Beimel. *Secure Schemes for Secret Sharing and Key Distribution*. PhD thesis, Israel Institute of Technology, Technion, Haifa, Israel, June 1996.
3. Mihir Bellare and Shafi Goldwasser. Verifiable partial key escrow. In *ACM Conference on Computer and Communications Security*, pages 78–91, 1997.
4. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.
5. George Robert Blakley, Jr. Safeguarding cryptographic keys. In *AFIPS 1979, National Computer Conference*, volume 48, pages 313–317, 1979.
6. Dan Boneh and Xavier Boyen. Efficient selective-ID secure identity-based encryption without random oracles. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology—EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 2004.
7. Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *Advances in Cryptology—CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2001.
8. Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. *SIAM J. Comput.*, 32(3):586–615, 2003. Ealier version in [7].
9. Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults (extended abstract). In *26th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 383–395. IEEE, 1985.
10. Mache Creeger. Cloud computing: An overview. *Queue*, 7:2:3–2:4, June 2009.
11. Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 427–437. IEEE, 1987.

12. Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology—Proceedings of CRYPTO '84*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 1985.
13. Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.
14. Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 89–98. ACM, 2006.
15. Antoine Joux. A one round protocol for tripartite Diffie-Hellman. In Wieb Bosma, editor, *ANTS*, volume 1838 of *Lecture Notes in Computer Science*, pages 385–394. Springer, 2000.
16. Antoine Joux. A one round protocol for tripartite Diffie-Hellman. *J. Cryptology*, 17(4):263–276, 2004. Earlier version in [15].
17. Mike Klien. Six Benefits of Cloud Computing, 2010. `http://resource.onlinetech.com/the-six-benefits-of-cloud-computing/`.
18. Luther Martin. Federated Key Management for Secure Cloud Computing. Presentation by Voltage Security, Inc.: `http://storageconference.org/2010/Presentations/KMS/17.Martin.pdf`, May 2010.
19. Silvio Micali. Fair public-key cryptosystems. In Ernest F. Brickell, editor, *Advances in Cryptology—CRYPTO '92*, volume 740 of *Lecture Notes in Computer Science*, pages 113–138. Springer, 1993.
20. Silvio Micali and Adi Shamir. Partial key-escrow. Manuscript, 1996.
21. Escrowed encryption standard (EES). FIPS PUB 185, National Institute of Standards and Technology, February 1994.
22. National Institute of Standards and Technology. *NIST Special Publication 800-57: Recommendation for Key Management — Part 1: General (revised)*, 2007.
23. Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology—CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1992.
24. Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *Advances in Cryptology—EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 457–473. Springer, 2005.
25. Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In Michael J. Wiener, editor, *Advances in Cryptology—CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 148–164. Springer, 1999.
26. Michael Scott. *MIRACL—A Multiprecision Integer and Rational Arithmetic C/C++ Library*. Shamus Software Ltd, Dublin, Ireland, 2010. Available at `http://www.shamus.ie/`.
27. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
28. Victor Shoup. Encryption algorithms—part 2: Asymmetric ciphers. Final Committee Draft 18033-2, ISO/IEC, December 2004. `http://www.shoup.net/iso/std6.pdf`.
29. Markus Stadler. Publicly verifiable secret sharing. In Ueli M. Maurer, editor, *Advances in Cryptology—EUROCRYPT '96*, volume 1070 of *Lecture Notes in Computer Science*, pages 190–199. Springer, 1996.

A Proof of Soundness

Proof of Soundness. Let U^* be an arbitrary non-uniform PPT adversary. Let $\mathcal{T}$ be an arbitrary access tree (representing an access structure $\mathbb{A}$), and let S be such that $\mathcal{T}(S) = 1$. Let (PK^*, π^*) be the output of U^* on input $\mathcal{T}$ (we assume that the advice string is in-built in the description of U^*). If $\texttt{Verify}(PK^*, \pi^*, \mathcal{T}) = 1$ then we have the following:

1. For a node x, let $\{\widehat{A}^*_{x,i}\}_{i=1}^{d_x}$ be the values parsed by `Verify`. From the uniqueness of the discrete logarithm in the prime-order groups, we have that for every x and every $0 \leq i \leq d_x$, there exists a *unique* value $\alpha^*_{x,i}$ such that $\widehat{A}^*_{x,i} = e(g,g)^{\alpha^*_{x,i}}$. This fixes a *unique* polynomial of degree d_x for the node x.Also the algorithm tests that for the root node, $\widehat{A}^*_{r,0} = PK = e(g,g)^s$; we have that $q^*_r(0) = s$.
2. Let x be a leaf node, and let $\langle B^*_x, C^*_x \rangle$ be the parsed encapsulated values. First, we claim that $\text{DecryptNode}(\pi^*, SK_S) = g^{q^*_x(0)}$ for every x such that $\text{id}(x) \in S$.
   From the test in (2), we have that there exist unique $R^*_x \in \mathbb{Z}_p$ and $\lambda^*_x \in \mathbb{G}_1$) such that: $B^*_x = g^{R^*_x}$, $C^*_x = \lambda^*_x \cdot PP^{R^*_x}_{\text{id}(x)}$, and $\widehat{A}^*_{x,0} = e(g, \lambda^*_x)$. This implies that $\lambda^*_x = g^{q^*_x(0)}$. Observe that the output of $\text{DecryptNode}(\pi^*, SK_S)$ is λ^*_x if $\text{id}(x) \in S$. This proves the claim.
3. Finally, from the test (1), we have for every node x: $q^*_x(0) = q^*_{\text{parent}(x)}(\text{id}(x))$. It follows that for every node x, the value F_x computed by the `Reconst` algorithm returns $g^{q^*_x(0)}$. As a result, the output of the `Reconst` algorithm is: $F_r = g^{q^*_r(0)} = SK$.

This establishes the soundness of the protocol.

B Empirical benchmarks

In the following tables, we list the observed timings of our implementation of the `GenShare`, `Verify`, and `Reconst` algorithms. For comparison, we also implemented and measured the performance of the corresponding algorithms in Schoenmakers' scheme [25], and both sets of timings are provided in the tables below. For further details regarding our implementation, see Section 4.

80 bit	$k = 1$	5	10	15	20	25	30	35	40	45	50
$n = 10$	70 110	80 110	80 110								
15	110 160	110 160	120 160	110 170							
20	160 210	150 210	150 230	150 220	160 220						
25	190 270	190 260	200 280	190 270	200 280	190 270					
30	230 310	230 320	230 320	220 310	230 320	230 320	230 330				
35	270 360	260 370	270 360	270 370	260 370	270 370	270 380	260 390			
40	300 410	300 420	310 420	300 430	300 420	290 430	300 440	300 430	310 440		
45	350 460	340 470	330 470	350 470	340 470	350 480	350 480	340 490	340 480	340 520	
50	380 520	380 520	380 520	370 520	370 520	380 540	380 530	390 540	380 540	370 560	370 540

112 bit	$k = 1$	5	10	15	20	25	30	35	40	45	50
$n = 10$	320 360	320 370	320 410								
15	480 540	480 560	480 560	470 580							
20	630 740	640 730	670 740	630 750	650 770						
25	790 910	790 910	790 930	810 930	800 950	800 960					
30	980 1080	970 1090	960 1100	960 1120	990 1130	970 1130	980 1170				
35	1120 1270	1120 1260	1110 1290	1120 1290	1120 1300	1140 1310	1110 1320	1120 1350			
40	1280 1440	1290 1440	1300 1450	1280 1460	1370 1480	1280 1490	1270 1500	1280 1520	1280 1540		
45	1450 1620	1450 1640	1440 1620	1470 1630	1440 1660	1440 1670	1430 1840	1430 1690	1460 1730	1450 1720	
50	1590 1810	1600 1790	1580 1830	1590 1810	1590 1850	1590 1850	1600 1880	1610 1880	1590 1880	1600 1890	1600 1900

128 bit	$k = 1$	5	10	15	20	25	30	35	40	45	50
$n = 10$	760 830	760 830	770 870								
15	1150 1210	1140 1260	1140 1270	1140 1280							
20	1530 1600	1520 1630	1520 1640	1560 1670	1520 1750						
25	1880 2010	1890 2020	1900 2050	1900 2080	1890 2120	1890 2120					
30	2290 2400	2260 2410	2290 2440	2250 2480	2260 2520	2280 2810	2270 2560				
35	2700 2830	2650 2830	2680 2880	2650 2880	2660 2900	2650 2940	2670 2990	2700 3020			
40	3100 3180	3030 3220	3030 3280	3060 3300	3020 3500	3170 3330	3020 3360	3060 3410	3050 3430		
45	3440 3630	3470 3650	3380 3650	3420 3650	3410 3690	3450 3740	3400 3760	3400 3780	3450 3860	3400 3840	
50	3800 4000	3800 4040	3810 4090	3810 4070	3790 4120	3780 4430	3780 4150	3790 4250	3780 4240	3770 4230	3940 4290

256 bit	$k = 1$	5	10	15	20	25	30	35	40	45	50
$n = 10$	44140 30430	44480 31570	44740 32570								
15	66920 45660	66620 46530	66790 47870	66410 48840							
20	88360 60290	88190 61160	89870 62580	88760 64060	92600 70620						
25	111120 75400	110910 76410	111830 77460	111600 78900	110720 80530	110660 81100					
30	133120 90520	133640 91350	133580 92510	132860 94550	133010 95400	133350 96240	132520 97790				
35	155400 105040	155310 106440	157830 108020	155500 108950	167470 110310	154880 111290	155760 113630	155460 114240			
40	178990 120650	180610 121830	193690 122900	176750 123580	179140 125150	180810 126980	177330 127510	177860 129210	178090 129470		
45	199700 136270	198990 135750	198270 137610	199030 138630	202850 139540	200080 142110	199290 142130	201540 162080	201210 144850	220940 146050	
50	223930 151200	222360 150740	221820 152370	222500 154600	220730 155340	226480 156660	221660 158420	221080 158790	224450 159460	220410 162610	221810 170520

Fig. 2. Time in milliseconds for `GenShare`, at various security levels, for selected values of k and n. Top numbers in each cell are for our scheme; bottom numbers are for [25].

80 bit	$k = 1$	5	10	15	20	25	30	35	40	45	50
$n = 10$	100 70	100 80	130 120								
15	140 110	150 130	170 180	220 230							
20	180 140	190 150	210 200	260 280	330 370						
25	230 180	240 200	260 240	310 320	380 440	480 580					
30	270 210	290 230	310 280	360 370	420 480	520 620	630 800				
35	320 250	330 270	360 320	390 410	480 550	560 690	670 870	830 1100			
40	370 280	380 300	400 360	450 460	520 580	610 760	710 940	850 1180	1020 1450		
45	420 320	420 340	450 400	500 510	570 640	650 810	770 1020	910 1300	1090 1610	1250 1860	
50	470 380	470 380	490 440	540 550	620 690	700 880	820 1100	970 1390	1110 1640	1330 1980	1490 2310

112 bit	$k = 1$	5	10	15	20	25	30	35	40	45	50
$n = 10$	410 300	430 330	530 480								
15	610 430	640 570	720 630	940 900							
20	830 600	840 630	950 790	1150 1160	1390 1470						
25	1020 730	1040 780	1130 960	1310 1310	1610 1680	1940 2210					
30	1210 880	1230 940	1320 1130	1520 1420	1800 1880	2160 2430	2600 3180				
35	1420 1170	1450 1090	1540 1280	1710 1610	1990 2070	2350 2650	2790 3400	3350 4300			
40	1600 1170	1700 1230	1820 1440	1930 1780	2220 2260	2560 2890	2990 3670	3700 4580	4310 5710		
45	1810 1330	1840 1600	1910 1610	2120 1980	2450 2500	2740 3140	3240 3940	3730 4910	4390 6080	5110 7290	
50	2010 1450	2040 1550	2170 1770	2340 2150	2600 2730	2950 3390	3380 4220	3940 5210	4560 6390	5280 7700	6060 9130

128 bit	$k = 1$	5	10	15	20	25	30	35	40	45	50
$n = 10$	990 690	1050 780	1280 1120								
15	1510 1050	1550 1150	1770 1510	2170 2130							
20	1980 1390	2040 1490	2310 1880	2700 2510	3280 3510						
25	2470 1760	2530 1860	2740 2230	3190 2930	3770 3900	4590 5230					
30	3020 2090	3020 2230	3240 2620	3640 3340	4250 4410	5040 5680	6060 7430				
35	3520 3020	3560 2600	3780 3030	4200 3750	4760 4830	5570 6220	6560 7940	8380 10060			
40	4030 2770	4070 2910	4340 3410	4670 4210	5280 5350	6140 6740	7030 8550	8290 10800	9640 13550		
45	4480 3150	4520 3300	4720 3860	5160 4600	5790 5800	6870 7300	7550 9210	8730 11350	10210 14000	11700 16990	
50	4960 3480	5140 3670	5410 4200	5610 5200	6220 6270	7020 7930	8030 9810	9210 12260	10580 14930	12200 17960	14240 21640

256 bit	$k = 1$	5	10	15	20	25	30	35	40	45	50
$n = 10$	67990 28930	70720 31970	79360 45790								
15	101660 43570	105260 47030	112520 60520	129060 86390							
20	136150 57780	138440 61270	146660 75580	166150 101290	189490 140420						
25	173420 72300	173300 76110	182370 90900	200640 117560	222070 156970	265640 207800					
30	205330 87170	207350 90880	215660 108450	231710 133560	258000 172690	297980 249110	328120 292050				
35	240600 101080	241060 106180	262750 122500	266270 153180	290560 207970	321650 244380	361750 311540	411490 392170			
40	274870 115490	276150 121170	288990 137970	299010 168420	325440 207840	355200 262240	397230 330940	448800 413090	496840 509530		
45	306960 130950	307270 134880	316040 151680	333910 182950	374740 223810	389810 318080	428760 349830	491120 436530	535690 532520	594150 656010	
50	400070 144880	342270 149700	352770 167680	368720 199020	392030 243940	425850 341040	464270 372970	516150 458320	565050 556530	630380 670890	704710 798930

Fig. 3. Time in milliseconds for `Verify`, at various security levels, for selected values of k and n. Top numbers in each cell are for our scheme; bottom numbers are for [25].

80 bit	$k=1$	5	10	15	20	25	30	35	40	45	50
$n=10$	0 0	20 10	110 50								
15	10 0	30 10	120 40	280 110							
20	0 0	30 20	120 50	270 110	490 200						
25	0 0	30 10	120 40	270 100	480 190	850 300					
30	10 0	30 10	110 50	270 100	480 190	770 320	1160 430				
35	10 0	30 10	110 50	270 110	500 190	780 290	1150 420	1600 590			
40	0 0	30 10	120 50	270 110	490 190	780 290	1130 420	1560 590	2080 770		
45	0 0	30 10	110 40	260 100	490 180	780 290	1140 420	1580 580	2170 760	2640 990	
50	0 0	30 10	110 50	260 100	490 190	790 290	1130 430	1590 630	2080 760	2790 970	3300 1200

112 bit	$k=1$	5	10	15	20	25	30	35	40	45	50
$n=10$	10 0	100 40	460 190								
15	0 10	100 50	440 180	1100 430							
20	0 0	110 40	460 180	1090 420	2000 770						
25	10 0	110 40	450 180	1070 410	1960 780	3190 1220					
30	10 0	100 40	440 180	1060 410	1960 780	3140 1200	4660 1840				
35	0 0	110 50	460 180	1080 420	1970 830	3160 1200	4630 1760	6410 2460			
40	10 10	90 40	460 180	1060 410	1970 760	3160 1200	4590 1760	6430 2440	8490 3260		
45	0 0	110 40	440 180	1120 410	1960 760	3130 1200	4820 1770	6370 2420	8420 3240	10900 4120	
50	10 10	100 40	460 180	1070 410	2070 770	3160 1210	4610 1760	6350 2410	8400 3250	10730 4110	13570 5100

128 bit	$k=1$	5	10	15	20	25	30	35	40	45	50
$n=10$	20 10	220 90	1020 440								
15	20 10	220 100	980 410	2420 1010							
20	10 10	220 100	1000 420	2370 990	4410 1890						
25	20 0	220 110	1000 420	2390 990	4330 1850	7080 2930					
30	10 10	220 90	990 420	2350 980	4350 1850	6930 2910	10360 4300				
35	10 0	250 100	1020 430	2400 990	4460 1820	6990 2900	10360 4250	14230 5920			
40	20 10	210 90	1000 430	2350 1000	4340 1840	6960 2900	10190 4230	14120 5880	18830 7960		
45	10 10	230 110	980 410	2360 1000	4330 1800	7120 2890	10150 4230	14110 5830	18680 7710	24030 9900	
50	10 0	240 100	990 430	2380 980	4350 1830	6980 2930	10240 4220	14050 5900	18620 7820	23920 9900	30170 12510

256 bit	$k=1$	5	10	15	20	25	30	35	40	45	50
$n=10$	510 260	8330 3880	39510 18260								
15	520 260	8670 4120	37310 17440	93260 43280							
20	530 250	8720 3970	38750 17800	90380 42160	171110 79370						
25	510 250	8670 4090	38060 17660	90820 42440	168180 78020	288340 126620					
30	510 250	8310 3890	37160 17320	92560 42160	166860 77490	268980 124620	408860 185240				
35	510 250	9400 4320	38790 17880	91470 42550	169770 78280	269710 145900	402200 183820	563480 256290			
40	520 260	7930 3720	38490 17840	91830 42180	167840 78040	268260 125120	395790 183340	552200 253250	731230 339000		
45	520 250	9140 4070	37070 17250	90830 42350	166960 77050	268720 124730	394180 182300	548450 254770	726240 336120	938150 430460	
50	510 250	9080 4150	38140 17670	90530 42880	169500 78680	272900 125350	393910 183860	547660 253640	731220 333800	929300 429270	1169760 536560

Fig. 4. Time in milliseconds for `Reconst`, at various security levels, for selected values of k and n. Top numbers in each cell are for our scheme; bottom numbers are for [25].