Civitas: Implementation of a Threshold Cryptosystem

Adam M. Davis   Dmitri Chmelev   Michael R. Clarkson

{amd16,dc566,clarkson}@cs.cornell.edu
Department of Computer Science
Cornell University

Computing and Information Science Technical Report
http://hdl.handle.net/1813/11661

December 22, 2008

Civitas: Implementation of a Threshold Cryptosystem[*]

Adam M. Davis Dmitri Chmelev Michael R. Clarkson
{amd16,dc566,clarkson}@cs.cornell.edu
Department of Computer Science
Cornell University

December 22, 2008

Abstract

This paper describes the implementation of a threshold cryptosystem for Civitas, a secure electronic voting system. The cryptosystem improves the availability of Civitas by enabling tabulation to complete despite the failure of some agents. The implementation includes a sophisticated distributed key generation protocol, which was designed by Gennaro, Jarecki, Krawczyk, and Rabin. The cryptosystem is implemented in Jif, a security-typed language.

1 Introduction

Voting systems are hard to make trustworthy because they have strong, conflicting security requirements: voters must be convinced that their votes are tallied correctly, while the secrecy of those votes must also be maintained—even when someone tries to buy votes or physically coerce voters. Civitas [1] is an electronic remote voting system that satisfies these requirements and offers assurance through both cryptographic security proofs and information-flow analysis.

The original implementation of Civitas used a distributed El Gamal cryptosystem. A *cryptosystem* is a set of three protocols—key generation, encryption, and decryption—and a *distributed* cryptosystem is a cryptosystem in which a set of agents must cooperate to perform decryption. A distributed cryptosystem improves

[*]Supported in part by National Science Foundation grant 0430161. Michael Clarkson is supported by an Intel Foundation Ph.D. Fellowship.

confidentiality by preventing any single agent from decrypting messages. However, the cryptosystem used in Civitas requires all agents to be present throughout decryption, thus weakening availability.

In the work reported here, we replace Civitas's cryptosystem with a threshold cryptosystem that uses *Shamir secret sharing* [8] for the private key. If no more than t agents fail, then the decryption protocol is guaranteed to terminate with a correct decryption. An agents *fails* when it deviates from the specified protocol, perhaps by crashing, by exhibiting malicious behavior, by committing accidental errors, etc.; an agent that has not failed is *honest*. Similarly, the key generation protocol is guaranteed to terminate with a correctly shared private key if no more than t agents fail. Moreover, no collusion of t or fewer agents can gain any information about the distributed private key. This threshold cryptosystem offers a better tradeoff between confidentiality and availability than the original cryptosystem.

2 Design

2.1 Key Generation

We use the distributed key generation protocol proposed by Gennaro et al. [6]. The protocol runs in two phases: commitment to a jointly generated secret x, and extraction of the public key. Assuming that at most $t < n/2$ tellers can fail, the protocol guarantees that a set of at least $t + 1$ tellers that follow the protocol correctly is able to successfully generate a key, and that any collusion of up to t tellers does not reveal the value of x. The value of x is never revealed by the protocol itself.

2.1.1 Commitment phase

We assume an El Gamal group $\mathcal{G}$ described by parameters p, q and g, where p and q are primes, such that $p = 2kq + 1$ for some integer constant $k > 0$, group $\mathcal{G}$ is the order q subgroup of $\mathbb{Z}_p^*$, and g is a generator of $\mathcal{G}$.

Each teller i selects two random polynomials $f_i(z) = a_{i0} + a_{i1}z + .. + a_{it}z^t$ and $f'_i(z) = b_{i0} + b_{i1}z + .. + b_{it}z^t$ of degree t. (Recall that t is the maximum number of tellers that is allowed to fail.) The coefficients of each polynomial are randomly chosen from $\mathbb{Z}_q$. Let $z_i = a_{i0} = f_i(0)$; this is the random value that teller i contributes to the joint secret x. Each teller commits to the selected polynomials by running an instance of Pedersen-VSS [7], using f_i and f'_i and an additional parameter h randomly selected from $\mathcal{G}$. Pedersen-VSS is a verifiable secret-sharing scheme that allows one of the agents, called the *dealer*, to share a secret with a set of n agents, such that a set of $t+1$ honest agents can reconstruct the secret and detect whether the dealer was malicious.

Every teller sends points $s_{ij} = f_i(j) \bmod q$ and $s'_{ij} = f'_i(j) \bmod q$ to every other teller j on *private channels*, which must be authenticated and confidential. Call these points the *private commitments* of teller i. Since z_i is a term in f_i, every point s_{ij} contains information about z_i. Moreover, given at least $t+1$ points, z_i can be successfully recovered. To prove that the private commitments are correct, each teller also publishes

$$C_{ik} = g^{a_{ik}} h^{b_{ik}} \bmod p \quad \text{for } k \in [0...t].$$

Each teller i can verify the correctness of the private commitments received from teller j by checking that

$$g^{s_{ji}} h^{s'_{ji}} = \prod_{k=0}^{t} (C_{jk})^{i^k} \bmod p \quad \text{for } j \in [1...n]. \tag{1}$$

If the private commitments fail the check, then teller i issues a *complaint* against teller j. Teller j responds to this complaint by revealing s_{ji} and s'_{ji} that match (1) to all tellers. A teller that receives more than t complaints or fails to provide a response that satisfies (1) is automatically disqualified and its z_i is not included in the joint secret x. A teller that receives fewer than t complaints or responds with the correct private commitment is allowed to continue participating in the key generation protocol. Note that tellers that deviate from the protocol can generate at most t complaints against a teller i that behaves correctly. These complaints reveal at most t points of f_i, but at least $t+1$ points are necessary to determine the value of z_i. Thus complaints and responses do not reveal any information about the joint secret.

At the end of the commitment phase each teller calculates a set of qualified tellers, $Qual$, that correctly shared s_{ij} and s'_{ij}. Gennaro et al. prove that this set is the same for all tellers. The secret key share for teller i is

$$x_i = \sum_{j \in Qual} s_{ji} \bmod q.$$

The public key share for teller i is

$$y_i = g^{z_i} \bmod p.$$

The joint secret—that is, the shared private key—is

$$x = \sum_{i \in Qual} z_i \bmod q.$$

2.1.2 Public key extraction phase

The tellers extract the value of the shared public key Y via Feldman-VSS [4]. Feldman-VSS is a verifiable secret sharing scheme that extends Shamir secret sharing by allowing the recipients of the shares to verify that they are *consistent*—that is, any set of $t+1$ shares identifies the secret selected by the dealer. If a teller i misbehaves during this phase, then its contribution to x can be reconstructed from the set $\{s_{ij} \mid j \in Qual\}$ of private commitments by a set of at least $t+1$ honest tellers from $Qual$, using Lagrange interpolation to reconstruct $f_i(0)$.

Hereafter, we consider only tellers that are members of $Qual$. Each teller i publishes

$$A_{ik} = g^{a_{ik}} \bmod p \quad \text{for } k \in [0...t].$$

Note that A_{i0} is the value of the public key share y_i of teller i. Each teller i can verify the correctness of the values A_{jk} received from teller j by verifying that

$$g^{s_{ji}} = \prod_{k=0}^{t} (A_{jk})^{i^k} \bmod p. \tag{2}$$

If the check fails, then teller i complains against teller j by publicly revealing values s_{ji} and s'_{ji} that fail (2) but satisfy (1).

If at least one valid complaint is issued against teller i, then a set of at least $t+1$ honest tellers can reconstruct z_i by running the reconstruction phase of Pedersen-VSS, using Lagrange interpolation:

$$z_i = \sum_{j \in Qual} s_{ij} \cdot \lambda_{j,Qual} \bmod q,$$
$$\lambda_{j,Qual} = \prod_{l \in Qual \setminus \{j\}} \frac{l}{l-j}.$$

The value of the public shared key is

$$Y = \prod_{i \in Qual} y_i \bmod p. \tag{3}$$

A detailed proof of correctness of the above scheme can be found in [6] and is omitted here.

2.1.3 Communication channels

Public messages require only authenticity and integrity, which are already provided by the existing bulletin board interface in Civitas.

The communication model used in [6] assumes the existence of private channels, because Pedersen-VSS requires that the private commitments are sent privately. We provide such a channel using a combination of encryption, signatures, and the bulletin board, which was already implemented in Civitas. Each private message, along with the identifiers of the tellers involved, is signed under the private key of the sender. The result, together with the identifiers of the tellers involved, is encrypted under the public key of the recipient and posted to the bulletin board. This approach binds the identifiers of the principals involved to both layers of signing and encryption, defending against attacks in [3]. Our implementation guarantees secrecy, authenticity and integrity of the private messages and satisfies the requirements of the communication model in [6].

2.2 Encryption

Encryption under the threshold scheme is identical to the non-distributed, non-threshold case using Y, defined in (3), as the public key.

2.3 Decryption

We use the decryption scheme proposed by Cramer et al. [2]. Let ciphertext c be the encryption of message M under key Y with random nonce r, where

$$\begin{aligned} c &= (a, b), \\ a &= g^r \bmod p, \\ b &= M \cdot Y^r \bmod p. \end{aligned}$$

The decryption protocol is as follows:

- Teller i publishes share $a_i = a^{x_i} \bmod p$, along with a zero-knowledge proof that
$$\log_g g^{x_i} = \log_a a_i. \tag{4}$$
- Teller i collects the shares posted by the other tellers in $Qual$.
- Teller i discards the shares that fail the proof.

- Teller i computes, using a set Λ of at least $t+1$ shares that satisfy (4),

$$\begin{aligned} A &= \prod_{j\in\Lambda} a_j^{\lambda_{j,\Lambda}} \bmod p \\ &= a^{\sum_{k\in Qual} z_k \bmod q} \bmod p \\ &= Y^r \bmod p, \end{aligned}$$

  where

$$\lambda_{j,\Lambda} = \prod_{l\in\Lambda\setminus\{j\}} \frac{l}{l-j}.$$

The decryption of c is $\frac{b}{A} \bmod p = M$.

3 Implementation

3.1 Integration and New Functionality

Even though the algorithms for encryption and decryption under the pre-existing distributed scheme are not identical to those in the new scheme, we successfully reused parts of the existing implementation in implementing the threshold scheme. Whereas encryption remained unchanged, decryption required only minor modifications. In contrast, the key generation protocol was replaced almost entirely and necessitated the implementation of two new kinds of channels.

Civitas already had all the functionality required for generating members of El Gamal groups and mathematical operations within those groups, as well as methods to compute zero-knowledge proofs to exhibit the knowledge and equality of discrete logarithms. The latter is necessary for the verification of decryption shares as part of the distributed decryption process, the former is used in key generation, encryption and decryption.

3.1.1 Decryption

We did not change the process by which decryption shares are generated; however, we did change the mechanism by which decryption shares are combined. Civitas originally reported an error and failed upon identifying a single missing or incorrect decryption share, where *incorrect* means that the share's accompanying proof failed to verify. Such occurrences are now merely logged and the process is allowed to continue. During combination, the number of correct shares is compared to the minimum requirement $(t+1)$ and, if insufficient, an error is raised and decryption is abandoned. We also changed combination of decryption shares to use Lagrange coefficients.

To support the new functionality, we made some minor additions to the ElGamalParameters class, extending it with additional fields: t, denoting the maximum number of failed nodes to be tolerated, and h, needed for Pedersen-VSS. Both parameters are shared by all participants.

3.1.2 Private Channels

The security proof of Gennaro et al. [6] assumes channels that prevent modification of messages, and untappable channels that ensure confidentiality between pairs of nodes. But according to both Cramer et al.[2] and Gennaro [5], these assumptions can be discharged with appropriate cryptographic mechanisms for confidentiality, authentication, and integrity. Thus, on top of the bulletin board we implemented two channel abstractions, one private but both providing authentication and integrity. The basic, non-private channel utilizes cryptographic hashes (for integrity) and signatures (for authenticity). The private channel adds encryption to these features for confidentiality. The channel itself is implemented in jif-src/tabulation/server/BBChannel.jif, with two additional interfaces used for the message types it sends: XMLSerializableBroadcast, and a sub-interface XMLSerializableUnicast, both in jif-src/common.

The key generation protocol in the new cryptosystem is fundamentally different than that which previously existed in Civitas. Along with the communication channels just described, this protocol represents the most significant changes to the codebase. The message classes used by the key generation protocol are implemented in jif-src/common:

- Commitment
- PrivProofResponse
- PubKeyShareComplaint
- PublicKeyShareList
- PublicKeyVote
- ReconInfo
- SharePoint
- SharePointComplaints
- ShareResponsesBase

In addition, we added several new methods to XMLDeserializers to facilitate their recovery from the XML format in which they are posted to the BulletinBoard. A helper class, QualSet, also in jif-src/common, manages the qualification status of tellers during the key generation process. We added the Point class to jif-src/crypto to provide a holder for the actual representation of the pair of polynomial evaluations that make up a SharePoint, which is a CivitasBigInteger. The concrete implementation of that class, PointC, can be found in java-src/crypto/concrete.

We added several new functions to CryptoFactoryC (and their signatures to CryptoFactory):

- generatePolynomial
- generateThreshKeyGenCommitment
- evalPolyModQ (despite its name, it does not perform modular arithmetic)
- validateSharePointVsCommitment
- getMaxNumberOfFailedTellers
- aggregateSharePointsSumFactory
- thresholdPublicKeyShareListFactory
- constructKeyPairShare
- validatePublicKeyShareListVsPoint
- reconstructPublicShare
- productModP
- elGamalKeyShareFactory
- elGamalPublicKeyFactory

In implementing key generation, we had to address a few non-obvious details:

- To compute a given Lagrange coefficient we first compute the value of the denominator using BigInteger values, then take the modular inverse of the result and perform a modular multiplication by the BigInteger value of the numerator. The resulting value still belongs to $\mathcal{G}$ and is represented as a BigInteger.
- Calculation of s_{ij}: The s_{ij} values are not members of $\mathcal{G}$, and thus are calculated (when evaluating the polynomials) without modular arithmetic.

- Calculation of s_{ii}: It is essential to the correct determination of private key shares that each teller compute the evaluation of its polynomial at its own index (s_{ii}). Although this value is never transmitted nor validated by other tellers, it is used in the production of the private key share.

- Public key dissemination: The public key that each teller calculates privately at the conclusion of the key generation protocol must be agreed upon and stored. This is done via quorum. Since the system is guaranteed to have only t failed tellers, then any public key value agreed upon by at least $t + 1$ tellers is guaranteed to include a vote from a correct teller and therefore be the correct value. Each teller therefore posts its vote for the public key. All consumers collect these votes for the public key and select the one that receives that quorum of votes.

3.2 Testing

Due to constraints (mostly imposed by Jif itself) on the ability to call individual functions provided by CryptoFactoryC, we conducted unit testing of the non-channel code with a separate testing method implemented in TTProtocol.jif. This method simulates the execution of the entire key generation protocol, including reconstruction and the encrytpion/decryption of a trivial message from the perspective of all participants. This test exercises all of the CryptoFactoryC helper methods. But this test does not provide coverage of the message classes, nor does it directly test the key generation protocol method. For the former, these classes are trivial and readily tested via the channel tests described next. For the latter, we chose to test key generation in a non-distributed way because of the difficulty of engineering a distributed test. Integration tests, conducted via the existing experiment framework, provide the assurance of the actual code path.

We tested both the private and public channels by using them in the old (i.e., non-threshold) Civitas key generation protocol. It was much easier to use this protocol than to create a separate stand-alone unit test, because the set-up of the channel requires a large number of Civitas components to be initialized. Since the key share commitment phase of the old protocol uses hashed and signed messages, it was an ideal test case for the new public channel that we built. To test the public channel, we modified Civitas's old key generation protocol to use the public channel instead of posting messages to the bulletin board. To test the private channel, we introduced a dummy message that was sent privately during the run of the key generation protocol. We aborted key generation if there was an error sending or receiving the private message; since all tellers are required to be present to generate the key in the old protocol, we could reliably detect these errors.

A Protocol Specification

In the following protocols, let $\mathcal{N}$ be the space of nonces and let $publish(m)$ denote publishing message m on the bulletin board BB.

El Gamal Threshold Parameters

$((p, q, g), t, h)$ - where (p, q, g) are the regular El Gamal parameters, $h \leftarrow \mathcal{G}$ is the member of El Gamal group $\mathcal{G}$, and t is the maximum number of tellers allowed to fail. It is assumed that $t < \frac{n}{2}$, where n is the total number of tellers, and that the adversaries can't compute $\log_g h$.

ALGORITHM: Send Public Message

```
Input:  $meta$ - meta string
        $from\_id$ - sender identifier
        $m$ - message to be sent
        $k_s$ - RSA private signing key
1. $r \leftarrow \mathcal{N}$
2. $s = \mathsf{Sign}_{\mathsf{RSA}}(hash(meta, from\_id, m); k_s)$
3. $publish(meta, from\_id, m, s)$
```

ALGORITHM: Send Private Message

```
Input:  $meta$ - meta string
        $from\_id$ - sender identifier
        $to\_id$ - recipient identifier
        $m$ - message to be sent
        $k_s$ - RSA private signing key
        $K_e$ - RSA public encryption key
1. $r \leftarrow \mathcal{N}$
2. $k \leftarrow \mathsf{Gen}_{\mathsf{AES}}(1^l)$, where $l$ is the the security parameter for AES
3. $c_k = \mathsf{Enc}_{\mathsf{RSA}}(k; K_e)$
4. $s = \mathsf{Sign}_{\mathsf{RSA}}(hash(meta, from\_id, to\_id, m); k_s)$
5. $c_m = \mathsf{Enc}_{\mathsf{AES}}(meta, from\_id, to\_id, m, s; k)$
6. $publish(meta, from\_id, to\_id, c_k, c_m)$
```

ALGORITHM: Receive Public Messages

```
Input:   $meta$ - meta string
         $from\_id$ - sender identifier
         $K$ - RSA public key of teller with id $from\_id$
Output:  $\mathsf{RecvPubMsg}(meta, from\_id)$
1. Output
```

$$\{(meta, from\_id, m) \,|\, (meta, from\_id, m, s) \in BB \\ \wedge \mathsf{Ver}_{\mathsf{RSA}}(hash(meta, from\_id, m), s; K)\}$$

ALGORITHM: Receive Private Messages

```
Input:   $meta$ - meta string
         $from\_id$ - sender identifier
         $to\_id$ - recipient identifier
         $k_d$ - RSA private decryption key
         $K$ - RSA public key of teller with id $from\_id$
Output:  $\mathsf{RecvPrivMsg}(meta, from\_id, to\_id, k_d)$
1. $S_M = \emptyset$
2. For each message $\{m : m = (meta, from\_id, to\_id, c_k, c_m) \in BB\}$:
    - $k = \mathsf{Dec}_{\mathsf{RSA}}(c_k; k_d)$
    - $(meta', from\_id', to\_id', M, s) := \mathsf{Dec}_{\mathsf{AES}}(c_m; k)$
    - if $\mathsf{Ver}_{\mathsf{RSA}}(hash(meta', from\_id', to\_id', M), s; K)$
      then $S_M := S_M \cup M$
3. Output $S_M$
```

PROTOCOL: Distributed Threshold El Gamal Key Generation

Due to:	Gennaro et al. [6]
Principals:	Tabulation tellers $\mathrm{TT}_1, ..., \mathrm{TT}_n$
Public Input:	Distributed Threshold El Gamal Parameters (p, q, g, h, t)
	Set of RSA public keys for each teller, $\{K_{TT_i} : 1 \leq i \leq n\}$
	eid - election id; $id_i, ..., id_n$ - teller ids
Private input (TT_i):	RSA private signing key ks_{TT_i}
	RSA private decryption key k_{TT_i}
Output:	Public key Y, public key shares y_i

1. TT_i: Generate polynomials and public and private commitments:
   - Let polynomial $f_i(z) = a_{i0} + a_{i1}z + ... + a_{it}z^t$, where $a_{ik} \leftarrow \mathbb{Z}_q$ for $k \in [0...t]$
   - Let $z_i = a_{i0} = f(0)$
   - Let polynomial $f'_i(z) = b_{i0} + b_{i1}z + ... + b_{it}z^t$, where $b_{ik} \leftarrow \mathbb{Z}_q$ for $k \in [0...t]$
   - $C_{ik} = g^{a_{ik}}h^{b_{ik}} \bmod p$ for $k \in [0...t]$
   - $s_{ij} = f_i(j)$, $s'_{ij} = f'_i(j) \bmod q$ for $j \in [1...n]$
2. TT_i: Send Commitments:
   - Send Public Message$(pub_cmt, i, (C_{i0}, ..., C_{it}), ks_{TT_i})$
   - Send Private Message$(priv_cmt, i, j, (s_{ij}, s'_{ij}), ks_{TT_i}, K_{TT_j})$
3. TT_i: Verify commitments:
   - Receive Public Messages(pub_cmt, j) for $j \in [1...n]$
   - Receive Private Messages$(priv_cmt, j, i, k_{TT_i})$ for $j \in [1...n]$
   - Verify that

$$g^{s_{ji}}h^{s'_{ji}} = \prod_{k=0}^{t}(C_{jk})^{i^k} \bmod p \text{ for } j \in [1...n]$$

   If the check fails issue a complaint from i against j:
   Send Public Message$(cmt_compl, i, (i, j), ks_{TT_i})$
4. TT_i: Retrieve complaints against teller i and send responses:
   - $CP =$ Receive Public Messages(cmt_compl, j) for $j \in [1...n]$
   - For every complaint from teller TT_j against teller TT_i, (j, i): Send Public Message$(cmt_compl_resp, i, ((i, j), s_{ij}, s'_{ij}), ks_{TT_i})$
   - For every complaint against TT_j, increment complaints counter $c_j := c_j + 1$

```
5. TT$_i$: Verify complaint responses and determine the set of qualified tellers:
    - $CR =$ Receive Public Messages$(cmt\_compl\_resp, j)$
      for $j \in [1...n]$
    - For every $j \in [1...n]$:
      If either:
        $(c_j > t)$, or
        $CR$ contains a response from teller $TT_j$ that fails the check
        in step 3,
      then $Disq := Disq \cup \{TT_j\}$
    - $Qual = \{TT_1, ..., TT_n\} \cap Disq$
6. TT$_i$: Compute private key share $x_i$ and publish public key share commitments:
    - $x_i = \sum_{j \in Qual} s_{ji} \mod q$
    - $A_{ik} = g^{a_{ik}} \mod p$ for $k \in [0...t]$
    - Send Public Message$(keyshare\_cmt, i, (A_{i1}, ..., A_{it}), ks_{TT_i})$
7. TT$_i$: Verify public key share commitments for every teller $j \in Qual$:
    - $CP =$ Receive Public Messages$(keyshare\_cmt, j)$
    - $Disq := \emptyset$
    - Verify that
          $g^{s_{ji}} = \prod_{k=0}^{t} (A_{jk})^{i^k} \mathrm{mod}\, p$
      If the check fails:
      Send Public Message$(keyshare\_cmt\_compl, i, ((i, j), s_{ji}, s'_{ji}), ks_{TT_i})$,
      where $s_{ji}, s'_{ji}$ satisfy the check at step 3.
8. TT$_i$: Verify complaints and update the set of disqualified tellers:
    - $CP =$ Receive Public Messages$(keyshare\_cmt\_compl, j)$
      for $j \in Qual$
    - For each teller $j$ that received a complaint in $CP$:
      Verify that the complaint is valid using the checks in steps 3
      and 7 (a valid complaint will satisfy the check in 3, but will
      fail the check in 7).
      If the complaint is valid:
        $Disq := Disq \cup \{TT_j\}$
        Send Public Message$(reconst\_cmt, i, ((j, i), s_{ji}, s'_{ji}), ks_{TT_i})$
```

9. TT$_i$: Collect missing commitments for the disqualified tellers, perform reconstruction of the missing public key shares:
   - $CP = \textsf{Receive Public Messages}(reconst\_cmt, j)$ for $j \in Disq$
   - For each teller $j \in Disq$:
     $z_j = \sum_{k \in Qual} s_{jk} \cdot \lambda_{k,Qual} \bmod q$,
     where $s_{jk}, s'_{jk} \in CP$ pass the check in step 3 and

$$\lambda_{k,Qual} = \prod_{l \in Qual \setminus \{k\}} \frac{l}{l-k}$$

10. TT$_i$: $y_i = g^{z_i} \bmod p$
    $Y = \prod_{j \in Qual} y_j \bmod p$

PROTOCOL: Distributed Threshold El Gamal Decryption

Due to:	Cramer et al. [2]
Principals:	TT_i - tabulation teller i
Public Input:	(p, q, g, h, t) - Distributed Threshold El Gamal Parameters
	Y - shared public key
	$c = (a, b)$ - ciphertext, where $a = g^r \bmod p$ and $b = M \cdot Y^r \bmod p$
	for plaintext M and public key y
	y_i - public key share
Private input(TT_i):	x_i - private key share of teller i
	$Qual$ - set of qualified tellers
Output:	$\textsf{ThreshDistDec}(c; X)$

1. TT_i: Publish share $a_i = a^{x_i} \bmod p$ and proof $\textsf{EqDlogs}(g, a, g^{x_i}, a_i)$
2. TT_i: Let $\Lambda = \{j : j \in Qual \wedge \textsf{EqDlogs}(g, a, g^{x_j}, a_j)\}$
3. TT_i: Abort unless $|\Lambda| \geq t + 1$
4. TT_i: Compute $A = \prod_{j \in \Lambda} a_j^{\lambda_{j,\Lambda}} \bmod p$, where $\lambda_{j,\Lambda} = \prod_{l \in \Lambda \setminus \{j\}} \frac{l}{l-j}$
5. TT_i: Output $M = \frac{b}{A} \bmod p$

References

[1] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *Proc. IEEE Symposium on Security and Privacy*, pages 354–368, May 2008.

[2] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *Proc. International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, pages 103–118, May 1997.

[3] Don Davis. Defective sign & encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML. In *Proc. General Track of USENIX Annual Technical Conference*, pages 65–78, June 2001.

[4] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 427–438, 1987.

[5] Rosario Gennaro. Personal communication, December 2008.

[6] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptology*, 20(1):51–83, 2007.

[7] Torben Pryds Pedersen. A threshold cryptosystem without a trusted party. In *Proc. Annual International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, pages 522–526, April 1991.

[8] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.