

# Practical RSA Threshold Decryption for Things That Think

Roel Peeters\*, Svetla Nikova, and Bart Preneel

KULeuven, ESAT/SCD/COSIC and IBBT  
Kasteelpark Arenberg 10, 3001 Heverlee, Belgium  
`{firstname.lastname}@esat.kuleuven.be`

**Abstract.** Progressing towards an era of ever more small devices with computational power and storage capabilities, a new kind of security approach is needed. We propose a practical scheme to achieve threshold security for Things That Think. The private data of the user remains protected as long as the number of corrupted devices is lower than the threshold. We describe a procedure for key distribution, key redistribution, encryption and decryption. To show that our solution is feasible, we have implemented a proof of concept.

**Keywords:** mobile devices, threshold decryption, RSA, key management

## 1 Introduction

People are carrying around more and more Things That Think and our society is progressing towards an era of an ubiquitous network society [3]. Devices like a mobile phone, laptop, built-in contact-less smart cards, not only have computational power, they can also store data. While losing your mobile phone or laptop is bad, losing the data on these devices is worse. When you forget your mobile phone on the train, you don't want the finder to have access to the phone numbers of your friends, your text messages or the pictures you took. If your laptop is stolen, data from the company you are working for, can be compromised. The CSI survey of 2007 [15] reports a loss, in comparison with the overall reported losses due to computer crime, of 5.79%, due to theft of mobile devices, and 6.79%, due to theft of the data contained by these mobile devices. Theft of data can be prevented by encrypting the data and only when needed, decrypting the content.

An approach based on a user having several Things That Think and the notion of secret sharing [16], was first proposed by Desmedt *et al.* [8]. Each device is in possession of the public key and a share of the corresponding private key. Data can be encrypted under the public key, which is known to every device,

---

\* Roel Peeters is funded by a research grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

we do not require any interaction. For decryption, interaction with other devices is needed, since each device only possesses a share of the private key.

Shoup [17] proposed an efficient threshold RSA signature scheme. Damgård and Koprowski [6] and Damgård and Dupont [7] proposed schemes that facilitate distributed key generation, but are less efficient than the one from Shoup. In order to build a practical system on Things That Think, there are certain constraints in time and energy consumption. Because of these performance requirements, we will build our procedures further upon the scheme of Shoup and we do not consider distributed key generation, our procedure for key distribution makes use of a trusted dealer.

We also propose a key redistribution protocol, based on the work of Wong, Wang and Wing [19]. Key redistribution is used to protect against adversaries and to add or remove devices.

This paper is organized as follows. First we describe the models and assumptions. Then we describe the procedures. In section 4 we make an analysis in terms of costs and discuss the implemented proof of concept. Conclusions are drawn in the last section.

## 2 Models and assumptions

First of all we need to distribute the key among all devices. This key will be used to encrypt and decrypt private data. The user has  $n$  devices, of which he will choose the threshold  $k$  (default  $k = \lfloor n/2 \rfloor + 1$ ). This threshold is the number of devices that need to cooperate in order to get a decryption of an encrypted file. By deploying a hybrid encryption scheme, the encrypted data remains on the device which encrypted the data. The symmetric key, used to encrypt the data, needs to be reconstructed at the time of decryption by a joint effort of the mobile devices. Key redistribution has no effect on the encryption or decryption procedures as the value of the key does not change, only the values of the keyshares change.

Since we are working in a wireless setting and all devices are nearby, we use the atomic broadcast communication model, meaning that all devices get all messages at the same time. To communicate in private to another device we need to construct a private channel. Private channels come with a cost: one encryption operation at the sender side and one decryption operation at the receiver side.

The considered adversary is mobile, active and computationally bounded. He is capable of eavesdropping all traffic over the network and injecting messages at will. Corrupted devices are under the control of the adversary, which has full access to these devices' memory and can send messages using these devices. We assume that at any point in time the number of corrupted devices,  $t$ , is less than  $k$  devices. If however the number of corrupted devices is less than  $k$  but bigger than  $n - k$ , the user might not be able to decrypt the data himself. If  $k$  is bigger than  $n/2$ , as for the default choice of  $k$ , this can not occur. Our approach protects against corrupted devices that are no longer part of set of

trusted devices. It does not protect against devices within this set. Encrypted data on such corrupted devices can still be decrypted.

Key distribution is the most critical phase in the process, we assume that during this phase there are no active adversaries and that the dealer is not corrupted.

### 3 Procedures

#### 3.1 Key distribution

We start from the most efficient threshold RSA signing protocol, as proposed by Shoup [17], and modify it in the following ways: we remove the requirement of safe primes, there is no need for the elements to be in  $Q_N$ , the subgroup squares in  $\mathbf{Z}_N^*$ , and calculation of the keyshares is no longer done in a subgroup. Each modification will now be discussed into more detail.

The first two modifications are interconnected. Shoup's protocol requires safe primes in order to make zero-knowledge proofs (in the decryption phase in order to prove that the partial decryption is correct). The RSA standard [1] states that  $d$  should be calculated as follows,  $de \equiv 1 \pmod{\lambda(N) = \text{lcm}(p-1, q-1)}$ . When using safe primes, we can calculate  $d = e^{-1} \pmod{m}$ , where  $m = p'q' = 2\lambda(N)$ . Calculating  $d$  is done modulo a number that has large factors. When using standard primes this can not be guaranteed. A consequence of calculating  $d$  as such, is that all calculations need to be done in the subgroup of squares  $Q_N$ . Removing the requirement of safe primes makes computation easier since they do not need to be done in the subgroup of squares. The disadvantage is that the zero-knowledge proofs now have a larger error probability, this is discussed in section 3.4.

Reduction of the keyshares can only be done in the key distribution procedure and not in the redistribution procedure, as reduction of keyshares requires the value of  $\lambda(N)$  to be known. Knowing  $\lambda(N)$  makes it trivial to compute  $d$ . Not doing the reduction of the keyshares makes the scheme no longer minimal, the size of each keyshare exceeds the size of the original secret. In the setting of a user with a limited amount of Things That Think, the increase in size of the keyshares is small in comparison to the keysize, this is discussed in more detail in section 4.1.

After key distribution the trusted dealer erases the private key  $d$  and all values that make it trivial to compute  $d$ , e.g.,  $p$ ,  $q$  and  $\lambda(N)$ . The dealer is not necessarily one of the devices, it could for instance be within a metal box (to construct a Faraday cage) that performs the key distribution in which the user puts the devices he intends to use in this threshold decryption setup.

When procedures and devices become efficient enough to meet the time and energy consumption constraints, a distributed key generation procedure [4, 5, 11] can replace this key distribution procedure. The main problem of distributed key generation is that for RSA we need to generate two large prime numbers. Since the probability to generate a random prime is inversely proportional to the

required bit length, and even quadratic inversely proportional for safe primes, a large number of rounds of communication and a large amount of computation is required. Not using a trusted dealer has the advantage that there is no longer a single point of attack.

**Key distribution:**

1. Trusted dealer generates two primes  $p$  and  $q$  of length  $h/2$ , calculates the RSA modulus  $N = pq$ ,  $\lambda(N) = lcm(p-1, q-1)$ , chooses an prime  $e > n$ , coprime with  $\lambda(N)$ , and computes the secret  $d$  such that  $de \equiv 1 \pmod{\lambda(N)}$ .
2. Trusted dealer constructs a polynomial of order  $k-1$ :  $f(x) = d + c_1x + \dots + c_{k-1}x^{k-1}$ . For each device  $1 \leq i \leq n$  compute  $d_i = f(i)$ , the secret share of that device. The secret share of each device is sent over a private channel. A random  $v \in \mathbf{Z}_N^*$  is chosen to make commitments,  $v_i = v^{d_i}$ , for  $1 \leq i \leq n$ , are the public verification keys. The public verification keys are broadcasted.

The dealing phase can be made verifiable, making use of the verification keys, used to verify the validity of a partial decryption. Verification prevents a malicious dealer from handing out incorrect shares, however it does not prevent a malicious dealer from keeping the secret key  $d$ . We briefly sketch how to make the key distribution verifiable, but we omit this in our procedure. We can make use of this result in key redistribution. The idea is very similar to the Pedersen verifiable secret sharing (VSS) [14], but instead of committing to the coefficients of the polynomial, we commit to the keyshares. Since verification is used within the context of RSA encryption, we do not commit to the private key, as the decryption can be verified by encrypting it with the public key.

$$\begin{aligned}
 v^{\Delta d} &= v^{\sum_{j \in S} \lambda_{0,j}^S d_j} \\
 v^{\Delta d} &= \prod_{j \in S} v_j^{\lambda_{0,j}^S} \\
 v^{\Delta} &= (v^{\Delta d})^e = \left( \prod_{j \in S} v_j^{\lambda_{0,j}^S} \right)^e \tag{1}
 \end{aligned}$$

Each device  $i$  also checks if  $v_i$  equals  $v^{d_i}$ . Procedures needed when either of the verifications fail, are outside of the scope of this paper.

### 3.2 Key redistribution

Key redistribution is essential for security as this is a means to revoke decryption rights of devices, e.g. for devices no longer in possession of the user. Because the mobile devices are mass market devices, hence more vulnerable to attacks, shares of the private key of each device will be updated after performing a number of decryptions. This is called proactive key updating [9, 10, 12]. While all the keyshares of the devices change, the private key and the corresponding public key remain the same. We do not need to construct new prime numbers, this

makes it possible to have a procedure without trusted dealer. Key redistribution done by a cooperation of all involved devices removes any single point of attack.

It is possible to redistribute the secret key from a  $(k, n)$  threshold sharing scheme to a  $(k', n')$  threshold sharing scheme, this was proposed by Wong, Wang and Wing [19]. The threshold and number of devices are  $k$  and  $n$  before key redistribution,  $k'$  and  $n'$  after key redistribution, respectively. The new threshold is decided by the user, the default value is  $k' = \lfloor n'/2 \rfloor + 1$ . Devices that do not participate in the key redistribution are no longer part of the set of trusted devices and can not be used anymore to decrypt data. Noack and Spitz [13] present another method for adding and removing devices. Adding a device is done by sending subshares to this device, and this device will send updates to the other devices. Removing a device is done by reconstructing the keyshare of this device and distributing this keyshare among the other devices. We will modify the first scheme, as this provides verifiability of the participants and the possibility to update keyshares without removing or adding a device.

Triggering key redistribution can be done without (automatically) or with (manually) consent of the user. Automatic redistribution is done after a certain number of decryptions is performed, it is triggered by a device whose local counter for decryptions is higher than this number. This is only possible when all devices are present, the threshold can not be altered. In this case  $n' = n$  and  $k' = k$ . Manual key redistribution is triggered at request of the user. Requests for key redistribution need to be authenticated. Typically manual key redistribution is done if not all  $n$  devices are present (at least  $k$  need to be present), if automatic key redistribution fails, to add or remove a device.

Key redistribution consists of two phases: in the first phase the existing keyshares are shared, in the second phase new keyshares are constructed. At least  $k$  devices need to share their keyshare, these devices are the contributing devices. All devices that receive shares of the old keyshares are participating devices. The number of contributing devices is  $n''$ , the number of participating devices is  $n'$ . Before the second phase, each participating device verifies that all his received subshares are correct subshares of the old keyshares of the contributing devices. In order to verify this we can not make use of the result of the previous paragraph as this would require the public exponents  $e_i$ , corresponding to the keyshares, to be known. This would expose multiples of  $\lambda(N)$  and hence would allow to factor  $N$ . Instead we use the technique proposed by Wong, Wang and Wing [19], based upon the Pedersen VSS [14].

In the second phase, each participating device combines the received subshares into his new keyshare and broadcasts a commitment to this keyshare. These commitments allow us to verify, by using the mechanism described in the previous paragraph, that all new keyshares are shares of the original private key. We now present this solution in more detail.

*Phase 1* Contributing devices construct polynomials of order  $k' - 1$  and compute subshares. Subshares are sent over private channels to all participating devices.

$$f_i(x) = d_i + c_{i,1}x + \dots + c_{i,k'-1}x^{k'-1} \quad d_{i,j} = f_i(j)$$

Commitments are made to the coefficients of the polynomial and broadcasted. Because we already have verification keys from the initial key distribution, we use  $v$  for the commitments and do not need to commit to the old keyshare which is already known to all devices.

$$v^{c_{i,1}}, v^{c_{i,2}}, \dots, v^{c_{i,k'-1}}$$

For each received subshare, we verify that this is a correct subshare of the sender's old keyshare.

$$\forall i \in S' : v^{d_{i,j}} \equiv v^{d_i} \prod_{b=1}^{k'-1} (v^{c_{i,b}})^{j^b}$$

If verification fails, an authenticated complaint is broadcasted against the device that sent this subshare. The user is notified and asked how to proceed, e.g., try again or restart the procedure but exclude the device, subject of a valid complaint, from contributing.

*Phase 2* After successfully completing the first phase, each participating device combines the received subshares into his new keyshare and commits to it.

$$\Delta d_i^{new} = \sum_{j \in S'} \lambda_{0,j}^S d_{i,j} \quad v_i^{new} = v^{d_i^{new}}$$

We verify if all new keyshares are shares of the original secret, by evaluating Eq. (1). If Eq. (1) does not hold, an authenticated complaint is broadcasted. At least all honest devices ( $\geq \max\{k, k'\}$ ) broadcast a complaint. If Eq. (1) does hold, then any number of corrupted devices can broadcast a complaint. The user is again notified if verification fails and asked how to proceed.

**Key redistribution:**

*Phase 1*

1. All contributing devices construct a polynomial of order  $k' - 1$ :  $f_i(x) = d_i + c_{i,1}x + \dots + c_{i,k'-1}x^{k'-1}$ . For each device  $1 \leq i \leq n'$  compute  $d_{i,j} = f_i(j)$ , subshares of his keyshare, and transmit to each device the corresponding subshare over a private channel. Commitments are made to the coefficients of the polynomial,  $v^{c_{i,1}}, v^{c_{i,2}}, \dots, v^{c_{i,k'-1}}$ , and broadcasted.
2. All devices check, that each received subshare is a correct share of the old keyshare of that particular device:

$$\forall i \in S' : v^{d_{i,j}} \equiv v^{d_i} \prod_{b=1}^{k'-1} (v^{c_{i,b}})^{j^b}.$$

*Phase 2*

1. Each device constructs his new keyshare and commits to it.

$$\Delta d_i^{new} = \sum_{j \in S'} \lambda_{0,j}^S d_{i,j} \quad v_i^{new} = v_i^{d_i^{new}}.$$

2. All devices check if all new keyshares are shares of the original secret:

$$v^{\Delta^{new}} = \left( \prod_{j \in S} v_j^{new \lambda_{0,j}^S} \right)^e$$

### 3.3 Encryption

We deploy a hybrid encryption scheme, consisting of a public-key part and a secret-key part. The main motivation for this approach is to minimize the amount of communication in the decryption process, which requires interaction between the user's Things That Think. To encrypt a file, we do not interact with other devices. The encrypted file is only stored on the device that encrypts the file.

We use a *key encapsulation mechanism* (KEM) and a *data encapsulation mechanism* (DEM). As a KEM, we can use a public-key encryption scheme, generate a random bit string, and then encrypt it under the public key. The RSA cryptography standard [1] proposes *Optimal Asymmetric Encryption Padding* (OAEP) as a padding scheme for encryption. Shoup proposed OAEP+ [18], with a tighter security proof. RSA-KEM [2] provides a much better security reduction than RSA-OAEP+. This advantage becomes even more pronounced when one analyzes the security of many messages encrypted under a single public key. The security of RSA-KEM key encapsulation mechanism does not degrade at all as the number of ciphertexts increases. For the DEM we use a highly efficient method for encrypting data, e.g. AES.

RSA-KEM uses the public group key  $(e, N)$ . First a random number  $x$  is generated within the range  $[0, \dots, N]$ . The symmetric key  $K$  to encrypt the data is derived from the random number  $x$ , using a *key derivation function* (KDF),  $K = KDF(x, SymmetricKeyLength)$ . The second input for this KDF is the length of the symmetric key.

The file is encrypted under the symmetric key  $K$  using AES. The random number  $x$  is encrypted using the public group key  $(e, N)$ ,  $y = x^e \bmod N$  and appended to the encrypted file.

#### **Encryption:**

1. Generate a random number  $x$  from  $[0, \dots, N]$ .
2. Derive the symmetric key  $K = KDF(x, SymmetricKeyLength)$ .
3. Encrypt the file with key  $K$ .
4. Encrypt the random number with public group key  $y = x^e \bmod N$ , append  $y$  to the encrypted file.

### 3.4 Decryption

To decrypt an encrypted file we first need to derive the symmetric key used for the encryption. The random number used to derive the symmetric key is encrypted under the public group key  $(e, N)$ . This can be decrypted by joint effort of the user's mobile devices. Using the random number we derive the symmetric key and decrypt the encrypted file using this symmetric key.

A difference between signing data, as described in [6, 7, 17], and decrypting data, is that signing can be done publicly, while decrypting has to be done in private. The goal of signing data is authenticity, using data encryption provides confidentiality. The request for decryption needs to be authorized, the device requesting the decryption needs to be part of the set of trusted devices. To prevent other parties from decrypting, the partial decryptions need to be transmitted over a private channel to the device requesting the decryption. Each device within the set of trusted devices, also the corrupted devices, can decrypt all encrypted data in its memory.

*Requesting decryption* The device requesting the decryption needs to somehow authenticate itself as being part of the set of trusted devices. One way of proving that one is part of the set of trusted devices, is to prove that one knows its secret keyshare. When using this authentication mechanism the devices do not need to store public authentication keys from the other devices. Another advantage is that we already have a procedure to update the keyshares while a updating the public authentication key of a device is more difficult. Recall that when a device is permanently corrupted (e.g. sold, lost, stolen), this device is no longer part of the set of trusted devices after performing key redistribution. The private key has been redistributed and the device in question has not received a new keyshare. A device proves knowledge of the keyshare by performing a Schnorr signature  $(c, z)$  on the ciphertext we want to decrypt and the public encryption key of this device. This public encryption key is needed to set up the private channel. The  $L_1$  bit challenge  $c$  is created by using a cryptographic hash function  $H$ , where  $L_1$  is the secondary security parameter (e.g.,  $L_1 = 128$ ). Choose a random value  $r \in \{0, \dots, 2^{h+2L_1-1}\}$  and compute:

$$v' = v^r \quad c = H(v', y, \text{public encryption key}) \quad z = d_i c + r.$$

Each device verifies the Schnorr signature of the device requesting the decryption before partially decrypting the ciphertext:

$$v' = v^z v_i^{-c} \quad c = H(v', y, \text{public encryption key})$$

*Partial decryption* Each device uses his part of the private key to partially decrypt the ciphertext.  $x_i = y^{d_i} \bmod N$ . To validate the partial decryption, a proof of correctness is constructed. The proof of correctness proves in zero knowledge that  $\log_y(x_i) = \log_v(v_i)$ . The  $L_1$  bit challenge  $c_i$  is created by using a cryptographic hash function  $H$ . Choose a random value  $r \in \{0, \dots, 2^{h+2L_1-1}\}$  and compute:

$$v' = v^r \quad y' = y^r \quad c_i = H(v', y') \quad z_i = d_i c_i + r.$$

The proof of correctness consists of  $(c_i, z_i)$ . The proof of correctness is verified by computing  $v'$  and  $y'$ , using the public verification keys, and checking  $c_i$ :

$$v' = v^{z_i} v_i^{-c_i} \quad y' = y^{z_i} x_i^{-c_i} \quad c_i = H(v', y')$$

The partially decrypted ciphertext  $x_i$  from each device should only be shared with the device requesting the decryption. To construct a private channel, the partial decrypted ciphertext is encrypted under the public encryption key of the device device that requests the decryption. The proof of correctness  $(c_i, z_i)$  can be transmitted in plain.

*Combination and decryption* In order to communicate the partial decryptions over a private channel, the partial decryption are encrypted. This means that the device requesting decryption first needs to decrypt the partial decryptions using his private encryption key. Now he can verify the partial decryptions and combine the valid partial decryptions into the decryption  $x$  of the ciphertext. From  $x$  the symmetric key  $K$  is reconstructed, using the same key derivation function as for encryption, and the encrypted file is decrypted.

Any subset  $S$  of  $k$  valid partial decryptions can be combined into a valid decryption.

$$w = \prod_{j \in S} x_j^{\lambda_{\delta, j}^S}$$

Now we have  $w^e = y^\Delta$ . Since  $\gcd(e, \Delta) = 1$ , the extended Euclidian algorithm can be used to find two integers  $a$  and  $b$ , such that  $\Delta a + eb = 1$ . The decrypted ciphertext is equal to  $w^a y^b$ . Since the error probability of the zero-knowledge proofs is no longer negligible, as discussed in section 3.1, the decryption is verified by encrypting it again under the public key and compare it to the ciphertext. The value of  $a$  and  $b$  are independent of the ciphertext and can be calculated at key distribution and key redistribution. In the latter case these values only need to be recalculated if  $\Delta$  changes, e.g., when a device is added or removed.

After using the decrypted data, e.g., getting a telephone number of one of your contacts, it is important to encrypt this data again and make sure the plaintext is no longer available in memory.

**Decryption:**

Extract the ciphertext  $y$  from the encrypted file.

*Requesting decryption*

1. Broadcast together with the ciphertext and the public encryption key of this device, the Schnorr signature  $(c, z)$  on the ciphertext and the public encryption key.

$$v' = v^r \quad c = H(v', y, \text{public encryption key}) \quad z = d_i c + r.$$

2. The Schnorr signature is verified by computing  $v'$ , using the public verification keys, and checking  $c$ :

$$v' = v^z v_i^{-c} \quad c = H(v', y, \text{public encryption key})$$

### *Partial decryptions*

1. Each device uses his part of the private key to partially decrypt the ciphertext and construct a proof of correctness. The partial decryption is sent over a private channel back to the device that requested the decryption. The proof of correctness  $(c_i, z_i)$  is broadcasted.

$$x_i = y^{d_i} \bmod N \quad v' = v^r \quad y' = y^r \quad c_i = H(v', y') \quad z_i = d_i c_i + r$$

### *Combination and decryption*

1. The device that requested the decryption verifies the proof of correctness.

$$v' = v^{z_i} v_i^{-c_i} \quad y' = y^{z_i} x_i^{-c_i} \quad c_i = H(v', y')$$

2. Any subset  $S$  of  $k$  valid partial decryptions can be combined into a valid decryption.

$$w = \prod_{j \in S} x_j^{\lambda_{0,j}^S} \quad x = w^a y^b$$

The decryption  $x$  is checked by encrypting it again and comparing this to the ciphertext  $y$ .

3. Derive the symmetric key  $K = KDF(x, \text{SymmetricKeyLength})$ , and decrypt the file using this key.

## 3.5 Other uses

**Backup** When a device is stolen or lost, access to the data is denied to the thief or finder, but also the legitimate user has no longer access to his data. This can be prevented by backing up the encrypted data. The encrypted data can be stored anywhere and only be reconstructed by a device in the set of trusted devices with the cooperation of at least  $(k - 1)$  other devices within this set.

**Transferring encrypted data** For transferring data to a device outside the set of trusted devices, this data does not need to be decrypted first. The data can be transmitted in the encrypted form to this device together with a partial decryption and a Schnorr signature on the ciphertext. The public encryption key of the device in question is used in the challenge for the Schnorr signature. The partial decryption and the Schnorr signature are encrypted under the public encryption key of this device. The device outside the set of trusted devices can initiate a decryption of the data as long as the keyshares are not updated.

## 4 Evaluation

For the numerical examples, we will use the following parameters: the number of devices  $n = 10$ , the threshold  $k = 6$ , the security parameter  $h = 1024$  and the

secondary security parameter  $L1 = 128$ . In key redistribution  $n''$  devices (at least  $k$ ) reshare their keyshares to the  $n'$  participating devices in the key distribution.  $k'$  is the new threshold after redistribution. For illustration purposes, we assume  $n'' = k$ ,  $n' = n$  and  $k' = k$ . We assume that in order to construct the private channel we need 1 modular exponentiation at the sender and 1 at the receiver. In Table 4.4 we give an overview of the required communication and computation for each device in the different procedures.

#### 4.1 Information rate

The keyshare size  $h'$  increases linear with the number of key redistributions and logarithmic with the threshold  $k$ . After key distribution the expected size of the keyshares is  $h_0$ , after the first key redistribution  $h_1$ , etc. Each key redistribution the size of the keyshares is expected to increase with  $\log_2(k)$  bit.

$$h_0 = h + \log_2(a) \quad \text{with } a = \frac{n^k - 2^{k-1}n}{2^{k+1}(n-2)}$$

$$\text{for } x \neq 0 \quad h_x = h + \log_2\left(\frac{(2a+1)k^{x+1} - (a+1)k^x - ak}{k-1}\right) \approx h + \log_2(a) + x \log_2(k)$$

$$\text{Information rate} = \frac{\text{size of secret}}{\text{size of share}} = \frac{h}{h'}$$

For the chosen values, after key distribution the expected size of the keyshares is 1034 bit, the information rate is 99%. After 100 key redistributions the expected size of the keyshares is 1294 bit, the information rate is 79%.

#### 4.2 Communication

##### Key distribution

The dealer transmits the keyshares over private channels, resulting in  $nh'$  bit. The public key and the verification data are broadcasted, this is bounded by  $(n+2)h$  bit. For the illustration values, the total communication is 2,76 kByte.

##### Key redistribution

*Phase 1* Each device that reshapes his keyshare sends the subshares over private channels to all participating devices,  $n'h'$  bit, and broadcasts commitments to the polynomial,  $k'h'$  bit. The total communication is  $n''(n'+k')h'$  bit. For the illustration values and the 100th key redistribution, the total communication is 15,16 kByte.

*Phase 2* Each participating device commits to his new keyshare, requiring  $h$  bit of communication for each device. This adds 1.25 kByte to the total communication.

### Decryption

The device requesting the decryption broadcasts the ciphertext to be decrypted, his public key and a Schnorr signature on the ciphertext, resulting in  $3h + 3L1$  bit. Each other device sends over a private channel to the device requesting the decryption, his partial decryption and corresponding proof of correctness, resulting in  $3h + 3L1$  bit. For the illustration values, the total communication is 4,21 kByte.

### 4.3 Computation

#### Key distribution

The dealer constructs an RSA key pair, the  $n$  keyshares and the  $n$  verification keys. The keyshares are sent over private channels to the devices. The computational effort for the dealer is dominated by the effort needed to construct two distinct large prime numbers, needed to construct the RSA key pair.

All the devices, except the dealer, receive their keyshares over a private channel, they need to perform 1 modular exponentiation.

#### Key redistribution

*Phase 1* Each contribution device constructs  $n'$  subshares,  $n'(2k' - 3)$  multiplications and  $n'(k' - 1)$  additions. The subshares are sent over private channels to the devices,  $n'$  modular exponentiations. To generate the verification data, we need  $k' - 1$  modular exponentiations. Each participating device performs  $n''k'$  modular exponentiations and  $n''(k' - 1)$  multiplications to decrypt and verify the received subshares.

*Phase 2* Each participating device constructs his new keyshare and commits to it, requiring 1 modular exponentiation,  $n'' + 1$  multiplications and  $n''$  additions. To verify if the new keyshares are shares of the original secret, we need an extra  $(n' + 2)$  modular exponentiations and  $n'$  multiplications.

#### Decryption

The device requesting the decryption calculates his partial decryption, 1 modular exponentiation, and a Schnorr signature on the ciphertext, 1 modular exponentiation, 1 multiplication and 1 hash function evaluation. All other devices first verify the Schnorr signature, requiring 2 modular exponentiations, 1 multiplication and 1 hash function evaluation. After successful verification they calculate their partial decryption and construct a proof of correctness, 3 modular exponentiations, 1 multiplication and 1 hash function evaluation. To construct a private channel 1 extra modular exponentiation is required at sender and at receiver. The device requesting decryption needs to verify at least  $k - 1$  partial decryptions,  $4k - 4$  modular exponentiations,  $2k - 2$  multiplications and  $k - 1$  hash function evaluation. In order to combine the partial decryptions and verify the decryption,  $k + 3$  modular exponentiations and  $k + 1$  multiplications are needed.

#### 4.4 Implementation

To show that our solution is feasible, we created a proof of concept. We implemented the procedures in Java, using the BigInteger library for the computations and the MulticastSocket library to model the broadcast channel. Each device is a process and all processes run on the same machine, a dual core 1.8 GHz processor and 2 GB of RAM. We encrypted short strings, this can be justified by the fact that we are using a hybrid encryption/decryption scheme and symmetric encryption/decryption is much more efficient. For the above values we found that, on average, the key distribution takes 5300 ms and generates 3.93 kByte of traffic. Key redistribution takes 9700 ms and generates 18.72 kByte of traffic. Encryption takes 40 ms. Decryption takes 2700 ms and generates 3.69 kByte of traffic. Timing and communication results are more or less constant, except for key distribution where timing is depending on how fast we can find two distinct large prime numbers.

**Table 1.** Evaluation in terms of computation and communication. The number of devices  $n = n' = 10$ , the threshold  $k = k' = 6$ , the number of contributing devices in key redistribution  $n'' = k = 6$ , the security parameter  $h = 1024$  and the secondary security parameter  $L1 = 128$ .

Procedure Device	Computational effort		Communication [kByte]
	mod. exponentiations	multiplications	
<b>Key distribution</b>			
Dealer	generate 2 large prime numbers		$n(h + h') + 2h$ 2,78
Other devices	1	-	-
<b>Key redistribution</b>			
Contributing devices	$n''k' + n' + 2$ 48	$n''k' + (2n' - 1)(k' - 1) + 1$ 132	$(n' + k')h' + h$ 2,175
Other devices	$n''k' + n' + 3$ 49	$n''k' + n' + 1$ 47	$h$ 0,125
<b>Decryption</b>			
Requesting device	$5k + 5$ 35	$3k$ 18	$3h + 3L1$ 0,42
Other devices	6	2	$3h + 3L1$ 0,42

## 5 Conclusion and future work

In order to have a fully autonomous threshold decryption scheme for Things That Think, we introduced four procedures. Key distribution is the first step. When the key is distributed, we can encrypt and decrypt private data on the mobile

devices. The key redistribution mechanism makes sure that the key material on these devices is updated at regular times and makes it possible to add or remove devices.

Constructing large prime numbers in a distributed manner results in too much communication and computational effort. For this reason a trusted dealer is required for the initial key distribution. Although a trusted dealer introduces a single point of failure, by requiring the erasure of all information relevant to reconstruct the private key without using the threshold setting after executing the key distribution, this single point of failure only exists for a brief moment in time.

Key redistribution is done for two reasons: mass market devices are more vulnerable to attacks and it allows the user to have a dynamic group of trusted devices. Key redistribution is done periodically or on request of the user, e.g. to add or to remove a device. This is done without a trusted dealer, hence we introduce no single point of failure. Since the key remains the same, encrypted data can still be decrypted.

A hybrid encryption scheme is deployed, the data is encrypted under a symmetric key, and this symmetric key is encrypted under the group key. In order to decrypt, at least the threshold number of devices need to cooperate to reconstruct the symmetric key that is used to decrypt the data. We provide a way to authenticate a request for decryption, making use of the verification data, that does not require the devices to store extra keys.

By implementing a proof of concept, we have shown that our approach is feasible. The time needed for an encryption is negligible. Decryption requires around 3 seconds and 4 kByte of communication. Key redistribution takes around 10 seconds and 20 kByte of communication.

Corrupted devices that are still part of the set of trusted devices, e.g. virus, can decrypt all encrypted data in its memory. A trivial solution for this would require the user to interact with at least the threshold of devices to approve this decryption. It remains an open question how to solve this in a user-friendly way.

## References

1. PKCS #1 v2.1: RSA Cryptography Standard. Technical report, RSA Laboratories, 2002.
2. SO/IEC 18033-2: Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric ciphers. Technical report, ISO, 2006.
3. Internet of Things in 2020. Technical report, Joint European Commission / EPoSS Expert Workshop, 2008.
4. J. Algesheimer, J. Camenisch, and V. Shoup. Efficient Computation Modulo a Shared Secret with Application to the Generation of Shared Safe-Prime Products. In *CRYPTO '02: Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology*, volume 2442 of *LNCS*, pages 417–432, London, UK, 2002. Springer-Verlag.
5. D. Boneh and M. Franklin. Efficient Generation of Shared RSA Keys. *Journal of the ACM*, 48(4):702–722, 2001. Extended abstract in proceedings of Crypto '97.

6. I. Damgård and M. Koprowski. Practical Threshold RSA Signatures without a Trusted Dealer. In *EUROCRYPT '01: Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques*, volume 2045 of *LNCS*, pages 152–165, London, UK, 2001. Springer-Verlag.
7. I. B. Damgård and K. Dupont. Efficient Threshold RSA Signatures with General Moduli and No Extra Assumptions. In *Public Key Cryptography - PKC 2005*, volume 3386 of *LNCS*, pages 346–361. Springer-Verlag, 2005.
8. Y. Desmedt, M. Burmester, R. Safavi-Naini, and H. Wang. Threshold Things That Think (T4): Security Requirements to Cope with Theft of Handheld/Handless Internet Devices. In *Symposium on Requirements Engineering for Information Security*, West Lafayette, Indiana, USA, 2001.
9. Y. Frankel, P. Gemmell, P. D. MacKenzie, and M. Yung. Optimal Resilience Proactive Public-Key Cryptosystems. In *IEEE Symposium on Foundations of Computer Science*, volume 1294 of *LNCS*, pages 384–393. Springer-Verlag, 1997.
10. Y. Frankel, P. Gemmell, P. D. MacKenzie, and M. Yung. Proactive RSA. In *CRYPTO '97: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, pages 440–454, London, UK, 1997. Springer-Verlag.
11. Y. Frankel, P. D. MacKenzie, and M. Yung. Robust Efficient Distributed RSA-key Generation. In *The Thirtieth Annual ACM Symposium on Theory of Computing - STOC '98*, pages 663–672. ACM Press, New York, 1998.
12. A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive Public Key and Signature Systems. In *ACM Conference on Computer and Communications Security*, pages 100–110, 1997.
13. A. Noack and S. Spitz. Dynamic Threshold Cryptosystem without Group Manager. Cryptology ePrint Archive, Report 2008/380, 2008. <http://eprint.iacr.org/>.
14. T. P. Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In J. Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91*, volume 576 of *LNCS*, pages 129–140. Springer-Verlag, 1992.
15. R. Richardson. CSI Computer Crime and Security Survey 2007, 2007.
16. A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
17. V. Shoup. Practical Threshold Signatures. In B. Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques*, volume 1807 of *LNCS*, pages 207–220, Bruges, Belgium, May 2000. Springer-Verlag.
18. V. Shoup. OAEP Reconsidered. *Lecture Notes in Computer Science*, 2139:239–259, 2001.
19. T. M. Wong, C. Wang, and J. M. Wing. Verifiable Secret Redistribution for Threshold Sharing Schemes. Technical Report CMU-CS-02-114, Carnegie Mellon University, 2002.