RSA Threshold Signature Scheme for 3 parties

Me

March 5, 2025

Abstract

Your abstract.

We here describe our method to do an efficient 2 out of 3 RSA threshold signature without trusted dealer. We chose to build an RSA modulus made of 4 prime numbers. This is inspired by the work of Damgard et als in [DMS15]. However, this work focus on having a 2 out of 2 threshold scheme, we adapted it to our setup. We start by doing a 2 out of 2 setup and then distribute securely and verifiably the secret of each generating parties into 3 additive shares, one for each.

This design choice halves the security parameter of our scheme. Nevertheless, by using 4096 RSA modulus, we have the desired 2048-bits RSA security. The advantage of choosing this method is that the DKG takes few seconds (to confirm, might be even less, depends on latency aswell) where any other TSS RSA (like [BF01, DM10, ADN06]) can take hours.

Scheme 0.1 $\pi_{ThSigRSA}$

This protocol allows to create shares of signing key and sign in a threshold manner in a 2 out of 3 setup. It also provides a verification algorithm and a key refresh algorithm. The details of each algortihm is desrcibe in the following figures. It is an adaptation of the 2 out of 2 4-prime RSA TSS from [DMS15].

1. DKG: same as Figure 0.2
2. Sign: same as Figure 0.3
3. Verify: same for any RSA signature, also depicted in Figure 0.4
4. KeyRefresh: same as Figure 0.5

Scheme 0.2 DKG

given 3 parties, this algorithm outputs shares and public parameters (N, N_1, N_2) to achieve RSA TSS where the RSA modulus is of size λ.

Players: Parties $P_1 \dots P_3$
Inputs: e, λ
Outputs: $\{< d_1 >, < d_2 >\}$

$\mathcal{P}_i$.DKG.Round1(e)-->$N_i, \{< d_i >^j\}_{j\in\mathbb{Z}_3}, \pi_{i,1}, \pi_{i,2}, \pi_{i,3}$

if i=1 or i=2 **then**
  Pick randomly 2 prime numbers in $[2^{\lambda/4-1} + 2^{\lambda/4-2}; 2^{\lambda/4} - 1]$: p_i, q_i
  Compute $N_i = p_i * q_i$
  Compute $d_i = e^{-1} \mod (p_i - 1)(q_i - 1)$
  Compute $< d_i > \leftarrow$ RepSec$(3, d_i)$
  Send $< d_i >^j$ to P_j
  Broadcast N_i
  Compute $c_i =$ SHA3(sid||N$_i$)
  Compute $(\pi_{i,1}, \pi_{i,2}, \pi_{i,3}) \leftarrow$ ProveCD$(c_i, N_i, < d_i >)$
  Broadcast $\pi_{i,1}, \pi_{i,2}, \pi_{i,3}$

$\mathcal{P}_i$.DKG.Round2$(\{N_1, N_2, \{\pi_{k,1}, \pi_{k,2}, \pi_{k,3}\}_{k\in\mathbb{Z}_2}, < d_1 >^i, < d_2 >^i)$-->$N$

Receive $< d_1 >^i$ and $< d_2 >^i$, if one contain 0, ABORT
if $i \neq 1$ **then**
  Compute $c_1 =$ SHA3$(sid||N_1)$
  Run VerifyCD$(c_1, \pi_{1,1}, \pi_{1,2}, \pi_{1,3}, < d_1 >^i, N_1, e)$
if $i \neq 2$ **then**
  Compute $c_2 =$ SHA3$(sid||N_2)$
  Run VerifyCD$(c_2, \pi_{2,1}, \pi_{2,2}, \pi_{2,3}, < d_2 >^i, N_2, e)$
Compute $N = N_1 * N_2$
Publish N

Scheme 0.3 Sign

given 3 parties and secret key setup, this algorithm produces an RSA Signature on a message m that has been encoded using the EMSA-PSS encoding with empty nonce ([(IE16]). Round 2 only needs to be done once by anyone and this entity does not need to know any secret information.

Players: at least 2 Parties
Inputs: $N_1, N_2, \{< d_k >^i\}_{k\in\mathbb{Z}_2}, m$
Outputs: σ

$\mathcal{P}_i$.Sign.Round1(m)-->$\{\sigma_{1,j}, \sigma_{2,j}\}_{j\in\mathbb{Z}_3}$

P_i computes $\sigma_{1,i-1} =$ SimpleRSASign$(N_1, < d_1 >^i_{i-1}, m)$
P_i computes $\sigma_{1,i+1} =$ SimpleRSASign$(N_1, < d_1 >^i_{i+1}, m)$
P_i computes $\sigma_{2,i-1} =$ SimpleRSASign$(N_2, < d_2 >^i_{i-1}, m)$
P_i computes $\sigma_{2,i+1} =$ SimpleRSASign$(N_2, < d_2 >^i_{i+1}, m)$
P_i publishes $(\sigma_{1,i-1}, \sigma_{1,i+1}, \sigma_{2,i-1}, \sigma_{2,i+1})$

Agregator.Sign.Round2$(\{\sigma_{1,i}, \sigma_{2,i}\}_{i\in\mathbb{Z}_3})$-->$\sigma$

Compute $\sigma_1 = \prod_{i=1}^{3} \sigma_{1,i}$
Compute $\sigma_2 = \prod_{i=1}^{3} \sigma_{2,i}$
Compute $\sigma \leftarrow$ CRT$(\sigma_1, \sigma_2, N_1, N_2)$.
Publish σ

Scheme 0.4 RSAVerify

This algorithm verifies a RSA signature, it will verify RSA signatures that are generated by one or many parties without distinction as in the original RSA paper [RSA78].

Players: anyone with the public information
Inputs: m, σ, N, e
Outputs: *valid* or *invalid*

Verifier.RSAVerify(m, σ, N, e)-->*valid*/*invalid*
1: computes $\alpha = \sigma^e \mod N$
2: **if** $\alpha == H(m)$ **then**
return *valid*
3: **else**
return *invalid*

In the key refresh algorithm 0.5, each party will add a random to one of their share and subtract it to the other. Then by transmitting this random to the two other parties, they will be able to do the same operation on the correct share. This is done twice because we have some kind of double RSA setup with shares for both. A commitment is used to ensure that both parties receive the same random. In this protocol Commit and Open can be instantiated by any statistically hiding and computationally binding commitment. For efficiency reasons, we recommend using a salted hash commitment (ref specs) with hash function to be SHA3.

Scheme 0.5 KeyRefresh

This algorithm creates fresh shares from previous shares for a three party protocol. This allows to sequentially run multiple signature algorithms while having only one execution of the DKG.

Players: P_1, P_2, P_3
Inputs: $\{< d_j >\}_{j\in\mathbb{Z}_2}$
Outputs: $\{< d'_j >\}_{j\in\mathbb{Z}_2}$

$\mathcal{P}_k$.KeyRef.Round1$(\{< d_j >\}_{j\in\mathbb{Z}_2})$-->$A_{1,k}, A_{2,k}$
1: pick randomly $a_{1,k} \in [-2^{\log N+128}, 2^{\log N+128}]$ and $s_{1,k} \in \{0,1\}^*$
2: pick randomly $a_{2,k} \in [-2^{\log N+128}, 2^{\log N+128}]$ and $s_{2,k} \in \{0,1\}^*$
3: Compute $(A_{1,k}, s_{1,k}) \leftarrow$ Commit$(a_{1,k})$
4: Compute $(A_{2,k}, s_{2,k}) \leftarrow$ Commit$(a_{2,k})$
5: Broadcast $A_{1,k}, A_{2,k}$

$\mathcal{P}_k$.KeyRef.Round2$(A_{1,k+1}, A_{1,k-1}, A_{2,k+1}, A_{2,k-1})$-->$a_{1,k}, s_{1,k}, a_{2,k}, s_{2,k}$
1: Send $a_{1,k}, s_{1,k}$ to P_{k-1} and P_{k+1}
2: Send $a_{2,k}, s_{2,k}$ to P_{k-1} and P_{k+1}

$\mathcal{P}_k$.KeyRef.Round3$(a_{1,k-1}, a_{1,k+1}, a_{2,k-1}, a_{2,k+1})$-->$\{< d'_j >^i\}_{j\in\mathbb{Z}_2}$
1: If any of the share is equal to 0, ABORT
2: Run Open$(a_{1,k-1}, A_{1,k-1}, s_{1,k-1})$
3: Run Open$(a_{1,k+1}, A_{1,k+1}, s_{1,k+1})$
4: Run Open$(a_{2,k-1}, A_{2,k-1}, s_{2,k-1})$
5: Run Open$(a_{2,k+1}, A_{2,k+1}, s_{2,k+1})$
6: if one check fails, ABORT
7: Compute $< d'_1 >^i_{k-1} =< d_1 >^i_{k-1} + a_{1,k+1} - a_{1,k}$
8: Compute $< d'_1 >^i_{k+1} =< d_1 >^i_{k+1} - a_{1,k-1} + a_{1,k}$
9: Compute $< d'_2 >^i_{k-1} =< d_2 >^i_{k-1} + a_{2,k+1} - a_{2,k}$
10: Compute $< d'_2 >^i_{k+1} =< d_2 >^i_{k+1} - a_{2,k-1} + a_{2,k}$

Scheme 0.6 SimpleRSASign

This algorithm produces a RSA signature in a classical way with only one signer. This signature scheme has been published in [RSA78].

Inputs: m, d, N
Outputs: σ

$\mathcal{P}_i$.SimpleRSASign(N, d, m)--$\rightarrow\sigma$
1: computes $\sigma = H(m)^d \mod N$
return σ

Scheme 0.7 CRT

This algorithm constructs an element of $\mathbb{Z}/\mathbb{Z}_{N_1 * N_2}$ from an element of $\mathbb{Z}/\mathbb{Z}_{N_1} \times \mathbb{Z}/\mathbb{Z}_{N_2}$ where N_1 and N_2 are coprime.

Inputs: $a \in \mathbb{Z}/\mathbb{Z}_{N_1}, b \in \mathbb{Z}/\mathbb{Z}_{N_2}, N_1, N_2$
Outputs: $c \in \mathbb{Z}/\mathbb{Z}_{N_1 * N_2}$

$\mathcal{P}_i$.CRT(a, b, N_1, N_2)--$\rightarrow c$
1: computes r_1 and s_1 with extended GCD such that $r_1 \cdot N_1 + s_1 \cdot N_2 = 1$
2: computes $e_1 = s_1 \cdot N_2$
3: computes $e_2 = r_1 \cdot N_1$
4: computes $c = a \cdot e_1 + b \cdot e_2$
return c

Scheme 0.8 RepSec(d)

This algorithm constructs replicated secret for 3 parties for a 2 out of 3 TSS.

Inputs: d
Outputs: $< d >$

$\mathcal{P}_i$.RepSec(d)--$\rightarrow d_1, d_2, d_3$
1: P_i picks randomly $d_1, d_2 \in [-2^{\log N + 128}, 2^{\log N + 128}]$
2: P_i computes $d_3 = d - d_1 - d_2$
3: Set $< d >^1 = (< d >^1_2, < d >^1_3) = (d_2, d_3)$
4: Set $< d >^2 = (< d >^2_3, < d >^2_1) = (d_3, d_1)$
5: Set $< d >^3 = (< d >^3_1, < d >^3_2) = (d_1, d_2)$
return $< d > = (< d >^1, < d >^2, < d >^3)$

Scheme 0.9 CorrectDistribution

This set of algorithms proves for a three party protocol, that the shares sum to the inverse of e mod N_j. The first algorithm computes a proof, the second one verify the proof. $\mathsf{HashtoRing}_i$ is a hash function that map a 256 bit elements into a ring element in $\mathbb{Z}_{N_i}$

$\mathcal{P}_i.\mathsf{ProveCD}((c, N_i, < d_i >)) \dashrightarrow \pi_{i,1}, \pi_{i,2}, \pi_{i,3}$

1: Compute $\pi_{i,1} = \mathsf{SimpleRSASign}(< d_i >_1^2, N_i, \mathsf{HashtoRing}_i(c))$
2: Compute $\pi_{i,2} = \mathsf{SimpleRSASign}(< d_i >_2^1, N_i, \mathsf{HashtoRing}_i(c))$
3: Compute $\pi_{i,3} = \mathsf{SimpleRSASign}(< d_i >_3^2, N_i, \mathsf{HashtoRing}_i(c))$
return $(\pi_{i,1}, \pi_{i,2}, \pi_{i,3})$

$\mathcal{P}_i.\mathsf{VerifyCD}(c, \pi_{j,1}, \pi_{j,2}, \pi_{j,3}, < d_j >^i, N_j, e) \dashrightarrow valid$

1: P_i verifies $\pi_{j,i-1} == \mathsf{HashtoRing}_j(c)^{<d_j>_{i-1}^i} \mod N_j$
2: P_i verifies $\pi_{j,i+1} == \mathsf{HashtoRing}_j(c)^{<d_j>_{i+1}^i} \mod N_j$
3: P_i verifies that $\mathsf{RSAVerify}(c, \pi_{j,1} \cdot \pi_{j,2} \cdot \pi_{j,3}, N_j, e) == valid$
4: **if** one of the check fails **then**
5: ABORT
return *valid*

Figure 1: the ideal functionality for 4 prime RSA $\mathcal{F}_{ThSign4RSA}$

- **Key generation initiate**: having received the same message (KeyGen,sid) from all honest parties in the same round, parse sid as (S,ssid), fix e and generate 2 RSA keys pairs with respect to the security parameter $\|$. Get (d_1, p_1, q_1, N_1) and (d_2, p_2, q_2, N_2). Chose at random $d_{1,2}, d_{1,3}, d_{2,1}, d_{2,3}$ in $[-2N^2; 2N^2]$ and set $d_{1,1} = d_1 - d_{1,2} - d_{1,3}$ and $d_{2,2} = d_2 - d_{2,1} - d_{2,3}$. Then send (KeyGen, sid, (e, N_1, N_2)) to $\mathcal{Z}$.
- **Key generation finalize**: Upon reception of (KeyGenFinish, sid) from $\mathcal{Z}$, send (KeyGen, sid, $(e, N_1, N_2, d_{1,i-1}, d_{1,i+1})$) to each P_i.
- **Signature generation initiate** having received (Sign, sid, m) from all honest P_i, store (Sign, sid, m) and send it to $\mathcal{Z}$.
- **Signature generation finalize** Upon reception of a message (Signature, sid, m, σ_1, σ_2) from $\mathcal{Z}$, if (Sign, sid, m) is stored and (m, $\sigma_1, \sigma_2, 0$) is not recorded, delete a, entry (Sign, sid, m), record (m, $\sigma, 1$) and send (Signature, sid, m, σ) to all P_i.
- **Signature verification** Upon reception of a message (Verify, sid, m, σ,PK) from some party P_i, give this message to $\mathcal{Z}$. Upon reception of a message (Verified, sid, m, σ, Φ) from $\mathcal{Z}$, send (Verified, sid, m, σ, f) to P_i where f is determined as follow:
  1. If PK=(e, N) and (m, $\sigma, 1$) is recorded, then set f=1
  2. Else, if PK=(e, N) but no (m, $\sigma, 1$) is recorded, then set f=0. Record (m, σ, 0)
  3. Else set f=Φ
- **Key refreshment** Upon reception of a key refreshment query, send `KeyRefresh` to S and if not more than 1 party is corrupted/quarantined, erase all records of parties P_j recorded as `quarantine`, $\mathtt{P_j}$
- **Corruption**
  1. Upon reception of a message (`corrupt`,P_j) from S, record P_j is corrupted.
  2. Upon reception of a message (`decorrupt`, P_j) from S, if P_j has been recorded as corrupted or if not more than 1 party is corrupted/quarantined, record (`quarantine`, P_j).

Theorem 1. *Under the RSA unforgeability and in the presence of up to one malicious adversary,* $\pi_{ThSigRSA}$ *UC-realizes* $\mathcal{F}_{ThSign4RSA}$.

Proof. Our protocol $\pi_{ThSigRSA}$ is already very close from $\mathcal{F}_{ThSign4RSA}$. Thus, the description of the simulator S is trivial: the simulator will honestly perform all operations from the honest parties. In particular, all queries will be transferred from the simulator to the functionality.

The remaining difference from the ideal world and the real world is how a signature verification can go through without being recorded as a valid signature by the ideal functionality. If this event occurs, this means that the adversary is able to forge a signature on a fresh message that can be verified using the RSA verification algorithm.

From there we can prove theorem 1 by contradiction. Let's assume $\pi_{ThSigRSA}$ doesn't UC-realizes $\mathcal{F}_{ThSign4RSA}$. With the previous argument it implies that there exists an adversary A that can forge a signature in our protocol. We now show how A can be used to forge and RSA signature:

Assuming a challenger C for the unforgeability game for the RSA signature scheme, C will provide a challenge (e, N') and expects a forgery on a fresh message m. C also provides an access to a signing oracle $OSign$.

Now the simulator S will tweak the realization of $\pi_{ThSigRSA}$ to make A forgery useful to answer C challenge: instead of computing a p_i, q_i, N_i, d_i for the honest parties, C picks randomly i such that P_i is not corrupted. Since P_1 and P_2 have symmetrical roles, let's assume $i = 2$. Now S will perform its interaction as followed:

1. picks randomly $d_{2,2}, d_{2,3}$
2. get a challenge c
3. get $\alpha \leftarrow OSign(c)$
4. set $\pi_{2,1} = \alpha \cdot H(c)^{-d_{2,2}-d_{2,3}}$, $\pi_{2,2} = H(c)^{d_{2,2}}$, $\pi_{2,3} = H(c)^{d_{2,3}}$
5. sends $d_{2,2}, d_{2,3}$ to P_1 (if corrupted)
6. broadcast $N', \pi_{2,1}, \pi_{2,2}, \pi_{2,3}$

We can see that S behavior is indistinguishable from an honest party. If A corrupts P_2 after this first round abort. Else continue.

The probability of abort here is $1/2$ since the adversary has no information on which party use N' and thus doesn't know the inverse of $e \mod N'$. The protocol can be run any number of time, To sign a message, the simulator will use the signing oracle from C. The protocol continues and a signature σ on a fresh message m my be output by A.

Now S do the following: compute $\sigma' = \sigma \mod N'$. σ' is sent to C as the answer to the unforgeability game. σ' is a valid signature because by construction we have $\sigma'^e \mod N' = H(m)$.

□

Completeness:

Theorem 2. *Under the random oracle model and the unforgeability of the RSA signature, DKG is complete.*

Proof. We can assume w.l.o.g. that P_1 is corrupted. Using the property of the random oracle model, we can parametrize the non interactive challenge to be a fresh message to sign to answer the challenger C of the RSA unforgeability. Thus $(\pi_{1,1} \cdot \pi_{1,2} \cdot \pi_{1,3})^e = H(c) \mod N_1$ is a valid signature on c and thus on a fresh message m. by verifying that the shares lie in $\pi_{i,j}$ we are assured that the sum of the shares is an exponent that allows to sign a message e.g. the sum of the shares is d.

Key Refresh: In the event of a key refresh where one sender is corrupted, the simulator will execute the protocol honestly. It will also verify that the shares sent to both honest parties are the same. If this is untrue we have two cases:

1. one of the commitment doesn't open to the share, in that case report missbehavior
2. if both share are opening of the commitment, this defies the binding property of the commitment (in our implementation using SHA3, it means that the corrupted parties is able to find a collision which is unlikely to happen).

□

Theorem 3. *The key refresh algorithm presented in 0.5 is zero knowledge and sound under the hiding and binding property of the commitment scheme and the semantic security of the encryption scheme used to create the secure channel.*

Proof. There are two entry point for an adversary to get some knowledge of the shares of 0:

- get some knowledge of the shares when they are sent through secure channel, this would directly mean that the semantic security of the underlying encryption scheme that has been used for creating the secure channel is not holding.
- get some knowledge of the shares from the publicly known commitment. This means that the adversary here is a good candidate for breaking the hiding property of the commitment scheme.

The soundness of the Key Resfresh is ensured by the binding property of the commitment scheme: the binding property make sure that both recipient of a share of 0 receives the same value, since one add it to its share and the other one is subtracting to it, the sum is the same and thus new shares and old shares are additive shares of the same integer. □

References

[ADN06] Jesús F Almansa, Ivan Damgård, and Jesper Buus Nielsen. Simplified threshold rsa with adaptive and proactive security. In *Advances in Cryptology-EUROCRYPT 2006: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28-June 1, 2006. Proceedings 25*, pages 593–611. Springer, 2006.

[BF01] Dan Boneh and Matthew Franklin. Efficient generation of shared rsa keys. *Journal of the ACM (JACM)*, 48(4):702–722, 2001.

[DM10] Ivan Damgård and Gert Læssøe Mikkelsen. Efficient, robust and constant-round distributed rsa key generation. In *Theory of Cryptography Conference*, pages 183–200. Springer, 2010.

[DMS15] Ivan Damgård, Gert Læssøe Mikkelsen, and Tue Skeltved. On the security of distributed multiprime rsa. In *Information Security and Cryptology-ICISC 2014: 17th International Conference, Seoul, South Korea, December 3-5, 2014, Revised Selected Papers 17*, pages 18–33. Springer, 2015.

[(IE16] Internet Engineering Task Force (IETF). Pkcs 1: Rsa cryptography specifications version 2.2, 2016.

[RSA78] Designers Ron Rivest, Adi Shamir, and Leonard Adleman. Rsa (cryptosystem). *Arithmetic Algorithms And Applications*, page 19, 1978.