Robust distributed symmetric-key encryption

Xunhua Wang and Ben Huson
wangxx@jmu.edu,husonjb@dukes.jmu.edu

August 18, 2020

Abstract

In distributed symmetric-key encryption (DiSE), a set of n distributed servers share a key (or key set) and any $t, t \leq n$, servers can collectively use the shared key (or key set) in a DiSE transaction to encrypt a message or decrypt a ciphertext without reconstructing the shared key (or key set). Each participating server contributes one or more partial results and one participating server called the initiator combines all partial results into a final result. An adversary who has compromised up to $(t-1)$ DiSE servers will *not* be able to access the shared key (or key set).

Due to the distributed nature of DiSE, a DiSE server that has been compromised by an adversary may return wrong partial results to the initiator. Worse, multiple DiSE servers compromised by the same adversary may collude to send back wrong partial results. In this article we developed a robust DiSE that allows an honest initiator to detect wrong partial results by an adversary. The robustness of our DiSE is built through redundant computation. Our robust DiSE can detect wrong partial results by an adversary who has compromised up to $min(t-1, n-t)$ servers. Next, the honest-initiator assumption is removed by rotating the initiator role among active servers across multiple DiSE transactions. A scalable, industry-level implementation for the robust DiSE has been developed and two cases, $(t = 3, n = 5)$ and $(t = 16, n = 24)$, have been tested to show the feasibility of robust DiSE. Our robust DiSE can be used to build intrusion-tolerant applications, such as intrusion-tolerant database encryption.

Keywords: Intrusion tolerance, robust computing, distributed symmetric-key encryption (DiSE), robust DiSE, threshold cryptography, distributed pseudo-random function

1 Introduction

Long-term secrets that must stay online for use over an extended period of time always deserve extra protection, as such secrets may attract persistent attacks from remote adversaries and thus have a high chance to be compromised. Such online secret could be one column of a database table, the private key of a long-term public/private key pair (such as RSA [35]), or a long-term symmetric key (such as an AES key [32]). When the database table column in the first case is encrypted with a cryptographic key, the security problem is reduced one of the latter cases.

Several security measures have been developed to mitigate the risks of online long-term cryptographic keys and each has its appropriate applications. In one approach, the private key of a public/private key pair is stored on a tamper-resistant smart card or a secure coprocessor [41]. It is used on the hardware and never leaves the hardware. It is believed to be very hard for an adversary to retrieve the private key without physically breaking the hardware. Since the computational power of such hardware is often limited compared to a general PC, this approach has its limits in the number of decryptions per second it can handle. For applications that require quick and frequent decryption, this approach might not be appropriate.

Another approach to protect a long-term online secret is to share the secret among multiple (say n; for example $n = 5$) powerful servers, which are also called *parties* and each party is assigned one or more *shares* of the secret. A *threshold* number (say $t, t \leq n$; for example, $t = 3$) of these servers can collectively use the shared secret in a function without actually reconstructing the shared secret. In such a collective effort, each participating party uses its share to calculate a *partial result* and any of them can combine all partial results into a final result. Since not all parties are needed in the shared function evaluation, this approach tolerates attacks and can still work even some parties have been compromised. Such shared function can be a decryption or digital signing and they are called *threshold decryption* and *threshold signing* respectively. Together they are called *threshold cryptography* [13, 15, 16, 14, 12].

Threshold cryptography differs from threshold secret sharing [36, 5] in that the shared secret is never reconstructed in its use. Several public-key functions, such as RSA decryption, RSA digital signing, and DSA digital signing, naturally lend themselves to the threshold structure very well. Multiple threshold cryptographic schemes based on public-key cryptography have been developed, including *threshold RSA* [22, 24, 33, 37] and *threshold DSA* [23, 25]. In threshold public-key cryptography, some additional desirable properties can also be achieved. For example, in threshold RSA and threshold DSA, each participant is able to provide an auxiliary value along with its partial result; other participants can use the auxiliary value to check the correctness of the corresponding partial result and thus can detect the malicious behavior by a participant [34, 24, 10].

In contrast, symmetric-key encryption schemes such as AES do not support function sharing naturally and the research on threshold symmetric-key encryption started much later. Probably due to this status, in many cases, the term of threshold cryptography often implicitly refers to threshold public-key cryptography only. Naor et al. [31] developed distributed *pseudo-random functions (PRF)*, which allow a threshold number of distributed parties to collectively evaluate a function without reconstructing the shared secret. Three types of threshold pseudo-random functions are developed in [31], including *bivariate polynomials*, *weak pseudo-random functions based on the decisional Diffie-Hellman (DDH) assumption* [6, 17], and *any pseudo-random function*. The first only works for bivariate polynomials, the second type can be used with AES (the shared function itself must be DDH but the output can be used as an AES key), while in the last type the shared function can be instantiated with any pseudo-random functions, including AES. Since AES-128 is roughly 1000 times faster than DH-3072, the third *distributed* PRF has performance advantages and can be used for *distributed/threshold symmetric-key encryption (DiSE)*.

In this distributed PRF, for n parties/servers with identities $i, 1 \leq i \leq n$ and threshold $t, t \leq n$, the shared secret is not just one key but a set of $u = \binom{n}{t-1}$ different symmetric keys $(k_1, k_2, \ldots, k_u)$, which are randomly generated by a trusted dealer; $\binom{n}{t-1}$ is the binomial coefficient.

$k_i, i \le i \le u$, will then be assigned to all members of each distinct $(n-(t-1))$-subset/combination of $\{1, 2, \ldots, n\}$ [27, page 8] and the dealer will then forget all k_i. As a result, each party will be assigned $\binom{n-1}{t-1}$ keys. This secret sharing scheme will be called the *t-n combinatorial secret sharing* hereafter. Let the shared PRF function be f (for example, f can be AES). For a given input x, any t parties will be able to collectively calculate $F(x) = w = (w_1 \oplus w_2 \oplus \ldots \oplus w_u)$, where $\oplus$ denotes exclusive-OR. In this collective computing each party j calculates a partial result $w_j = f_{k_j}(x)$ and any of them can act as the *combiner* to combine all w_j into w. Throughout this paper, F will be used to denote this distributed PRF scheme.

Agrawal et al. [1, 2] observed that for applications of distributed symmetric-key encryption, such as key distribution center (KDC) for enterprise network authentication, encryption with a block cipher in a distributed manner alone does *not* offer enough security, as an adversary may mount data integrity-related attacks. One such insecure example is the direct application of the distributed PRF F in the following way, which is called F-DiSE-0 hereafter: to encrypt message m, an entity first generates a random key w and uses w to encrypt m into $c_1 = E_w(m)$, where E is AES encryption; next, the entity asks t distributed parties to collectively calculate $y = F(j||c_1)$, where j is the identity of one participating server who acts as an initiator; next, the entity calculates $c_2 = y \oplus w$, and then uses (j, c_1, c_2) as the ultimate ciphertext. Note that c_1 is the encryption of m by w while c_2 is the encryption of w by the shared key set. An active adversary who has obtained access to the ciphertext can modify c_2 to a different value c_2'. Accordingly, in the later decryption of (j, c_1, c_2'), the same value y will be collectively calculated first; however, a different w will be recovered as $w' = c_2' \oplus y$ and as a result, a different plaintext will be recovered $m' = D_{w'}(c_1)$, where D is AES decryption. This modification attack will not be detected in decryption.

Agrawal et al. [1, 2] argued that a better building block for distributed symmetric-key encryption is *authenticated encryption* such as CCM and GCM [4, 18, 19], which provides both data confidentiality and integrity simultaneously. Agrawal et al. [1, 2] developed the following DiSE scheme based on F [31], which is called F-DiSE-1 hereafter: There are n distributed parties/servers with identity $i, 1 \le i \le n$, and all pairwise communication between them is performed over secure channels (such as TLS with mutual authentication through pre-configured public keys or digital certificates). As described earlier about F, with the *t-n* combinatorial secret sharing scheme, in the setup stage, $u = \binom{n}{t-1}$ random symmetric keys $(k_1, k_2, \ldots, k_u)$ will be generated and assigned to these n servers.

Let h be a cryptographic hash function and $||$ denote string concatenation. Upon receiving a request from a *DiSE client*, any one of these DiSE servers (say server j) can act as an *initiator*, called *encryptor* in [1], to invite other $(t-1)$ server to distributively encrypt a message m into c, as follows. Server j first generates a long random number ρ and calculates $\alpha = h(m||\rho)$. Next, server j chooses $(t-1)$ other active servers, sends $j||\alpha$ to them securely, and asks them to return partial results so that j can combine then into $w = F(j||\alpha)$. Server j assembles the final ciphertext c as $c = (c_1, j, \alpha)$, where $c_1 = \text{PRG}(w) \oplus (m||\rho)$ and PRG is a pseudo-random number generator such as Hash_DRBG and HMAC_DRBG [3].

Later, server i may agree to act as an initiator to decrypt a received or stored ciphertext $c' = (c_1', j', \alpha')$, which may or may not be c due to possible modification attacks. Server i will choose $(t-1)$ other active servers and together they will collectively calculate $w' = F(j'||\alpha')$. In the end, server i will compute $c_1' \oplus \text{PRG}(w')$ and parse the result into $(m'||\rho')$. Server i will check the validity of the plaintext by checking whether $\alpha' \stackrel{?}{=} h(m'||\rho')$. If they do not match, ciphertext c' has been

tampered with and will be rejected. (Both the collective encryption and collective decryption are called a *DiSE transaction*.)

It is worth noting that in this DiSE scheme, authenticated encryption is not achieved through existing authenticated encryption schemes such as CCM or GCM, but through a combination of stream encryption (as $\text{PRG}(w) \oplus (m||\rho)$) and a commitment through cryptographic hashing (in $\alpha = h(m||\rho)$). In this design, the computing of partial results w_i is highly parallelizable among participating servers.

Also, the collective computing in DiSE is essentially a distributed stream cipher, where the same keystream w is used to encrypt $(m||\rho)$ and to decrypt c_1 later. Consequently, when f is instantiated with AES, only AES encryption will be used and AES decryption is not needed. In a DiSE transaction, a participating server receives $(j||\alpha)$ and returns one or more partial results w_i. A participating server may not even know whether it is participating in an encryption or decryption.

`ADDITIONAL DISTRIBUTED ATTACKS:` In a distributed environment like DiSE, several attacks beyond message privacy and ciphertext integrity are likely to happen. In DiSE, a server that has been compromised by an adversary may misbehave and send back wrong partial results w_i. Due to the nature of F, this malicious behavior will not be detected by the initiator server j until when c is decrypted. For applications where decryption may happen at a much later time than encryption, such misbehavior will not be detected over an extended period of time and may cause severe consequences.

Also, in DiSE, a participating server can verify the identity of the initiator but has to blindly trust an authenticated initiator. Consequently, a DiSE server that has been compromised by an adversary may steal all stored ciphertext and act as an initiator to use other unwitting servers as a decryption oracle to decrypt all ciphertext.

`OUR CONTRIBUTION:` In this research, we developed a *robust* DiSE scheme, called *F-DiSE-R*, which allows an honest initiator to detect wrong partial results from DiSE servers that have been compromised and controlled by an adversary. Our robustness construction is based on *redundant computation* from additional honest participating servers and the redundant values allow an initiator to detect wrong partial results from malicious servers. Our robust DiSE works even in the presence of an adversary that controls multiple DiSE servers. A δ-adversary is an attacker that has controlled $\delta, \delta < t$, DiSE servers and these servers may collude in an attack. A DiSE is δ-robust if an initiator can detect malicious behaviors of a δ-adversary.

We observed that when $t = n$, there is no redundancy-based robust DiSE. When $t < n$, for any $\delta \leq \min(t-1, n-t)$, there is a δ-robust DiSE and we developed algorithms for an initiator to select robust subset of servers for robust collective computing, in which any malicious behaviors of servers controlled by a δ-adversary will be detected. When more active servers are available, an honest initiator in a robust DiSE scheme can even correct wrong partial results from a δ-adversary.

Next, in our robust DiSE, the honest-initiator assumption is removed by rotating the roles of initiator among all active DiSE servers across multiple DiSE transactions. Any attempt to overly use a compromised server as initiator will be statistically detected by other DiSE servers and that server can be blacklisted.

In this research, we also developed an industry-level robust DiSE implementation that is very

scalable and explored methods to speed up distributed computing in robust DiSE.

The remainder of this article is organized as follows. In Section 2, we review priori work related to this research. Section 3 gives the details of our robust DiSE scheme, including methods to detect and correct a δ-adversary. In Section 4 we give the details of an implementation for our robust DiSE scheme and some methods to speed up computations. Section 5 discusses several applications of our DiSE constructions. A summary of this research is given in Section 6.

2 Related work

Threshold cryptography has been studied, much more extensively and much earlier in the public-key setting [13, 11, 7] than in the symmetric-key setting [31, 8, 29].

In the public-key setting, additional threshold cryptography properties have been developed, including *robust* decryption and digital signing [22, 24, 37, 23, 25], *proactive* update of existing shares without changing the shared secret [26, 9], and threshold key generation *without* a trusted dealer [20, 28, 21]. These properties have not been well studied in the symmetric-key setting and methods in the public-key setting cannot be directly extended to the symmetric-key setting, due to the different nature of symmetric ciphers. Moreover, distributed symmetric-key cryptography faces new problems that are not much an issue for its public-key counterpart, due to its nature and the speed gap, roughly 1000 times, between public-key operations and symmetric-key operations.

2.1 Block cipher sharing through composition

An approach different from [31] to share a symmetric-key block cipher among multiple parties was proposed by Brickell et al. [8], where the *composition* of a block cipher with a sequence of symmetric keys $k_1, k_2, \ldots, k_u$, is shared. This approach uses the same t-n combinatorial secret sharing scheme [27], in which a subset of $\binom{n-1}{t-1}$ keys will be assigned to each party. If the shared PRF function is f, for a given input x, any t parties will be able to collectively calculate $w = G(x) = f_{k_u}(f_{k_{u-1}}(\ldots(f_{k_2}(f_{k_1}(x)))))$. This cascading way of composition of a block cipher requires computations to be distributively performed in a sequential mode and thus may be less efficient than $w = F(x) = \oplus_{j=1}^{u}(f_{k_j}(x))$ [31, 29].

Both Brickell et al. [8] and Martin et al. [29] focused on message confidentiality through encryption only and they used symmetric-key encryption, not authenticated encryption, as building blocks. As a result, their DiSE schemes may be vulnerable to the ciphertext modification attack described in Section 1.

3 Robust DiSE

In this section, we shall investigate how to develop a *robust* DiSE in the sense that an *honest* initiator is capable of detecting wrong partial results returned from δ compromised/malicious servers controlled by a δ-adversary. These δ compromised servers may collude in returning incorrect

partial results to the initiator, causing the final result to be incorrect. This attack is more severe for DiSE encryption than decryption, as in DiSE encryption the initiator has no effective ways to check whether the resulting ciphertext is correct or not. In contrast, in an F-DiSE-1 decryption, the initiator can indirectly check the partial results from participating servers by checking the integrity of the decrypted plaintext.

Public-key decryption and digital signing algorithms such as RSA and DSA often have nice algebraic structures, which allow each non-initiator participating server to provide auxiliary data that can be used to check its partial result. In contrast, symmetric-key encryption algorithms such as AES do not have this kind of algebraic property and thus the auxiliary-data for robustness approach does *not* apply.

3.1 The role of the initiator in robustness

In F-DiSE-1 [1] described in Section 1, when a DiSE client wants to use the distributed symmetric encryption, it picks one DiSE server as the initiator, which will create a DiSE transaction request on behalf of the client and to combine all partial results into a final result. In this process, the initiator generates a random value ρ, calculates α, and sends it to selected servers for partial results. Such an initiator is naturally trusted more than other servers, as it generates and knows ρ, combines w_i into w, and may even encrypt m int c for the client.

The very existence of an initiator may pose a threat for robustness, as a malicious initiator may simply use a wrong w in a DiSE encryption. A DiSE client can choose to remove an initiator from a DiSE computation by assuming all initiator's responsibilities itself, including monitoring the online/offline status of servers, choosing participating servers for a transaction, sending requests to all participating servers, receiving and checking the correctness of partial results from participating servers, and combining partial results into a final result. Some of these responsibilities (such as monitoring the status of servers and establishing persistent connections to active ones) are infrastructure-oriented and require non-trivial resources. A DiSE client assuming these responsibilities will not be lightweight.

In this article, our robust DiSE construction still assumes an honest, but not a fixed initiator. That is, a client will randomly pick a DiSE server for the initiator role and this initiator role only lasts for that DiSE transaction or a specific period. The initiator is assumed to be honest on the said message(s)/period. This choice allows a DiSE client to be lightweight and free of heavy communications.

3.2 Robust DiSE through redundant computation

Our following robust DiSE is based on F-DiSE-1 [1] described in Section 1. The share generation and distribution steps (by the dealer) remain the same and the servers receive the same number of keys. Our constructions for robustness are on how multiple servers are chosen for a collective computation and on how partial results w_i are checked, which is missing in F-DiSE-1. When no malicious/incorrect w_i values are received by the initiator, the ultimate value of w will actually not change from F-DiSE-1.

In a robust DiSE, when an initiator selects other participating servers for a DiSE transaction, instead of picking just $(t-1)$ other servers, the initiator will choose more servers so that each w_i not by the initiator must be computed by at least two different participating servers and these values of the same w_i will be compared against each other to detect adversaries. For example, to detect an adversary who has compromised just one DiSE server, the participating servers must be chosen in a way that every partial result $w_i = \text{AES}_{k_i}(x)$ not by the initiator is calculated exactly twice by two different participating servers. By comparing the two inbound values w_i, the initiator can detect a fake w_i value from the 1-adversary.

This idea of redundant computation for robustness actually works very well with the t-n combinatorial secret sharing [27] used in F-DiSE-1 [1], as in this secret sharing each key k_i is assigned to $(n-(t-1))$ servers, which are more than one server when $t \leq (n-1)$.

We have the following results about robustness through redundancy.

Observation 1 *When $t = n$, there is no robust DiSE through redundancy.*

In the $(t = n)$ case of F-DiSE-1, every server is needed in a collective computation and no key/server is redundant. As a result, no server's partial result can be verified through redundant computing. This conclusion is very different from public-key-based threshold decryption/signing, where robustness is achievable even for $t = n$ through auxiliary data.

Observation 2 *When $t \leq (n-1)$, there is a robust DiSE scheme against 1-adversary and $(t+1)$ servers, including the initiator, are needed to participate.*

The above observation can be proved by contradiction. For *any* subset of $(t+1)$ servers, in the t-n combinatorial secret sharing scheme [27], every k_i must be assigned to at least two different servers in this subset. If otherwise is true, that is if a k_i is actually assigned to just one (or less) server in this subset, by removing that server from the $(t+1)$ subset, the remaining t servers in the subset will not be able to reconstruct the whole key set $\{k_1, k_2, \ldots, k_n\}$. This contradicts the very threshold property of the t-n secret sharing: any t participants should be able to reconstruct the shared key set.

Thus, for *any* such subset of $(t+1)$ servers, in our robust DiSE, for each k_i, two such servers will be designated to calculate partial result w_i, which allows the initiator to check against 1-adversary for robustness.

This result can be further generalized as follows.

Observation 3 *Let $\gamma = min(t-1, n-t)$. For any $\delta, 1 \leq \delta \leq \gamma$, there is a robust DiSE scheme against δ-adversary and $(t+\delta)$ servers are needed to participate in this robust DiSE.*

The proof will be similar: for any $(t+\delta)$ servers in the t-n combinatorial secret sharing scheme, any key k_i must have been assigned to at least $(\delta+1)$ different servers. In the robust DiSE, such $(\delta+1)$ servers will be chosen to compute partial result w_i and through comparing the $(\delta+1)$ w_i values, the initiator will be able to detect a δ-adversary.

γ must be smaller than t as any t servers can reconstruct the shared key set. γ must be smaller than $(n-t+1)$ because in the t-n combinatorial secret sharing, each key is assigned $(n-t+1)$ times (to $(n-t+1)$ different servers respectively).

3.3 Robust subset and the assignment of w_i

A subset of servers that allow an honest initiator to do robust computing is called a *robust subset*. In the above observations, *any* set of $(t+\delta)$ servers is a robust subset. This makes it easy for an initiator as it can pick any other $(t+\delta-1)$ servers. Still, the initiator needs an algorithm to tell a participating server what keys to use to calculate what partial results.

Several factors may impact how the initiator should decide which servers calculate which partial results w_i, including the relative computational speed of the servers, the network bandwidth among the servers, and the values of (t, n). When (t, n) are small such as when $(t = 3, n = 5)$, such choice will be less significant. However, since the (t, n)-combinatorial secret sharing scheme is not very scalable, when (t, n) are big such as when $(t = 16, n = 24)$, the differences in the initiator's choices will be big. In the case of $(t = 16, n = 24)$, there will be $\binom{24}{16-1} = 1,307,504$ keys in the key set; the initiator, just like other servers, is assigned $\binom{n-1}{t-1} = \binom{24-1}{16-1} = 490,314$ keys. To reduce delay in communicating with other servers, the initiator may choose to calculate as many partial results as it can, from all of its assigned keys. To finish a DiSE transaction in a robust manner against 1-adversary, the initiator will need $\binom{24-1}{16-2} = 817,190$ w_i, each in duplicate, calculated from keys assigned to other $t = 16$ servers. On average, each of these non-initiator server will calculate $\frac{817,190\times 2}{16} \approx 102,149$ w_i.

All the t-combinations out of n servers can be generated in the dictionary order. All members of the first t-combination will be assigned key k_1; the second t-combination will get $k_2; \ldots$ the last t-combination will get key k_u, where $u = \binom{n}{t-1}$. The ordered t-combination is called Γ hereafter. From Γ, a data structure Ω can be derived to show the list of keys assigned to each server.

Algorithm for initiator For an initiator, the following algorithm can be used to find a robust subset, inform them to use which k_i to calculate w_i, receive w_i, check their correctness, and combine all partial results into final result w.

Within a DiSE transaction, to split the w_i load as evenly as possible among the participating servers, for each key that the initiator does not have (found from Ω), the initiator generates a random permutation of the key's owners (found from Γ) and assign that w_i to the first $(\delta+1)$ participating owners in the random permutation.

3.4 Correcting a δ-adversary

The algorithm in the last section allows a DiSE initiator to detect wrong partial results by a δ-adversary. However, it does not correct a wrong partial result value.

To correct an incorrect partial result by a malicious δ-adversary, more honest participating servers will be needed. More specifically, $(t+2\delta)$ participating servers will be needed. In such a robust DiSE with correcting capability, when the initiator checks partial result w_i from the participating servers, instead of simply comparing them, the initiator will pick the value with most votes and use it as the correct partial result.

The proof that $(t+2\delta)$ guarantees correcting goes as follows: in the t-n combinatorial secret sharing scheme, for *any* subset of $(t+2\delta)$ servers, each key k_i of the shared key set is assigned to

Algorithm 1 RobustDiSE $(t, n, u = \binom{n}{t-1}, \delta, \Omega)$

1: **if** $(\delta > \min(t-1, n-t))$ **then**
2: Return failure;
3: **end if**
4: Get the identities of a list of active servers (including itself), Λ
5: **if** (Λ has fewer than $(t+\delta)$ elements) **then**
6: Return failure;
7: **end if**
8: Generate a *random* $(t+\delta)$-subset of Λ, which is a robust subset.
9: Use (Λ, Ω) to decide which w_i should be calculated by which participating server in a load-balancing manner (see more details below) and send the corresponding key IDs to the participating servers respectively.
10: (Each participating server uses the designated key IDs, received from the initiator, to calculate the corresponding partial results and send them back to the initiator.)
11: The initiator calculates local partial results w_i from its local keys
12: **for** all remote partial results w_i received from other servers **do**
13: Compare the $(\delta+1)$ copies of w_i received. If not equal, return "δ-adversary detected"
14: **end for**
15: Combine all partial results (local and remote) into w as $w = w_1 \oplus w_2 \oplus \ldots \oplus w_u$
16: Return w

$(2\delta+1)$ servers. As a result, incorrect partial result value(s) by a δ-adversary based on key k_i will be outnumbered by the correct partial result value on k_i from those $(2\delta+1) - \delta = \delta + 1$ honest participating servers.

We thus have the following result on robust DiSE with correcting capability.

Observation 4 *Let $\lambda = min(t-1, \frac{n-t}{2})$. For any $\delta, 1 \leq \delta \leq \lambda$, there is a robust DiSE scheme with correcting capability against δ-adversary and $(t+2\delta)$ servers are needed to participate in this robust DiSE.*

3.5 A small example

In this section, we use a small example to explain how our robust DiSE works. Let $(t = 3, n = 5)$. In the t-n combinatorial secret sharing scheme, there are 10 symmetric keys $(k_1, k_2, \ldots, k_{10})$ in the key set to be shared. The t-combinations in the dictionary order are $\Gamma = \{ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE, CDE\}$. Table 1 described how these keys are assigned to each server.

Correspondingly, the Ω variable is given in Table 2.

If B is chosen as the initiator, when (C, D, E) are chosen as the other participating servers and the random permutation (E, D, C) is used in the algorithm, Table 3 gives the IDs of the partial results by each participating server in the robust DiSE for $(t = 3, n = 5, \delta = 1)$.

In Table 3, the computing load is almost but not exactly evenly distributed among the non-initiator servers: servers C and D calculate one more partial result than partial result than server E.

Table 1: Symmetric key assigned for ($t = 3, n = 5$)

Server	**Assigned keys**					
A	k_1	k_2	k_3	k_4	k_5	k_6
B	k_1	k_2	k_3	k_7	k_8	k_9
C	k_1	k_4	k_5	k_7	k_8	k_{10}
D	k_2	k_4	k_6	k_7	k_9	k_{10}
E	k_3	k_5	k_6	k_8	k_9	k_{10}

Table 2: Ω for ($t = 3, n = 5$)

Server	**IDs of assigned keys**					
A	1	2	3	4	5	6
B	1	2	3	7	8	9
C	1	4	5	7	8	10
D	2	4	6	7	9	10
E	3	5	6	8	9	10

Table 3: Robust DiSE for ($t = 3, n = 5, \delta = 1$)

Participating servers	**Partial results contributed**									
B (initiator)	w_1	w_2	w_3				w_7	w_8	w_9	
C				w_4	w_5					
D				w_4		w_6				w_{10}
E					w_5	w_6				w_{10}

When server B is chosen as the initiator, Table 4 gives the IDs of the partial results by each participating server in the robust DiSE for ($t = 3, n = 5, \delta = 1$), with correcting capability. In this case, all servers will be needed in this DiSE transaction.

Table 4: Robust DiSE with correcting capability for ($t = 3, n = 5, \delta = 1$)

Participating servers	**Partial results contributed**									
A				w_4	w_5	w_6				
B (initiator)	w_1	w_2	w_3				w_7	w_8	w_9	
C				w_4	w_5					w_{10}
D				w_4		w_6				w_{10}
E					w_5	w_6				w_{10}

3.6 Rotating initiators and abuse resistance

Assuming an honest initiator all the time is essentially against the very idea of robust computing. For a DiSE transaction, *a DiSE client* can pick an active server *randomly* and use it as an initiator for that transaction. This random choice allows the DiSE client to distribute its trust among all the active DiSE servers, not just on any single DiSE server.

This randomness strategy has additional security benefits: it allows DiSE servers to statistically detect a compromised server that has been used abnormally often as initiator to decrypt stolen ciphertexts. Under this randomness strategy, over an extended period of time, each active server should have acted as an initiator over roughly the same number of transactions. As a result, a compromised server that has been overly used as an initiator will be detected by other servers and may be disabled afterwards.

4 Implementation and optimization

For the robust DiSE scheme F-DiSE-R in Section 3, we developed an industry-level implementation in Java. This implementation uses Netty [39, 30], an asynchronous (i.e. non-blocking) and event-driven client/server application framework. As a result, our robust DiSE implementation has good scalability on concurrent network connections, supports code reuse, and is extensible for other distributed cryptography schemes.

In our implementation of F-DiSE-R, the pseudorandom function is instantiated as AES-256 and the cryptographic hash function in $\alpha = h(m||\rho)$ uses SHA-256.

This implementation consists of three components: *dealer*, *client*, and *server*. The dealer package is responsible for generating key sets and assigning them to DiSE servers. The *client* package implements the role of DiSE clients and the *server* package implements the functionality of DiSE servers, including those for an initiator. Once started, a DiSE server polls all other servers to build a list of active servers. It also periodically polls these servers for changes of server status. All the communications among DiSE servers and the communications between a DiSE client and DiSE servers are protected by transport layer security (TLS) with mutual authentication and with cipher suite
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256. All RSA-3072-based TLS certificates are issued by one certification authority (CA) and only this CA is trusted by the DiSE servers and clients. As a result, any adversary that does not have a valid certificate or the corresponding private key will not be able to communicate with DiSE servers; nor can it eavesdrop on, modify, forge, or replay DiSE communications.

All DiSE servers are connected in a local area network, with an actual end-to-end plain communication speed of 500 M bits per second. This speed was reported by iPerf 3.1 [38], a network performance measurement tool.

The CPUs of all these DiSE servers are 3.60GHz Intel Core i7-7700 with eight logic cores. On these cores, over 41-byte messages where each message is encrypted by a different key, the speed of AES-256 in a pure Java software implementation reaches 0.5G bits per second (bps), which forms a performance base line for our implementation. It is worth noting that this benchmark speed is very different from and thus *not* comparable to benchmark speeds by popular tools such as OpenSSL on bigger (say 16 kilobyte) messages or using hardware AES New Instructions (AES-NI). Encrypting bigger messages allows the cost of AES key scheduling to be amortized over many 128-bit data blocks and thus leads to better speeds; Using hardware AES-NI can also significantly improve AES-256 encryption speed.

In the remainder of this section, we shall first give important details of our implementation and then show our performance data.

4.1 Non-robust DiSE vs. robust DiSE

As described in Section 3.3, compared to non-robust DiSE, the major costs of robust DiSE are twofold. First, in δ-robust DiSE transaction (e.g., $\delta = 1$), δ additional servers are required to participate and the initiator will need to send bigger messages (twice as big when $\delta = 1$) to participating servers and receive bigger response messages from them. Second, the initiator needs

Table 5: Communication overhead, in bytes, of 1-robust DiSE for $(t = 16, n = 24)$

	Size of a request	Size of a response	Size of all requests	Size of all responses
Non-robust DiSE	408,673	3,064,472	6,130,095	45,967,080
Robust DiSE	817,273	6,128,972	13,076,368	98,063,552

to check non-local partial results for errors. Checking received partial results for robustness can be measured pretty accurately by the number of CPU cycles and its computing cost turns out to be negligible. The task of sending requests to and receives responses from other participating servers, however, involves network latency and its efficiency cannot be accurately predicted by the number of CPU cycles alone. In our environment, the communication latency is significant compared to the number of CPU cycles used by AES and has become a performance factor. This network latency factor is true for both non-robust DiSE and robust DiSE, but to a different degree. When (t, n) have small values, for example $(t = 3, n = 5)$, the additional communication cost of robust DiSE is relatively small.

However, when (t, n) have big values such as $(t = 16, n = 24)$, the size increases in the request messages from the initiator and the responses will be significant, due to the increase of number of keys in the shared key set. For 1-robust DiSE and non-robust DiSE, Table 5 gives the average size, in bytes, of one request that the initiator needs to send to a participating server, the average size of one response, the total size of all requests, and the total size of all responses. These messages include a 32-byte message header and a message body. In this case, the sizes of the requests and responses double. As a result, the throughput of DiSE transactions will be halved.

4.2 AES pre-key scheduling

The AES-256 encryptions in a DiSE transaction differ from AES-256 encryptions in other applications in two ways. First, DiSE AES-256 encryptions are performed on relatively small messages, which in our implementation have only 41 bytes. Encrypting small messages does *not* allow the cost of AES key scheduling to be amortized into many data block encryptions. Second, in one DiSE transaction, many AES-256 encryptions with different keys are performed. For example, in Table 3 where $(t = 3, n = 5)$, participating server D needs to encrypt $(j||\alpha)$ with key k_4, k_6, k_{10} individually to generate three partial results $w_4 = E_{k_4}(j||\alpha), w_6 = E_{k_6}(j||\alpha), w_{10} = E_{k_{10}}(j||\alpha)$.

Fortunately, these AES-256 keys are long-term keys and as a result, their key scheduling can be pre-computed when a DiSE server starts. In this way, no key scheduling is needed in any new DiSE transaction. This pre-computation can at least double the speed of DiSE computation.

4.3 Timing data of F-DiSE-R

Table 6 gives the time in seconds for one DiSE transaction in F-DiSE-R for the cases of $(t = 3, n = 5)$ and $(t = 16, n = 24)$. The latter case offers a much higher level of security, as an adversary will still *not* be able to access the shared key set even after successfully compromising 15 servers.

In the case of $(t = 3, n = 5)$, among the 0.007 seconds in a robust DiSE transaction, 0.0049 seconds (70% of the time) are for inter-server communications and the remaining time is for

Table 6: Performance data

	Time for one robust DiSE transaction
$(t = 3, n = 5)$	0.007 seconds
$(t = 16, n = 24)$	252.668 seconds

computation.

In the case of $(t = 16, n = 24)$, among the 252.668 seconds in a robust DiSE transaction, 249.749 seconds (98.8% of the time) are for inter-server communications and the remaining time is for computation. In both cases, the performance data is sensitive to communication costs and thus it makes sense to use fast communication links among the DiSE servers.

5 Discussions and applications of robust DiSE

Several factors in the applications of robust DiSE shall be discussed in this section.

5.1 Robust DiSE vs. robust threshold decryption

When a given a piece of data m needs to be protected by distributed cryptography, one can pick either AES-based robust DiSE or RSA-based robust threshold decryption [40]. In both solutions, the decryption of the ciphertext needs the collective efforts of t or more servers and the shared confidential key(s) is *not* reconstructed anywhere in the decryption. In other words, both solutions are intrusion-tolerant. Which of them is a more appropriate solution?

The data m in question is either application-specific data, a data-encrypting key for encrypting a sizeable chunk of application data such as a whole database table, or a data-encrypting key for encrypting a small piece of application data such as one row of a database table.

These cases may require different performances on the underlying distributed cryptographic scheme. For example, in the third case, a database table may have millions of rows/records and each row is encrypted by a different data-encrypting key for fine-grained security control. When these data-encrypting keys are protected by a robust threshold cryptography or DiSE scheme, to encrypt/decrypt the whole table, millions of calls to the robust threshold cryptography or DiSE scheme will be made in a very short period of time. This in turns demands the distributed cryptography scheme to be fast.

Symmetric-key encryption schemes such as AES-128 are often 1000 times faster than public-key encryption schemes such as RSA-3072 that achieve roughly the same level of security. As a result, when t is relatively small (say $t = 3$), F-DiSE-1 is much faster than threshold RSA [40] and F-DiSE-R is much faster than robust threshold RSA.

However, F-DiSE-1 and F-DiSE-R do not scale as well as existing threshold RSA schemes and they lose their performance advantages over threshold RSA when t becomes large, for two reasons. First, in F-DiSE-1 and F-DiSE-R, a set of symmetric keys are shared and are needed to be used in a DiSE transaction, while in a threshold RSA decryption [40], each participating server needs

to perform only one or two modular exponentiations and the combiner needs to perform t modular exponentiations. Second, as shown in Table 6, due to bigger message sizes, robust DiSE is more susceptible to communication cost. Roughly speaking,F-DiSE-R will be caught up by threshold RSA in performance when (t, n) grow to $(t = 11, n = 17)$.

5.2 Trade computation for less communication in F-DiSE-R

In F-DiSE-R, in a $(t = 16, n = 24)$ DiSE transaction, the initiator needs to send out $(817, 190 \times 2)$ requests to participating servers, which is a non-trivial task for applications with slow communication links. Fortunately, this communication cost can be reduced by asking each participating server to calculate, in a consistent manner, what partial results that it should calculate. In this computation-communication trade-off, all servers agree on both the IDs of the keys in the key set and the algorithm to select a server for each key.

From public information Ω (check Section 3.3), any server, including the initiator, can find the IDs of the keys assigned to each server. The initiator will calculate partial results from all of its assigned keys. Each non-initiator participating server receives Θ, a random permutation of the participating servers' IDs, from the initiator. For each non-initiator participating server, for each key that it is assigned, it can find out the t-combination for that key and this is the list of servers who have been assigned the key. If the participating server is among the first two participating servers in Θ, it will calculate a partial result for this key and send it to the initiator.

6 Summary

Long-term secrets that must be stored online for use over an extended period of time, such as a data-encrypting key for a database table, are attractive targets for attacks. In distributed symmetric-key encryption (DiSE), a set of symmetric keys are shared among n servers in a way that any $t, t \leq n$, of these servers can collectively encrypt a given message and decrypt a ciphertext with the key set without reconstructing it. As a result, any adversary who has successfully compromised up to $(t - 1)$ DiSE servers will still *not* be able to access the shared key set.

In a DiSE encryption/decryption transaction, each participating server contributes some partial results, which are combined into the final result by a participating server called the initiator. A DiSE is robust if an honest initiator is able to detect wrong partial results from participating servers that have been compromised and controlled by an adversary. In this article, we developed F-DiSE-R, a robust DiSE. Our robustness construction is through redundant computation. Next, the honest-initiator assumption was removed through rotating the initiator role among all DiSE servers across multiple DiSE transactions. Algorithms to select a robust subset and assign partial results to participating servers were given. We also developed a scalable and industry-level implementation for this robust DiSE, investigated methods to speed up computations, and obtained performance data for the cases of $(t = 3, n = 5)$ and $(t = 16, n = 24)$. Robust DiSE can be deployed to build intrusion-tolerant applications and will significantly enhance computer and applications security.

References

[1] Shashank Agrawal, Payman Mohassel, Pratyay Mukherjee, and Peter Rindal. DiSE: Distributed symmetric-key encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, pages 1993–2010, Toronto, ON, Canada, October 15-19 2018.

[2] Shashank Agrawal, Payman Mohassel, Pratyay Mukherjee, and Peter Rindal. DiSE: Distributed symmetric-key encryption. Cryptology ePrint Archive, Sep 19 2018. URL `https://eprint.iacr.org/2018/727.pdf`.

[3] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators. NIST Special Publication 800-90A Revision 1, June 2015. URL `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf`.

[4] John Black. *Encyclopedia of Cryptography and Security*, chapter Authenticated Encryption. Springer, 2005.

[5] G. R. Blakley. Safeguarding cryptographic keys. In *Proc. Nat. Computer Conf. AFIPS Conf. Proc*, pages 313–317, 1979.

[6] Dan Boneh. The decision Diffie-Hellman problem. In *Proceedings of the Third Algorithmic Number Theory Symposium*, volume 1423 of *Lecture Notes in Computer Science*, pages 48–63. Springer-Verlag, 1998.

[7] C. Boyd. Digital multisignatures. In H. Beker and F. Piper, editors, *Cryptography and coding*, pages 241–246. Clarendon Press, Royal Agricultural College, Cirencester, December 15–17 1989.

[8] E. Brickell, G. Di Crescenzo, and Y. Frankel. Sharing block ciphers. In E. Dawson, A. Clark, and C. Boyd, editors, *ACISP 2000*, volume 1841 of *Lecture Notes in Computer Science*, pages 457–470. Springer–Verlag, April 2000.

[9] R. Canetti, R. Gennaro, A. Herzberg, and D. Naor. Proactive security: Long-term protection against break-ins. *CryptoBytes*, 1(3):1–8, 1997.

[10] D. Chaum and T. P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *Advances in Cryptology - Crypto '92*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105, Berlin, 1992. Springer-Verlag.

[11] R. A. Croft and S. P. Harris. Public-key cryptography and re-usable shared secrets. In H. Beker and F. Piper, editors, *Cryptography and coding*, pages 189–201. Clarendon Press, Royal Agricultural College, Cirencester, December 15–17 1989.

[12] A. De Santis, Y. Desmedt, Y. Frankel, and M. Yung. How to share a function securely. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, pages 522–533, Montréal, Québec, Canada, May 23–25 1994. Clarendon Press.

[13] Y. Desmedt. Society and group oriented cryptography: a new concept. In *Advances in Cryptology, Proc. of Crypto '87*, pages 120–127, August 16–20 1988.

[14] Y. Desmedt. Threshold cryptography. In W. Wolfowicz, editor, *Proceedings of the 3rd Symposium on State and Progress of Research in Cryptography*, pages 110–122, Rome, Italy, February 15–16 1993. invited paper.

[15] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, *Advances in Cryptology — Crypto '89*, pages 307–315, Berlin, August 20–24 1990.

[16] Y. Desmedt and Y. Frankel. Shared generation of authenticators and signatures. In *Advances in Cryptology — Crypto '91*, pages 457–469, August 12–15 1992.

[17] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[18] Morris Dworkin. Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality. NIST Special Publication 800-38C, July 20 2007. URL `https://csrc.nist.gov/publications/detail/sp/800-38c/final`.

[19] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (GCM) and GMAC. NIST Special Publication 800-38D, November 2007. URL `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf`.

[20] Y. Frankel, P. D. MacKenzie, and M. Yung. Robust efficient distributed RSA-key generation. In *Proceedings of the 30th Annual ACM symposium on Theory of Computing*, pages 663–672, Dallas, TX, U.S.A., May 24 - 26 1998.

[21] Rosario Gennario, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In J. Stern, editor, *Advanced in Cryptology - EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 295–310, 1999.

[22] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust and efficient sharing of RSA functions. In *Advances in Cryptology — Crypto '96*, pages 157–172, August 18–22 1996.

[23] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. In *Advances in Cryptology — Eurocrypt '96*, pages 354–371, May 12–16 1996.

[24] R. Gennaro, T. Rabin, S. Jarecki, and H. Krawczyk. Robust and efficient sharing of RSA functions. *Journal of Cryptology*, 13(2):273–300, 2000.

[25] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust threshold DSS signatures. *Information and Computation*, 164(1):54–84, 2001.

[26] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing. In D. Coppersmith, editor, *Advances in Cryptology — Crypto '95*, volume 963 of *Lecture Notes in Computer Science*, pages 339–352, Santa Barbara, California, U.S.A., August 27–31 1995. Springer-Verlag.

[27] Chung L. Liu. *Introduction to Combinatorial Mathematics*. McGraw-Hill College, 1968. ISBN 0070381240.

[28] M. Malkin, T. Wu, and D. Boneh. Experimenting with shared generation of RSA keys. In *Proceedings of the Internet Society's 1999 Symposium on Network and Distributed System Security (SNDSS)*, pages 43–56, 1999.

[29] Keith M. Martin, Rei Safavi-Naini, Huaxiong Wang, and Peter R. Wild. Distributing the encryption and decryption of a block cipher. Cryptology ePrint Archive: Report 2003/005, 2003. URL `https://eprint.iacr.org/2003/005.pdf`.

[30] Norman Maurer and Marvin Allen Wolfthal. *Netty in Action*. Manning Publications, 2015. ISBN 9781617291470.

[31] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In J. Stern, editor, *EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 327–346, 1999.

[32] National Institute of Standards and Technology. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001. URL `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[33] T. Rabin. A simplified approach to threshold and proactive RSA. In *Advances in Cryptology, Proc. of Crypto'98*, pages 89–104, August 23-27 1998.

[34] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, page 73, Seattle, WA, USA, May 14 – 17 1989.

[35] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signature and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[36] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[37] V. Shoup. Practical threshold signatures. In *Advance in Cryptology – EUROCRYPT 2000*, pages 207–220, May 2000.

[38] The iPerf3 Team. iperf - the ultimate speed test tool for tcp, udp and sctp, 2019. URL `https://iperf.fr/iperf-download.php`.

[39] The Netty Project Community. The netty project, May 13 2020. URL `https://netty.io/`.

[40] T. Wu, M. Malkin, and D. Boneh. Building intrusion tolerant applications. In *Proceedings of the 8th USENIX Security Symposium*, pages 79–91, 1999.

[41] Bennet Yee. *Using Secure Coprocessors*. CMU-CS-94-149, Carnegie Mellon University, May 1994.