Improved Security for Decentralized Multi-Client Inner-Product Functional Encryption

Jérémy Chotard[1,2,3], Edouard Dufour-Sans[2,3], Romain Gay[4], Duong Hieu Phan[1], and David Pointcheval[2,3]

[1] XLIM, University of Limoges, CNRS, Limoges, France
[2] DIENS, École normale supérieure, CNRS, PSL University, Paris, France
[3] INRIA, Paris, France
[4] University of California, Berkeley

Abstract. Recently, Chotard *et al.* proposed a variant of functional encryption for Inner Product, where several parties can independently encrypt inputs, for a specific time-period or label, such that functional decryption keys exactly reveal the aggregations for the specific functions they are associated with. This was introduced as *Multi-Client Functional Encryption* (MCFE). In addition, they formalized a *Decentralized* version (DMCFE), where all the clients must agree and contribute to generate the functional decryption keys: there is no need of central authority anymore, and the key generation process is non-interactive between the clients. Eventually, they designed concrete constructions, for both the centralized and decentralized settings, for the inner-product function family.

Unfortunately, there were a few limitations for practical use, in the security model: (1) the clients were assumed not to encrypt two messages under the same label. Then, nothing was known about the security when this restriction was not satisfied; (2) more dramatically, the adversary was assumed to ask for the ciphertexts coming from all the clients or none, for a given label. In case of partial ciphertexts, nothing was known about the security either.

In this paper, our contributions are three-fold: we describe two conversions that enhance any MCFE or DMCFE for Inner Product secure in their security model to (1) handle repetitions under the same label and (2) deal with partial ciphertexts. The latter conversion exploits a new tool, which we call *Secret Sharing Encapsulation* (SSE). It keeps the individual ciphertexts of constant size. Eventually, we propose a new efficient technique to generate the functional decryption keys in a decentralized way, in the case of Inner Product, solely relying on plain DDH, as opposed to prior work of Chotard *et al.* which relies on pairings.

As a consequence, from the *weak* MCFE for Inner Product proposed by Chotard *et al.*, one can obtain an efficient Decentralized MCFE for Inner Product that handles repetitions and partial ciphertexts. This is the first DMCFE where individual ciphertexts are constant size, and whose security does not suffer from any artificial restriction.

Keywords. Functional Encryption, Inner Product, Multi-Client, Decentralized.

1 Introduction

Functional Encryption (FE) [SW05, O'N10, BSW11, GKP+13b, GGH+13] is an alternative to Fully Homomorphic Encryption (FHE) in the context of computation on encrypted data. While FHE outputs the result in an encrypted way, FE outputs the result in clear. Besides, FE generates restricted decryption keys for specific functions, that only decrypt their specific function applied to the message. This is in stark contrast with FHE which has no restrictions on the functions that can be computed on the encrypted data. In particular, FE achieves verifiability for free.

More concretely, for any function f, a functional decryption key dk_f allows, given any ciphertext c with underlying plaintext x, to compute $f(x)$, but does not leak any additional information about the plaintext x. While general definitions with some generic constructions have been proposed [SS10, GVW12, GKP+13b, GKP+13a, Wat15, ABSV15, GGG+14, BGJS16, BKS16], only linear and quadratic functions have been efficiently addressed. Abdalla, Bourse, De Caro, and Pointcheval [ABDP15] proposed the first FE for inner-product function family (IP-FE), based on the Decisional Diffie-Hellman (DDH) assumption, but for the selective security model only: encryption queries are known in advance. Agrawal, Libert and Stehlé [ALS16] achieved adaptive security for IP-FE. Extensions to quadratic functions have also been proposed [Gay16, BCFG17, DGP18].

While the basic definition of FE is quite general, as f could in theory be any function, it requires that the whole input x comes from one party, even if x is a vector $\vec{x} = (x_1, \cdots, x_n)$ with several coordinates. To allow for independent contributions from multiple sources in the case of vector-inputs, two lines of research have been developed: Multi-Input Functional Encryption (MIFE) [GGJS13, GKL+13, GGG+14] and Multi-Client Functional Encryption (MCFE) [GGG+14, GKL+13, CDG+18]. The latter essentially differs from the former with a label which specifies which inputs from the different clients can be combined together. As of today, in these settings, only linear functions have been efficiently addressed. Abdalla *et al.* [AGRW17] proposed an efficient Multi-Input Inner-Product Functional Encryption (IP-MIFE) scheme that relies on the k-Lin assumption in prime-order bilinear groups. Later, Abdalla *et al.* [ACF+18] removed the use of a pairing, building an IP-MIFE from plain DDH, LWE, or the DCR assumption, and adding other features. Recently, Chotard *et al.* [CDG+18] proposed an MCFE and a decentralized MCFE for Inner-Product (IP-MCFE and IP-DMCFE) from the SXDH assumption in prime-order bilinear groups.

1.1 Multi-Client Functional Encryption

For MCFE, as defined in [GGG+14, GKL+13], and more concretely in [CDG+18], both an index i for the client and a label ℓ (possibly a time-stamp) are used for the encryption: $(c_1 = \mathsf{Encrypt}(1, x_1, \ell), \ldots, c_n = \mathsf{Encrypt}(n, x_n, \ell))$. Only ciphertexts with the same label can be used together, in order to get $f(x_1, \ldots, x_n)$ during decryption. This is in contrast to MIFE, where no label, and possibly no index, are provided with the ciphertext, hence many combinations and re-ordering are possible. In such a case, in order to avoid trivial attacks, the adversary is strongly limited with the encryption queries and functional decryption key queries it is allowed to ask. Indeed, with FE and variants, the adversary can get some information from the plaintexts using functional decryption keys. But this should not jeopardize indistinguishability. Hence, one excludes illegitimate attacks that ask for messages that can easily be told apart just from functional decryption keys. Stated otherwise, the information leaked from the ciphertexts by any decryption key is much more important in the setting of MIFE than in MCFE.

In fact, any MCFE for a given functionality directly implies an MIFE for the same functionality, by simply using a fixed label for all encryptions. Reciprocally, an MIFE for general functions directly imply an MCFE for general functions, since the label can be part of the plaintext, and the function can check that every slot used the same label. However, this is not true for the case of smaller classes of functions, such as inner products, which is the focus of this paper. Put simply, adding labels provides a better control over which information is leaked about the encrypted plaintexts.

In addition to allowing multiple-source ciphertexts, [CDG+18] goes further in the distributed process: since senders are distinct and might want to keep control on their data, the validation of the functional decryption keys is thus critical, and cannot be given to a unique authority. They thus proposed a decentralized version of MCFE, where no authority is involved, but the generation of functional decryption keys remains an efficient process, without interactions between the clients.

1.2 Limitations on the Security Model

When dealing with multiple independent clients, it is clear that some input might be missing, leading to an incomplete ciphertext. While it could seem natural that no evaluation can be performed on an incomplete ciphertext, there is no guarantee that the functional decryption key cannot reveal some information about the received inputs.

$$\begin{tikzcd}
 & \begin{matrix}\text{IP-DMCFE}\\(\texttt{IND}^*)\end{matrix} & & \begin{matrix}\text{IP-DMCFE}\\(\texttt{IND})\end{matrix} \\
\begin{matrix}\text{IP-MCFE}\\(\texttt{IND}^*)\end{matrix} \arrow[ur, dashed, "\text{DSum}", "\text{[Section 8.5]}"'] \arrow[rr, "\text{SSE}", "\text{[Theorem 12]}"'] & & \begin{matrix}\text{IP-MCFE}\\(\texttt{IND})\end{matrix} \arrow[ur, dashed, "\text{DSum}", "\text{[Section 8.5]}"'] & \\
 & \begin{matrix}\text{IP-DMCFE}\\(\texttt{wtr-IND}^*)\end{matrix} & & \\
\begin{matrix}\text{IP-MCFE}\\(\texttt{wtr-IND}^*)\end{matrix} \arrow[uu, "\text{IP-FE}", "\text{[Theorem 22]}"'] \arrow[ur, dashed, "\text{DSum}", "\text{[Section 8.5]}"'] \arrow[rr, "\text{SSE}", "\text{[Theorem 12]}"'] & & \begin{matrix}\text{IP-MCFE}\\(\texttt{wtr-IND})\end{matrix} &
\end{tikzcd}$$

$$\begin{tikzcd}
\begin{matrix}\text{MCFE}\\(*\text{-}\texttt{IND}^*)\end{matrix} \arrow[r, "\text{SSE}", "\text{[Theorem 12]}"'] & \begin{matrix}\text{MCFE}\\(*\text{-}\texttt{IND})\end{matrix} \arrow[r, "\text{Sign}", "\text{[Section 7]}"'] & \begin{matrix}\text{MCFE}\\(*\text{-}\texttt{IND-CCA})\end{matrix}
\end{tikzcd}$$

Fig. 1. Contributions and Theorems. Here, `wtr` stands for: "without repetitions".

This is indeed an issue with the protocol proposed by Chotard *et al.* [CDG+18]: in the inner-product case, where one computes $\langle \vec{x}, \vec{y} \rangle$ on a ciphertext of the vector $\vec{x}$ given the functional decryption key for $\vec{y}$, if $y_j = 0$, x_j has no impact on the result. Then, it could seem fine to allow the use of the functional decryption without the j-th ciphertext. But because of the linear properties of the inner-product, and namely for the keys, from many functional keys, one can derive keys for vectors with some zeros, and then decrypt some meaningful information. One could of course keep track of all the possible linear combinations of the keys when deciding on legitimate attacks, but this is very specific to inner-product. Chotard *et al.* [CDG+18] simply decided to declare illegitimate all the attacks with some incomplete ciphertexts.

In an IP-MCFE, each client is allowed to send a unique scalar (one component of the vector). Of course, if he would like to send several, it is possible to register as multiple clients. But then, components would be independent, and would still require the limitation of one value per component and label, whereas in the MIFE, when vectors are input, it makes sense to allow mix-and-match between the inputs. In addition, requiring a unique component per label for each client, while under his responsibility, is a strong limitation. What happens when the client makes a mistake? This is not covered by the security analysis in [CDG+18].

1.3 Contributions

Our contributions are four-fold, as shown on figure 1, and essentially address the above limitations:

– We first deal with the limitation in the security model from [CDG+18], that requires complete ciphertexts: any attack with partial ciphertexts is declared illegitimate. We denote this (weak) security model `IND`*, while our target security model `IND` still considers such attacks legitimate. Our solution is generic, as this is an additional layer, that is applied to the ciphertexts so that, unless the ciphertext is complete (with all the encrypted components), no information leaks about the individual ciphertexts, and thus on each components. This technique relies on a linear secret sharing scheme, hence the name *Secret Sharing Encapsulation* (SSE). It can also be seen as a decentralized version of *All-Or-Nothing Transforms* [Riv97,Boy99,CDH+00]. We propose a concrete instantiation in pairing-friendly groups, under the Decisional Bilinear Diffie-Hellman problem, in the random oracle model. We stress that this conversion transforms

any $*$-IND*-secure MCFE (not necessarily for inner products) into $*$-IND-secure MCFE at constant cost: our conversion just adds two group elements to individual ciphertexts.
– We also show how to prove CCA security generically, simply by adding signatures to ciphertexts for each slot. This transformation applies to any IND-secure MCFE and is not limited to the IP functionality. Even though CCA security is the de facto security notion for encryption, the only way to obtain CCA secure MCFE prior to our work consists of applying the generic Naor Yung paradigm [NY90], that requires extra assumptions (simulation-sound Non Interactive Zero Knowledge arguments), and doubles the ciphertext size. This is true even assuming the random oracle model. In particular, the seminal Fujisaki Okamoto transform [FO99] is of no help for functional encryption, where decryption only recovers partial information of the plaintext.
– Moreover, when starting from the IP-MCFE from [CDG$^+$18], we show how another independent layer of IP-FE allows repetitions, where clients can encrypt vectors (and the global input is the concatenation of all the clients' vectors): more precisely, we will be able to remove the restriction of a unique input per client and per label (wtr-IND*, which stands for "without repetition"). We will thus enhance IND* with repetitions.
– Eventually, we propose an efficient decentralized algorithm to generate a sum of private inputs, hence called DSum, which can convert an IP-MCFE into IP-DMCFE: this technique is inspired from [KDK11], and only applies to the functional decryption key generation algorithm, so it is compatible with the two above conversions. Namely, this improves on the decentralization from [CDG$^+$18] since it does not require pairings.

Efficiency. All the above conversions preserve the efficiency of the underlying MCFE. While the SSE techniques introduce pairings, the two others do not: they only rely on the DDH or even CDH assumptions, in the random oracle model. But we also stress that our SSE techniques is constant-size. A concurrent and independent work [ABKW19] also proposes a compiler from IND*-security to IND-security, without pairings, but ciphertext size becomes linear in the number of clients.

Decentralizing the setup. As explained above, we use an algorithm DSum to decentralize the generation of functional decryption keys. Moreover, because the DSum protocol (described in Section 8), the SSE (described in Section 4) and the original MCFE from [CDG$^+$18] (recalled in Section 6.1) all have a decentralized setup, the resulting scheme also enjoys the same property, thereby completely getting rid of the need for a trusted party generating the private keys. That is, clients only have to agree on a pairing group and hash function to use, then each client can independently generate its public key and encryption (private) key pair. This technique is in the same vein as the idea introduced in a recent independent work [ACF$^+$19]. In that work, the SetUp is fully decentralized by definition, and their construction uses a two-round multi-party computation scheme in the same way as we do with the DSum: one round in the SetUp to output public parameters and another in the decryption key generation to compute the key. However, they do not handle labels, and their construction for inner products relies on the multi-input FE from [ACF$^+$18], which only achieves the IND* security notion.

Technical Tools. In order to deal with partial ciphertexts, we introduce a new tool, called *Secret Sharing Encapsulation* (SSE). In fact, the goal is to allow a user to recover the ciphertexts from the n senders only when she gets the contributions of all of them. At first glance, one may think this could be achieved by using *All-Or-Nothing Transforms* or (n, n)-*Secret Sharing*. However, these settings require an authority who operates on the original messages or generates the shares. Consequently, they are incompatible with our multi-client schemes. Our SSE tool can be seen as a decentralized version of *All-Or-Nothing Transforms* or of (n, n)-*Secret Sharing*: for each label ℓ, each user $i \in [n]$ can generate, on her own, the share $S_{\ell,i}$. And, unless all the shares $S_{\ell,i}$ have been generated, the encapsulated keys are random and perfectly mask all the inputs.

We believe that SSE could be used in other applications. As an example, AONT was used in some traitor tracing schemes [KY02, CPP05]. By using SSE instead of AONT, one would get *decentralized* traitor tracing schemes in which the tracing procedure can only be run if all the authorities agree on the importance of tracing a suspected decoder. This might be meaningful in practice to avoid the abuse of tracing, in particular on-line tracing, which might break the privacy of the users, in case the suspected decoders are eventually legitimate decoders.

2 Definitions and Security Models

In this section, we first review the definitions of MCFE from [CDG+18]. In Section 8, we recall the definition of DMCFE where the generation of functional decryption key is decentralized, and only require use individual secret keys, instead of the master secret key.

We consider the symmetric-key setting, where the encryption key ek_i can also serve to decrypt individual ciphertext for slot $i \in [n]$, using the $\mathsf{Decrypt}$ algorithm (as the functional decryption algorithm $\mathsf{FDecrypt}$, they both take as input a complete vector of individual ciphertexts, but the former uses ek_i to decrypt an individual ciphertext, while the latter uses dk_f to evaluate f on the global plaintext). The algorithm $\mathsf{Decrypt}$ will be used when proving CCA security of our scheme. We can thus enhance the MCFE definition as follows.

2.1 Multi-Client Functional Encryption

We define symmetric-key MCFE, where the user's encryption keys can be used to decrypt the user's components:

Definition 1 (Multi-Client Functional Encryption). *A multi-client functional encryption on $\mathcal{M}$ over a set of n senders is defined by five algorithms:*

- $\mathsf{SetUp}(\lambda)$*: Takes as input the security parameter λ, and outputs the public parameters* mpk*, the master secret key* msk *and the n private encryption keys* ek_i*;*
- $\mathsf{Encrypt}(\mathsf{ek}_i, x_i, \ell)$*: Takes as input a user encryption key* ek_i*, a value x_i to encrypt, and a label ℓ, and outputs the ciphertext $C_{\ell,i}$;*
- $\mathsf{DKeyGen}(\mathsf{msk}, f)$*: Takes as input the master secret key* msk *and a function $f : \mathcal{M}^n \to \mathcal{R}$, and outputs a functional decryption key* dk_f*;*
- $\mathsf{FDecrypt}(\mathsf{dk}_f, \ell, \vec{C})$*: Takes as input a functional decryption key* dk_f*, a label ℓ, and an n-vector ciphertext $\vec{C}$, and outputs $f(\vec{x})$, if $\vec{C}$ is a valid encryption of $\vec{x} = (x_i)_i \in \mathcal{M}^n$ for the label ℓ, or $\perp$ otherwise.*
- $\mathsf{Decrypt}(\mathsf{ek}_i, \ell, \vec{C})$*: Takes as input a label ℓ, a user encryption key* ek_i *and an n-vector ciphertext $\vec{C}$, and outputs a value x_i, if $\vec{C}$ is a valid encryption of $\vec{x} = (x_i)_i \in \mathcal{M}^n$ for the label ℓ, or $\perp$ otherwise.*

As usual, we will assume public parameters being implicitly included in the keys. But unlike [GGG+14, GKL+13, CDG+18], as we consider the symmetric setting, we allow users to decrypt using their own keys, to get back their component plaintexts at least when the ciphertexts were fully generated. Correctness states: given $(\mathsf{mpk}, \mathsf{msk}, (\mathsf{ek}_i)_i) \leftarrow \mathsf{SetUp}(\lambda)$, for any label ℓ, any function $f : \mathcal{M}^n \to \mathcal{R}$, and any vector $\vec{x} = (x_i)_i \in \mathcal{M}^n$, if $C_{\ell,i} \leftarrow \mathsf{Encrypt}(\mathsf{ek}_i, x_i, \ell)$, for $i \in \{1, \dots, n\}$, and $\mathsf{dk}_f \leftarrow \mathsf{DKeyGen}(\mathsf{msk}, f)$, then $\mathsf{FDecrypt}(\mathsf{dk}_f, \ell, \vec{C}_\ell = (C_{\ell,i})_i) = f(\vec{x} = (x_i)_i)$ and $\mathsf{Decrypt}(\mathsf{ek}_i, \ell, \vec{C}_\ell) = x_i$.

2.2 A New Indistinguishability Security Notion

We introduce a new indistinguishability-based security definition, which naturally addresses the shortcomings of the security achieved in prior work [CDG+18]: first, we authorize several challenge ciphertexts for the same user i and label ℓ, contrary to [CDG+18] where encryption is deterministic and therefore only provides security for a single challenge ciphertext per pair (i, ℓ). This make sense in applications where this condition is naturally satisfied, for instance when labels correspond to time stamps, but by removing this limitation, we broaden the range of applications for MCFE.

Second, we strengthen the security model by allowing the adversary to query the left-or-right encryption oracle for some honest users, but not necessarily all of them, leading to incomplete ciphertexts. In [CDG+18], attacks with incomplete ciphertexts are considered non-legitimate, which means that the possible leakage of information on the plaintext by partial decryption (where ciphertexts are known only for a fraction of the total set of users, for a given label) is not captured by the security model.

As in prior work [CDG+18], we consider the case where clients can be dishonest or corrupted. We thus have to consider collusions, where several clients give their secret keys to an adversary who will play on their behalf.

Eventually, we also consider CCA security, with additional functional decryption queries.

We define our new security notion below, and highlight the differences with the security definition from [CDG+18]. Namely, the extra requirements (in gray) corresponds to their weaker security definition, which we call IND*, while IND (framed) is the new, stronger, security notion.

Definition 2 (IND*, $\boxed{\text{IND}}$ -CCA -Security Game for MCFE). *Let us consider MCFE, a scheme over a set of n senders. No adversary $\mathcal{A}$ should be able to win the following security game against a challenger $\mathcal{C}$, with unlimited and adaptive access to the oracles* QEncrypt*,* QLeftRight*,* QFDecrypt*,* QDKeyGen*, and* QCorrupt *described below:*

- *Initialize: the challenger $\mathcal{C}$ runs the setup algorithm $(\mathsf{mpk}, \mathsf{msk}, (\mathsf{ek}_i)_i) \leftarrow \mathsf{SetUp}(\lambda)$ and chooses a random bit $b \overset{\$}{\leftarrow} \{0,1\}$. It provides* mpk *to the adversary $\mathcal{A}$;*
- *Encryption queries $\mathsf{QEncrypt}(i, x, \ell)$: outputs the ciphertext $C_{\ell,i} \leftarrow \mathsf{Encrypt}(\mathsf{ek}_i, x, \ell)$;*
- *Challenge queries $\mathsf{QLeftRight}(i, x^0, x^1, \ell)$: outputs the ciphertext $C_{\ell,i} \leftarrow \mathsf{Encrypt}(\mathsf{ek}_i, x^b, \ell)$;*
- *Functional decryption queries $\mathsf{QFDecrypt}(f, \ell, \vec{C})$: the oracle first asks for the functional decryption key dk_f, and then outputs $\mathsf{FDecrypt}(\mathsf{dk}_f, \ell, \vec{C})$.*
- *Functional decryption key queries $\mathsf{QDKeyGen}(f)$: outputs the functional decryption key $\mathsf{dk}_f \leftarrow \mathsf{DKeyGen}(\mathsf{msk}, f)$;*
- *Corruption queries $\mathsf{QCorrupt}(i)$: outputs the encryption key ek_i;*
- *Finalize: $\mathcal{A}$ provides its guess b' on the bit b, and this procedure outputs the result β of the security game, according to the analysis given below.*

The output β of the game depends on some conditions, where $\mathcal{CS}$ is the set of corrupted senders (the set of indexes i input to QCorrupt *during the whole game), and $\mathcal{HS}$ the set of honest (non-corrupted) senders. We set the output to $\beta \leftarrow b'$, unless one of the cases below is true, in which case we set $\beta \overset{\$}{\leftarrow} \{0,1\}$:*

1. *some $\mathsf{QLeftRight}(i, x_i^0, x_i^1, \ell)$-query has been asked for an index $i \in \mathcal{CS}$ with $x_i^0 \neq x_i^1$* $\boxed{\text{when encryption queries have been asked for all } i \in \mathcal{HS};}$
2. *for some label ℓ and for some function f asked to* QDKeyGen*, there exists a pair of vectors $(\vec{x}^0 = (x_i^0)_i, \vec{x}^1 = (x_i^1)_i)$ such that $f(\vec{x}^0) \neq f(\vec{x}^1)$, when*
   - *$x_i^0 = x_i^1$, for all $i \in \mathcal{CS}$;*
   - *$\mathsf{QLeftRight}(i, x_i^0, x_i^1, \ell)$-queries (or $\mathsf{QEncrypt}(i, x_i, \ell)$-queries if $x_i = x_i^0 = x_i^1$) have been asked for all $i \in \mathcal{HS}$;*
3. *for some label ℓ, a challenge query $\mathsf{QLeftRight}(i, x_i^0, x_i^1, \ell)$ has been asked for some $i \in \mathcal{HS}$, but challenge queries $\mathsf{QLeftRight}(j, x_j^0, x_j^1, \ell)$ or encryption queries $\mathsf{QEncrypt}(j, x_j, \ell)$ have not all been asked for all $j \in \mathcal{HS}$.*

4. *for some* $\mathsf{QFDecrypt}(f, \ell, \vec{C} = (C_i)_i)$*-query, we have*
   - *the answer was not* $\perp$*;*
   - *there exist two vectors* $\vec{x}^0$ *and* $\vec{x}^1$ *such that for all* $i \in \mathcal{CS}$*:* $x_i^0 = x_i^1$*, for all* $i \in \mathcal{HS}$*: either there is a query* $\mathsf{QLeftRight}(i, x_i^0, x_i^1, \ell)$ *that led to* C_i*, or* $x_i^0 = x_i^1 = x_i$ *and there is a query* $\mathsf{QEncrypt}(i, x_i, \ell)$ *that led to* C_i*.*
   - *the above vectors* $\vec{x}^0$ *and* $\vec{x}^1$ *satisfy* $f(\vec{x}^0) \neq f(\vec{x}^1)$*.*

We say MCFE is IND*-secure if for any adversary* $\mathcal{A}$*,* $\mathsf{Adv}^{\mathit{IND}}_{\mathit{MCFE}}(\mathcal{A}) = |\Pr[\beta = 1|b = 1] - \Pr[\beta = 1|b = 0]|$ *is negligible.*

We also define weaker versions of the security game:

- where the adversary must announce in advance the corruption ($\mathsf{QCorrupt}$) queries: static security (sta-IND*/sta-IND);
- where the adversary must announce in advance the challenge ($\mathsf{QLeftRight}$) queries: selective security (sel-IND*/sel-IND);
- where the adversary is limited to one encryption/challenge query on each (i, ℓ): later queries with the same i and ℓ will be ignored by $\mathsf{QEncrypt}$ and $\mathsf{QLeftRight}$: without-repetition security (wtr-IND*/wtr-IND).

Note that the two first above excluded cases are situations where the adversary could trivially distinguish the encrypted vectors, they are thus considered illegitimate attacks:

1. since we are dealing with symmetric-key encryption, where the encryption key and the decryption key are the same, a $\mathsf{QLeftRight}(i, x_i^0, x_i^1, \ell)$-query with $x_i^0 \neq x_i^1$, for $i \in \mathcal{CS}$ leaks b (either at the $\mathsf{QLeftRight}$-query time or at the corruption-time). In our stronger security model, this criteria is less restrictive, applying only when honest encryption are all queried;
2. for any functional decryption key, all the possible evaluations should not trivially allow the adversary to distinguish the ciphertexts generated through $\mathsf{QLeftRight}$-queries (on honest components), only when ciphertexts are complete;

And the last condition is classical, in the CCA-setting, and here in the functional encryption context, as we consider illegitimate functional decryption queries on challenge ciphertexts that could result in a different values depending on the value of b. For such illegitimate attacks, the guess of the adversary is not considered (a random bit β is output). Otherwise, this is a legitimate attack, and the guess b' of the adversary is output.

In [CDG$^+$18], there is the additional restriction on incomplete ciphertexts (in gray), that corresponds to the weaker IND* security notion: if for some label ℓ, a challenge-query $\mathsf{QLeftRight}(i, x_i^0, x_i^1, \ell)$ has been asked for some $i \in \mathcal{HS}$, but the ciphertext is incomplete (which means that there is not at least a challenge-query $\mathsf{QEncrypt}(j, x_j^0, x_j^1, \ell)$ or an encryption-query $\mathsf{QEncrypt}(j, x_j, \ell)$ for all $j \in \mathcal{HS}$), the attack is also considered illegitimate, and one sets $\beta \xleftarrow{\$} \{0, 1\}$.

Remark 3 (The role of the oracle $\mathsf{QEncrypt}$*).* Note that in the IND security game, the oracle $\mathsf{QEncrypt}$ can be simulated by $\mathsf{QLeftRight}$, queried on input $x_i^0 = x_i^1$. However, in the IND*, this oracle gives more power to the adversary: it is not possible for the adversary to query $\mathsf{QLeftRight}$ on some but not all input slots, for a given label (this is the condition 3. from Finalize), but it can query $\mathsf{QEncrypt}$ on incomplete ciphertexts, without triggering Finalize to output a random bit. This will be helpful when going from IND* to IND security, in Section 5.

[CDG$^+$18] gave a construction that only satisfies this weaker wtr-IND* security definition for Inner Product[1]. On the one hand, we show how to go, from any variant of IND*, to the same variant of IND, using an extra Secret Sharing Encapsulation, in Section 5. On the other

[1] In fact, their construction is only proven IND* secure without the oracle $\mathsf{QEncrypt}$, which is not equivalent. However, the proof can be simply adapted, see **??**

hand, we show how to allow repetitions for Inner Product, in Section 6, by adding a layer of single-input Functional Encryption for Inner Product. Since our Secret Sharing Encapsulation is generic (it is not even restricted to inner product encryption), it can be applied after applying the first transformation that permits multiple ciphertexts per input slot and label. Such a resulting scheme achieves a security notion with no artificial restrictions.

3 Notations and Assumptions

3.1 Groups

Prime Order Group. We use a prime-order group generator GGen, a probabilistic polynomial time (PPT) algorithm that on input the security parameter 1^λ returns a description $\mathcal{G} = (\mathbb{G}, p, P)$ of an additive cyclic group $\mathbb{G}$ of order p for a 2λ-bit prime p, whose generator is P.

We use implicit representation of group elements as introduced in [EHK$^+$13]. For $a \in \mathbb{Z}_p$, define $[a] = aP \in \mathbb{G}$ as the *implicit representation* of a in $\mathbb{G}$. More generally, for a matrix $\mathbf{A} = (a_{ij}) \in \mathbb{Z}_p^{n\times m}$ we define $[\mathbf{A}]$ as the implicit representation of $\mathbf{A}$ in $\mathbb{G}$:

$$[\mathbf{A}] := \begin{pmatrix} a_{11}P & \dots & a_{1m}P \\ \\ a_{n1}P & \dots & a_{nm}P \end{pmatrix} \in \mathbb{G}^{n\times m}$$

We will always use this implicit notation of elements in $\mathbb{G}$, i.e., we let $[a] \in \mathbb{G}$ be an element in $\mathbb{G}$. Note that from a random $[a] \in \mathbb{G}$, it is generally hard to compute the value a (discrete logarithm problem in $\mathbb{G}$). Obviously, given $[a], [b] \in \mathbb{G}$ and a scalar $x \in \mathbb{Z}_p$, one can efficiently compute $[ax] \in \mathbb{G}$ and $[a + b] = [a] + [b] \in \mathbb{G}$.

Pairing Group. We also use a pairing-friendly group generator PGGen, a PPT algorithm that on input 1^λ returns a description $\mathcal{PG} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, P_1, P_2, e)$ of asymmetric pairing groups where $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ are additive cyclic groups of order p for a 2λ-bit prime p, P_1 and P_2 are generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively, and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is an efficiently computable (non-degenerate) bilinear map. Define $P_T := e(P_1, P_2)$, which is a generator of $\mathbb{G}_T$. We again use implicit representation of group elements. For $s \in \{1, 2, T\}$ and $a \in \mathbb{Z}_p$, define $[a]_s = aP_s \in \mathbb{G}_s$ as the implicit representation of a in G_s . Given $[a]_1$, $[a]_2$, one can efficiently compute $[ab]_T$ using the pairing e. For two matrices $\mathbf{A}$, $\mathbf{B}$ with matching dimensions define $e([\mathbf{A}]_1, [\mathbf{B}]_2) := [\mathbf{AB}]_T \in \mathbb{G}_T$.

3.2 Computational Assumptions

Definition 4 (Computational Diffie-Hellman Assumption). *The Computational Diffie-Hellman (CDH) Assumption states that, in a prime-order group $\mathcal{G} \overset{\$}{\leftarrow} \mathsf{GGen}(1^\lambda)$, no PPT adversary can compute $[xy]$, from $[x]$ and $[y]$ for $x, y \overset{\$}{\leftarrow} \mathbb{Z}_p$, with non-negligible success probability.*

Equivalently, this assumption states it is hard to compute $[a^2]$ from $[a]$ for $a \overset{\$}{\leftarrow} \mathbb{Z}_p$. This comes from the fact that $4[xy] = [(x + y)^2] - [(x - y)^2]$.

Definition 5 (Decisional Diffie-Hellman Assumption). *The Decisional Diffie-Hellman (DDH) Assumption states that, in a group $\mathcal{G} \overset{\$}{\leftarrow} \mathsf{GGen}(1^\lambda)$, no PPT adversary can distinguish between the two following distributions with non-negligible advantage: $\{([a], [r], [ar]) \mid a, r \overset{\$}{\leftarrow} \mathbb{Z}_p\}$ and $\{([a], [r], [s]) \mid a, r, s \overset{\$}{\leftarrow} \mathbb{Z}_p\}$.*

Equivalently, this assumption states it is hard to distinguish, knowing $[a]$, a random element from the span of $[\vec{a}]$ for $\vec{a} = \binom{1}{a}$, from a random element in $\mathbb{G}^2$: $[\vec{a}] \cdot r = [\vec{a}r] = \binom{[r]}{[ar]} \approx \binom{[r]}{[s]}$.

Definition 6 (Decisional Bilinear Diffie Hellman Assumption). *The Decisional Bilinear Diffie Hellman (DBDH) Assumption states that, in a pairing group $\mathcal{PG} \xleftarrow{\$} \mathsf{PGGen}(1^\lambda)$, for any PPT adversary, the following advantage is negligible, where the probability distribution is over $a, b, c, s \xleftarrow{\$} \mathbb{Z}_p$:*

$$\begin{aligned}\mathsf{Adv}^{\mathit{DBDH}}_{\mathcal{PG}}(\mathcal{A}) = \mid &\Pr[1 \leftarrow \mathcal{A}(\mathcal{PG}, [a]_1, [b]_1, [b]_2, [c]_2, [abc]_T)] \\ &- \Pr[1 \leftarrow \mathcal{A}(\mathcal{PG}, [a]_1, [b]_1, [b]_2, [c]_2, [s]_T)] \mid.\end{aligned}$$

Definition 7 (Q-fold DBDH). *For any integer Q, the Q-fold DBDH assumption states for any PPT adversary, the following advantage is negligible, where the probability distribution is over $a, b, c_i, s_i \xleftarrow{\$} \mathbb{Z}_p$:*

$$\begin{aligned}\mathsf{Adv}^{\mathit{Q\text{-}DBDH}}_{\mathcal{PG}}(\mathcal{A}) = \mid &\Pr[1 \leftarrow \mathcal{A}(\mathcal{PG}, [a]_1, [b]_1, [b]_2, \{[c_i]_2, [abc_i]_T\}_{i\in[Q]})] \\ &- \Pr[1 \leftarrow \mathcal{A}(\mathcal{PG}, [a]_1, [b]_1, [b]_2, \{[c_i]_2, [s_i]_T\}_{i\in[Q]})] \mid.\end{aligned}$$

This Q-fold DBDH assumption is equivalent to classical DBDH assumption:

Lemma 8 (Random Self Reducibility of DBDH). *For any adversary $\mathcal{A}$ against the Q-fold DBDH, running within time t, there exists an adversary $\mathcal{B}$ running within time $t + 2Q(t_{\mathbb{G}_T} + t_{\mathbb{G}_2})$, where $t_{\mathbb{G}_T}$ and $t_{\mathbb{G}_2}$ denote respectively the time for an exponentiation in $\mathbb{G}_T$ and $\mathbb{G}_2$ (we only take into account the time for exponentiations here), such that*

$$\mathsf{Adv}^{\mathit{Q\text{-}DBDH}}_{\mathcal{PG}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathit{DBDH}}_{\mathcal{PG}}(\mathcal{B}).$$

Proof. Upon receiving a DBDH challenge $(\mathcal{PG}, [a]_1, [b]_1, [b]_2, [c]_2, [s]_T)$, $\mathcal{B}$ samples $\alpha_i, c'_i \xleftarrow{\$} \mathbb{Z}_p$ computes $[c_i]_2 := [\alpha_i \cdot c]_2 + [c'_i]_2$, $[s_i]_T := [\alpha_i \cdot s]_T + [c_i \cdot ab]_T$ for all $i \in [Q]$, and gives the challenge $(\mathcal{PG}, [a]_1, [b]_1, [b]_2, \{[c_i]_2, [s_i]_T\}_{i\in[Q]})$ to $\mathcal{A}$.

3.3 Single Input Functional Encryption

A private-key, single input Functional Encryption for a family $\mathcal{F}$ consists of the following PPT algorithms:

- $\mathsf{SetUp}(\lambda)$: on input a security parameter, it outputs a master secret key msk and a public key mpk. The latter is implicitly input of all other algorithms.
- $\mathsf{Encrypt}(\mathsf{msk}, m)$: on input the master secret key and a message, it outputs a ciphertext ct.
- $\mathsf{DKeyGen}(\mathsf{msk}, f)$: on input the master secret key and a function $f \in \mathcal{F}$, it outputs a decryption key dk_f.
- $\mathsf{Dec}(\mathsf{ct}, \mathsf{dk}_f)$: deterministic algorithm that returns a message or a rejection symbol $\perp$ if it fails.

Correctness and security, as defined below, must hold:

Correctness. For any message m, and any function f in the family $\mathcal{F}$, we have: $\Pr[\mathsf{Dec}(\mathsf{ct}, \mathsf{dk}_f) = f(m)] = 1$, where the probability is taken over $(\mathsf{msk}, \mathsf{mpk}) \leftarrow \mathsf{SetUp}(\lambda)$, $\mathsf{ct} \leftarrow \mathsf{Encrypt}(\mathsf{msk}, m)$, and $\mathsf{dk}_f \leftarrow \mathsf{DKeyGen}(\mathsf{msk}, f)$.

Indistinguishability. The security notion is defined by an indistinguishability game similar to the previous one for MCFE:

Definition 9 ($\mathtt{IND}$-Security Game for FE). *Let FE be a functional encryption scheme. No adversary $\mathcal{A}$ should be able to win the following security game:*

- *Initialize: runs $(\mathsf{msk}, \mathsf{mpk}) \leftarrow \mathsf{SetUp}(\lambda)$, choose a random bit $b \xleftarrow{\$} \{0, 1\}$ and returns mpk to $\mathcal{A}$.*

- $\mathsf{QLeftRight}(m_0, m_1)$*: on input two messages* (m_0, m_1)*, returns* $\mathsf{Enc}(\mathsf{msk}, m_b)$.
- $\mathsf{QDKeyGen}(f)$*: on input a function* $f \in \mathcal{F}$*, returns* $\mathsf{DKeyGen}(\mathsf{msk}, f)$.
- *Finalize: it outputs the guess* b' *of* $\mathcal{A}$ *on the bit* b*, unless some* f *was queried to* $\mathsf{QDKeyGen}$ *and* (m_0, m_1) *was queried to* $\mathsf{QLeftRight}$ *such that* $f(m_0) \neq f(m_1)$*, in which case it outputs a uniformly random bit, independent of* $\mathcal{A}$*'s guess.*

The adversary $\mathcal{A}$ *has unlimited and adaptive access to the left-right encryption oracle* $\mathsf{QLeftRight}$*, and to the key generation oracle* $\mathsf{QDKeyGen}$*. We say* FE *is* $\mathtt{IND}$*-secure if for any adversary* $\mathcal{A}$*,* $\mathsf{Adv}^{\mathtt{IND}}_{\mathsf{FE}}(\mathcal{A}) = |\Pr[b' = 1 | b = 1] - \Pr[b' = 1 | b = 0]|$ *is negligible.*

We can also define a weaker selective variant, where pairs (m_0, m_1) to $\mathsf{QLeftRight}$-queries are known from the beginning.

4 Secret Sharing Encapsulation

As explained in Section 2, the difference between our indistinguishability notion and the previous one [CDG+18], is that incomplete ciphertexts were considered illegitimate. This was with the intuition that no adversary should use it since this leaks no information. But actually, an adversary could exploit that in the real-life. Our new security notion requires the scheme to actually leak nothing in such a case.

Here, we present a generic layer, called the Secret Sharing Encapsulation (SSE), that we will use to encapsulate ciphertexts. It allows a user to recover the ciphertexts from the n senders only when he gets the contributions of all the servers. That is, if one sender did not send anything, the user cannot get any information from any of the ciphertexts of the other senders. More concretely, a share of a key $S_{\ell,i}$ is generated for each user $i \in [n]$ and each label ℓ. Unless all the shares $S_{i,\ell}$ have been generated, the encapsulation keys are random and mask all the ciphertexts.

After giving the definition of SSE, we provide a construction whose security is based on the DBDH assumption.

4.1 Definitions

Definition 10 (Secret Sharing Encapsulation (SSE)). *A secret sharing encapsulation on* $\mathcal{K}$ *over a set of* n *senders is defined by four algorithms:*

- $\mathsf{SSE.SetUp}(\lambda)$*: on input a security parameter* λ*, generates the public parameters* $\mathsf{pk}_{\mathsf{sse}}$ *and the personal encryption keys are* $\mathsf{ek}_{\mathsf{sse},i}$ *for all* $i \in [n]$*;*
- $\mathsf{SSE.Encaps}(\mathsf{pk}_{\mathsf{sse}}, \ell)$*: on input* $\mathsf{pk}_{\mathsf{sse}}$ *and the label* ℓ*, outputs a ciphertext* C_ℓ *and an encapsulation key* $K_\ell \in \mathcal{K}$*;*
- $\mathsf{SSE.Share}(\mathsf{ek}_{\mathsf{sse},i}, \ell)$*: on input* $\mathsf{ek}_{\mathsf{sse},i}$ *and the label* ℓ*, outputs the share* $S_{\ell,i}$*;*
- $\mathsf{SSE.Decaps}(\mathsf{pk}_{\mathsf{sse}}, (S_{\ell,i})_{i\in[n]}, \ell, C_\ell)$*: on input all the shares* $S_{\ell,i}$ *for all* $i \in [n]$*, a label* ℓ*, and a ciphertext* C_ℓ*, outputs the encapsulation key* K_ℓ*.*

Correctness. For any label ℓ, we have: $\Pr[\mathsf{SSE.Decaps}(\mathsf{pk}_{\mathsf{sse}}, (S_{\ell,i})_{i\in[n]}, \ell, C_\ell) = K_\ell] = 1$, where the probability is taken over $\big(\mathsf{pk}_{\mathsf{sse}}, (\mathsf{ek}_{\mathsf{sse},i})_{i\in[n]}\big) \leftarrow \mathsf{SSE.SetUp}(\lambda)$, $(C_\ell, K_\ell) \leftarrow \mathsf{SSE.Encaps}(\mathsf{pk}_{\mathsf{sse}}, \ell)$, and $S_{\ell,i} \leftarrow \mathsf{SSE.Share}(\mathsf{ek}_{\mathsf{sse},i}, \ell)$ for all $i \in [n]$.

Indistinguishability. We want to show that the encapsulated keys are indistinguishable from random if not all the shares are known to the adversary. We could define a Real-or-Random security game [BDJR97] for all the masks. Instead, we limit the Real-or-Random queries to one label only, and for all the other labels, the adversary can do the encapsulation by itself, since it just uses a public key. This is well-known that a hybrid proof among the label indices (the order they appear in the game) shows that the One-Label security is equivalent to the Many-Label security. The One-Label definition will be enough for our applications.

Definition 11 (`1-Label-IND`-Security Game for SSE). *Let us consider an* SSE *scheme over a set of* n *senders. We define the following security game against a challenger* $\mathcal{C}$*, where the adversary has unlimited and adaptive access to the oracles* $\mathsf{QRealRandom}$*,* QShare*, and* $\mathsf{QCorrupt}$ *described below.*

– *Initialize(i^*): the adversary announces an index $i^*[n]$. The challenger $\mathcal{C}$ runs the setup algorithm* $(\mathsf{pk}_{\mathsf{sse}}, (\mathsf{ek}_{\mathsf{sse},i})_{i\in[n]}) \leftarrow \mathsf{SetUp}(\lambda)$ *and chooses a random bit* $b \overset{\$}{\leftarrow} \{0,1\}$*. It provides* $\mathsf{pk}_{\mathsf{sse}}$ *to the adversary* $\mathcal{A}$*.*
– *Challenge queries* $\mathsf{QRealRandom}(\ell)$*: outputs a ciphertext* C_ℓ*, together with an encapsulation key* K_ℓ^b*, where* $(C_\ell, K_\ell^0) \leftarrow \mathsf{SSE.Encaps}(\mathsf{pk}_{\mathsf{sse}}, \ell)$*, and* $K_\ell^1 \overset{\$}{\leftarrow} \mathcal{K}$*, where* $\mathcal{K}$ *is the encapsulation key space;*
– *Sharing queries* $\mathsf{QShare}(i, \ell)$*: outputs* $S_{\ell,i} \leftarrow \mathsf{SSE.Share}(\mathsf{ek}_{\mathsf{sse},i}, \ell)$*;*
– *Corruption queries* $\mathsf{QCorrupt}(i)$*: outputs the encapsulation key* $\mathsf{ek}_{\mathsf{sse},i}$*;*
– *Finalize: $\mathcal{A}$ provides its guess b' on the bit b, and this procedure outputs this $\beta \leftarrow b'$ if the following condition is satisfied:* $\mathsf{QRealRandom}$ *is only queried on at most one label ℓ^* and i^* was not queried to* $\mathsf{QCorrupt}$ *and (i^*, ℓ^*) was not queried to* QShare*. If this condition is not satisfied, Finalize outputs a random bit β.*

We say this SSE *is `1-Label-IND`-secure if for any PPT adversary $\mathcal{A}$, its advantage* $\mathsf{Adv}_{\mathsf{SSE}}^{\mathtt{1\text{-}Label\text{-}IND}}(\mathcal{A}) = |\Pr[\beta = 1|b = 1] - \Pr[\beta = 1|b = 0]|$ *is negligible.*

We can also define the weaker static variant, where corruptions are known from the beginning.

4.2 Construction of the Secret Sharing Encapsulation

Let us exhibit a concrete construct for our main tool SSE, in the random oracle model, under the DBDH assumption.

– $\mathsf{SSE.SetUp}(\lambda)$: Takes as input a security parameter λ and generates $\mathcal{PG} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p_{\mathsf{sse}}, P_1, P_2, e) \overset{\$}{\leftarrow} \mathsf{PGGen}_{\mathsf{sse}}(1^\lambda)$. Generates a full domain hash function $\mathcal{H}_{\mathsf{sse}}$ from $\{0,1\}^\lambda$ into $\mathbb{G}_1$. It also generates $\vec{t} \overset{\$}{\leftarrow} \mathbb{Z}_p^n$. The public parameters $\mathsf{pk}_{\mathsf{sse}}$ consist of $(\mathcal{PG}, \mathcal{H}_{\mathsf{sse}}, [\vec{t}]_2)$, and the personal encapsulation keys are $\mathsf{ek}_{\mathsf{sse},i} = t_i$, the i-th coordinate of $\vec{t}$.
– $\mathsf{SSE.Share}(\mathsf{ek}_{\mathsf{sse},i}, \ell)$: Takes as input the key $\mathsf{ek}_{\mathsf{sse},i} = t_i$ and the label ℓ and outputs the share $S_{\ell,i} = t_i \cdot \mathcal{H}_{\mathsf{sse}}(\ell) \in \mathbb{G}_1$.
– $\mathsf{SSE.Encaps}(\mathsf{pk}_{\mathsf{sse}}, \ell)$: Takes as input the public key $\mathsf{pk}_{\mathsf{sse}} = (\mathcal{PG}, \mathcal{H}_{\mathsf{sse}}, [\vec{t}]_2)$ and the label ℓ, samples $r \overset{\$}{\leftarrow} \mathbb{Z}_p$, and outputs the ciphertext C_ℓ and the encapsulation key K_ℓ defined as: $C_\ell = [r]_2, K_\ell = e(\mathcal{H}_{\mathsf{sse}}(\ell), [r \sum_{j\in[n]} t_j]_2)$.
– $\mathsf{SSE.Decaps}(\mathsf{pk}_{\mathsf{sse}}, (S_{\ell,i})_{i\in[n]}, \ell, C_\ell)$: Takes as input all the shares $S_{\ell,i}$ for all $i \in [n]$, a label ℓ and a ciphertext C_ℓ, and outputs an encapsulation key

$$K_\ell = e\big(\sum_j S_{\ell,j}, C_\ell\big).$$

We stress here that K_ℓ is not unique for each label ℓ: whereas $S_{\ell,i}$ deterministically depends on ℓ and the client i, K_ℓ is randomized by the random coins r in C_ℓ. Hence, with all the shares, using a specific C_ℓ, one can recover the associated K_ℓ. Correctness follows from the fact that the above decapsulated key K_ℓ is equal to

$$e\big(\sum_j t_j \mathcal{H}_{\mathsf{sse}}(\ell), [r]_2\big) = e\big(\mathcal{H}_{\mathsf{sse}}(\ell), [r \cdot \sum_j t_j]_2\big),$$

where the pair (C_ℓ, K_ℓ) has been generated by $\mathsf{SSE.Encaps}(\mathsf{pk}_{\mathsf{sse}}, \ell)$ with random r. The intuition for the security is that given all the $S_{\ell,i} = t_i \cdot \mathcal{H}_{\mathsf{sse}}(\ell)$ for a label ℓ, one can recover the masks $K_\ell = e(\mathcal{H}_{\mathsf{sse}}(\ell), [r \sum_j t_j]_2)$ using $C_\ell = [r]_2$. However if $S_{\ell,i}$ is missing for one slot i, then all the encapsulation keys K_ℓ are pseudo-random, from the DBDH assumption.

4.3 Security Proof

Let $\mathcal{A}$ be a PPT adversary against the security of the above SSE. We build a PPT adversary $\mathcal{B}$ against the q_r-fold DBDH such that:

$$\mathsf{Adv}^{\texttt{1-Label-IND}}_{\mathsf{SSE}}(\mathcal{A}) \leq (1+q_h) \cdot \mathsf{Adv}^{q_r\text{-}\mathsf{DBDH}}_{\mathcal{PG}}(\mathcal{B}),$$

where q_h denotes the number of $\mathcal{H}_{\mathsf{sse}}$ queries (explicit or implicit) and q_r the number of challenge-queries to the $\mathsf{QRealRandom}$ oracle. Applying Lemma 8, one can reduce the security to the DBDH assumption.

$\mathcal{B}$ receives a q_r-fold DBDH challenge $\big(\mathcal{PG}, [a]_1, [b]_1, [b]_2, \{[c_i]_2, [s_i]_T\}_{i\in[q_r]}\big)$, where q_r denotes the number of queries of $\mathcal{A}$ to its oracle $\mathsf{QRealRandom}$, and receives $i^\star \in [n]$ from $\mathcal{A}$.

Then, $\mathcal{B}$ guesses $\rho \overset{\$}{\leftarrow} \{0, \ldots, q_h\}$. Intuitively, ρ is a guess on when the random oracle is going to be queried on $\ell^\star$, the first label used as input to $\mathsf{QRealRandom}$ (without loss of generality, we can assume $\mathsf{QRealRandom}$ is queried at least once by $\mathcal{A}$, otherwise the security is trivially satisfied), with $\rho = 0$ indicating that the adversary never queries $\mathcal{H}_{\mathsf{sse}}$ on $\ell^\star$ before querying $\mathsf{QRealRandom}$.

Then, $\mathcal{B}$ samples $t_i \overset{\$}{\leftarrow} \mathbb{Z}_p$ and sets $\mathsf{ek}_{\mathsf{sse},i} := t_i$ for all $i \in [n]$, $i \neq i^\star$, and sets $[t_{i^\star}]_2 = [b]_2$. It returns $\mathsf{pk}_{\mathsf{sse}} = (\mathcal{PG}, [\vec{t}]_2)$ to $\mathcal{A}$.

For any query $\mathsf{QCorrupt}(i)$: if $i \neq i^\star$, $\mathcal{B}$ returns $\mathsf{ek}_{\mathsf{sse},i}$, otherwise $\mathcal{B}$ stops simulating the experiment for $\mathcal{A}$ and returns 0 to its own experiment.

For any query to the random oracle $\mathcal{H}_{\mathsf{sse}}$, if this the ρ'th new query, then $\mathcal{B}$ sets $\mathcal{H}_{\mathsf{sse}}(\ell_\rho) := [a]_1$. For others queries, $\mathcal{B}$ outputs $[h]_1$ for a random $h \overset{\$}{\leftarrow} \mathbb{Z}_p$. $\mathcal{B}$ keeps track of the queries and outputs to the random oracle $\mathcal{H}_{\mathsf{sse}}$, so that it answers two identical queries with the same output.

For any query to $\mathsf{QRealRandom}(\ell)$: if ℓ has never been queried to the random oracle $\mathcal{H}_{\mathsf{sse}}$ before (directly, or indirectly via QShare) and $\rho = 0$, then $\mathcal{B}$ sets $\mathcal{H}_{\mathsf{sse}}(\ell) := [a]_1$; if ℓ was queried to random oracle as the ρ'th new query (again, we consider direct and indirect queries to $\mathcal{H}_{\mathsf{sse}}$, the latter coming from QShare), then we already have $\mathcal{H}_{\mathsf{sse}}(\ell) = [a]_1$. In both cases, $\mathcal{B}$ sets $C_\ell \leftarrow [c_j]_2$, for the next index j in the q_r-fold DBDH instance, computes $K_\ell \leftarrow [s_j]_T + e([a]_1, (\sum_{i\neq i^\star} t_i) \cdot [c_j]_2)$, and returns (C_ℓ, K_ℓ) to $\mathcal{A}$. Otherwise, the guess ρ was incorrect: $\mathcal{B}$ stops simulating the experiment for $\mathcal{A}$, and returns 0 to its own experiment. Moreover, if $\mathcal{A}$ ever calls $\mathsf{QRealRandom}$ on different labels ℓ, then $\mathcal{B}$ stops simulating this experiment for $\mathcal{A}$ and returns 0 to its own experiment.

For any query to $\mathsf{QShare}(i, \ell)$: if the random has been called on ℓ, then $\mathcal{B}$ uses the already computed input $\mathcal{H}_{\mathsf{sse}}(\ell)$; otherwise, it computes $\mathcal{H}_{\mathsf{sse}}(\ell)$ for the first time as explained above. If $i = i^\star$ and $\ell = \ell_\rho$, then $\mathcal{B}$ stops simulating the experiment for $\mathcal{A}$ and returns 0 to its own experiment. Otherwise, that means either $i \neq i^\star$, in which case $\mathcal{B}$ knows $t_i \in \mathbb{Z}_p$, or $\ell \neq \ell_\rho$, in which case $\mathcal{B}$ the discrete logarithm of $\mathcal{H}_{\mathsf{sse}}(\ell)$. In both cases, $\mathcal{B}$ can compute $S_{\ell,i} := t_i \cdot \mathcal{H}_{\mathsf{sse}}(\ell) \in \mathbb{G}_1$, which it returns to $\mathcal{A}$.

At the end of the experiment, $\mathcal{B}$ receives the output α from $\mathcal{A}$. If its guess ρ was correct, $\mathcal{B}$ outputs α to its own experiment, otherwise, it ignores α and returns 0.

When $\mathcal{B}$'s guess is incorrect, it returns 0 to its experiment. Otherwise, when it is given as input a real q_r-fold DBDH challenge, that is $s_j = abc_j$ for all indices $j \in [q_r]$, then $\mathcal{B}$ simulates the 1-label-IND security game with $b = 0$. Indeed, since $b = t_{i^\star}$, for the j-th query to $\mathsf{QRealRandom}$, we have:

$$\begin{aligned}
K_{\ell^\star} &= [s_j]_T + e([a]_1, (\sum_{i\neq i^\star} t_i) \cdot [c_j]_2) = [abc_j]_T + e([a]_1, (\sum_{i\neq i^\star} t_i) \cdot [c_j]_2) \\
&= e([a]_1, [bc_j]_2) + e([a]_1, (\sum_{i\neq i^\star} t_i) \cdot [c_j]_2) = e([a]_1, [bc_j]_2 + (\sum_{i\neq i^\star} t_i) \cdot [c_j]_2) \\
&= e([a]_1, (b + \sum_{i\neq i^\star} t_i) \cdot [c_j]_2) = e([a]_1, \sum_i t_i \cdot [c_j]_2) = e(\mathcal{H}_{\mathsf{sse}}(\ell^\star), \sum_i t_i \cdot [c_j]_2)
\end{aligned}$$

where $C_{\ell^\star} = [c_j]_2$. When given as input a random q_r-fold DBDH challenge, the simulation corresponds to the case $b = 1$. Finally, we conclude using the fact that the guess ρ is correct with probability exactly $\frac{1}{q_h+1}$.

5 Strengthening the Security of MCFE using SSE

We now show how we can enhance the security of any MCFE using a Secret Sharing Encapsulation as defined in Section 4. Namely, we show that the construction of Section 5.1 is IND secure if the underlying the MCFE is IND*-secure, thereby removing the complete-ciphertext restriction, as incomplete ciphertexts do not leak any information thanks to the SSE layer.

5.1 Generic Construction of IND-Secure MCFE

Let $\mathsf{MCFE} = (\mathsf{SetUp}, \mathsf{Encrypt}, \mathsf{DKeyGen}, \mathsf{FDecrypt}, \mathsf{Decrypt})$ be a Multi-Client Functional Encryption (Definition 1), $\mathsf{SSE} = (\mathsf{SSE.SetUp}, \mathsf{SSE.Encaps}, \mathsf{SSE.Decaps})$ be a Secret Sharing Encapsulation (Definition 10), and $\mathsf{SKE} = (\mathsf{SEnc}, \mathsf{SDec})$ be Symmetric Encryption scheme (Appendix A) with same key space as SSE, and whose message space is the ciphertext space of MCFE. We define $\mathsf{MCFE}' = (\mathsf{SetUp}', \mathsf{Encrypt}', \mathsf{DKeyGen}', \mathsf{FDecrypt}', \mathsf{Decrypt}')$ as follows:

- $\mathsf{SetUp}'(\lambda)$: It executes $(\mathsf{mpk}, \mathsf{msk}, (\mathsf{ek}_i)_i) \leftarrow \mathsf{SetUp}(\lambda)$ and $(\mathsf{pk}_{\mathsf{sse}}, (\mathsf{ek}_{\mathsf{sse},i})_i) \leftarrow \mathsf{SSE.SetUp}(\lambda)$. The public parameters mpk' consist of $\mathsf{mpk} \cup \mathsf{pk}_{\mathsf{sse}}$, while the encryption keys are $\mathsf{ek}'_i = \mathsf{ek}_i \cup \mathsf{ek}_{\mathsf{sse},i}$ for $i = 1, \ldots, n$, and the master secret key is $\mathsf{msk}' = \mathsf{msk}$;
- $\mathsf{Encrypt}'(\mathsf{ek}'_i, x_i, \ell)$: It parses the encryption key ek'_i as $\mathsf{ek}_i \cup \mathsf{ek}_{\mathsf{sse},i}$, runs $C_{\ell,i} \leftarrow \mathsf{Encrypt}(\mathsf{ek}_i, x_i, \ell)$, executes $(D_{\ell,i}, K_{\ell,i}) \leftarrow \mathsf{SSE.Encaps}(\mathsf{pk}_{\mathsf{sse}}, \ell)$, and $S_{\ell,i} \leftarrow \mathsf{SSE.Share}(\mathsf{ek}_{\mathsf{sse},i}, \ell)$. The ciphertext $C'_{\ell,i}$ is then set to the tuple $(E_{\ell,i} = \mathsf{SEnc}(K_{\ell,i}, C_{\ell,i}), D_{\ell,i}, S_{\ell,i})$;
- $\mathsf{DKeyGen}'(\mathsf{msk}', f)$: With $\mathsf{msk} = \mathsf{msk}'$, it runs $\mathsf{dk}_f = \mathsf{DKeyGen}(\mathsf{msk}, f)$;
- $\mathsf{FDecrypt}'(\mathsf{dk}_f, \ell, (C'_{\ell,i})_{i\in[n]})$: Takes as input a functional decryption key dk_f, a label ℓ, and ciphertexts $(C'_{\ell,i} = (E_{\ell,i}, D_{\ell,i}, S_{\ell,i}))_{i\in[n]}$. It operates in two steps; first it applies $\mathsf{SSE.Decaps}_{\mathsf{sse}}(\mathsf{pk}_{\mathsf{sse}}, (S_{\ell,j})_{j\in[n]}, \ell, D_{\ell,i})$ on all the ciphertexts $D_{\ell,i}$ to get all the encapsulation keys $K_{\ell,i}$'s and thus all the plaintexts $C_{\ell,i}$'s using SDec on $E_{\ell,i}$. Then it runs $\mathsf{FDecrypt}(\mathsf{dk}_f, \ell, (C_{\ell,i})_{i\in[n]})$;
- $\mathsf{Decrypt}'(\mathsf{ek}_i, \ell, (C'_{\ell,i})_{i\in[n]})$: Takes as input an encryption key ek_i, a label ℓ, and ciphertexts $(C'_{\ell,i} = (E_{\ell,i}, D_{\ell,i}, S_{\ell,i}))_{i\in[n]}$. It operates in two steps; first it applies $\mathsf{SSE.Decaps}_{\mathsf{sse}}(\mathsf{pk}_{\mathsf{sse}}, (S_{\ell,j})_{j\in[n]}, \ell, D_{\ell,i})$ on all the ciphertexts $D_{\ell,i}$ to get all the encapsulation keys $K_{\ell,i}$'s and thus all the plaintexts $C_{\ell,i}$'s using SDec on $E_{\ell,i}$. Then it runs $\mathsf{Decrypt}(\mathsf{ek}_i, \ell, (C_{\ell,i})_{i\in[n]})$.

5.2 Security Analysis

We now show that this generic construction MCFE' achieves IND-security, assuming the underlying MCFE is IND*-secure (see Definition 2), SSE is 1-Label-IND-secure (see Definition 11), and the symmetric encryption is one-time secure (see definition in Appendix A). More precisely, we can state the following security result:

Theorem 12. *For any adversary $\mathcal{A}$ running within time t, against the IND-security of the above MCFE',*

$$\mathsf{Adv}^{\mathit{IND}}_{\mathit{MCFE}'}(\mathcal{A}) \leq (n+1)\cdot L \times \left(\begin{array}{c} \mathsf{Adv}^{\mathit{IND}^*}_{\mathit{MCFE}}(t) + 2\cdot \mathsf{Adv}^{\mathit{1\text{-}Label\text{-}IND}}_{\mathsf{SSE}}(t') \\ +q_e \cdot \mathsf{Adv}^{\mathsf{OT}}_{\mathsf{SKE}}(t'') \end{array} \right),$$

with t' and t'' quite close to t, where L is the total number of labels queried to the oracle $\mathsf{QLeftRight}'$*, and q_e is the maximum number of queries to* $\mathsf{QLeftRight}'$ *for a given label. In addition* $\mathsf{Adv}(t)$*, for any security notion, is the maximum advantage an algorithm can get within time t.*

We stress that this security result keeps all the properties of the basic MCFE and the SSE schemes:

– if MCFE and SSE are both secure against adaptive corruptions, MCFE' is also $\mathtt{IND}$ against adaptive corruptions;
– if MCFE is secure with repetitions (see Section 6), MCFE' is also $\mathtt{IND}$ with repetitions.

The proof uses a hybrid argument that goes over all the labels $\ell_1, \dots, \ell_L$ used as input to the queries $\mathcal{A}$ makes to the $\mathsf{QLeftRight}'$ oracle. We define the following hybrid games, for all $\rho = 0, \dots, L$:

Game $\mathbf{G}_\rho$: This hybrid game outputs right answers for the $\mathsf{QLeftRight}'$-queries involving the first ρ labels, and left answers for the other labels, to the $\mathtt{IND}$-adversary $\mathcal{A}$, as follows:

– Initialize: it gets the global parameters $(\mathsf{mpk}, \mathsf{msk}, (\mathsf{ek}_i)_{i\in[n]}) \leftarrow \mathsf{SetUp}(\lambda)$, $(\mathsf{pk}_{\mathsf{sse}}, (\mathsf{ek}_{\mathsf{sse},i})_{i\in[n]}) \leftarrow \mathsf{SSE.SetUp}(\lambda)$ and it returns the public ones $\mathsf{mpk}' = \mathsf{mpk} \cup \mathsf{pk}_{\mathsf{sse}}$ to the adversary $\mathcal{A}$;
– $\mathsf{QEncrypt}'(i, x, \ell_j)$: it returns $\mathsf{Encrypt}'(\mathsf{ek}_i, x, \ell_j)$;
– $\mathsf{QLeftRight}'(i, x^0, x^1, \ell_j)$: if $j \leq \rho$, it returns $\mathsf{Encrypt}'(\mathsf{ek}_i, x^1, \ell_j)$, if $j > \rho$, it returns $\mathsf{Encrypt}'(\mathsf{ek}_i, x^0, \ell_j)$;
– $\mathsf{QDKeyGen}'(f)$: it returns $\mathsf{DKeyGen}'(\mathsf{msk}, f)$;
– $\mathsf{QCorrupt}'(i)$: it returns $\mathsf{ek}'_i = \mathsf{ek}_i \cup \mathsf{ek}_{\mathsf{sse},i}$;
– Finalize: as in Definition 2, for $\mathtt{IND}$-security.

For any hybrid game $\mathbf{G}_\rho$, we denote by $\mathsf{Adv}_{\mathbf{G}_\rho}(\mathcal{A}) := \Pr[\beta = 1]$, where β is the output of Finalize. Note that $\mathsf{Adv}^{\mathtt{IND}}_{\mathsf{MCFE}'}(\mathcal{A}) = |\mathsf{Adv}_{\mathbf{G}_0}(\mathcal{A}) - \mathsf{Adv}_{\mathbf{G}_L}(\mathcal{A})|$. Lemma 13 states that for all $i \in [L]$, $|\mathsf{Adv}_{\mathbf{G}_{i-1}}(\mathcal{A}) - \mathsf{Adv}_{\mathbf{G}_i}(\mathcal{A})|$ is negligible, which concludes the proof.

Lemma 13. *For any adversary $\mathcal{A}$ against the $\mathtt{IND}$-security of the above MCFE', for all $\rho \in [L]$, there exist PPT adversaries $\mathcal{B}_\rho$, $\mathcal{B}'_\rho$, and $\mathcal{B}''_\rho$ such that*

$$|\mathsf{Adv}_{\mathbf{G}_{\rho-1}}(\mathcal{A}) - \mathsf{Adv}_{\mathbf{G}_\rho}(\mathcal{A})| \leq (n+1) \cdot \begin{pmatrix} \mathsf{Adv}^{\mathtt{IND}^*}_{\mathsf{MCFE}}(\mathcal{B}_\rho) + \\ 2 \cdot \mathsf{Adv}^{\mathtt{1\text{-}Label\text{-}IND}}_{\mathsf{SSE}}(\mathcal{B}'_\rho) + q_e \cdot \mathsf{Adv}^{\mathsf{OT}}_{\mathsf{SKE}}(\mathcal{B}''_\rho) \end{pmatrix}$$

Proof (of Lemma 13). Actually, two cases can happen between games $\mathbf{G}_{\rho-1}$ and $\mathbf{G}_\rho$, for each $\rho \in \{1, \dots, L\}$: either all the honest components of the ciphertext are generated under ℓ_ρ or not all of them. We first make the guess, and then deal with the two cases: if they are all generated (for honest clients), this is the simple $\mathtt{IND}^*$ security game for the underlying MCFE, otherwise there is an honest index i^* for which the ciphertext has not been generated, and the SSE scheme will help, together with the symmetric encryption scheme:

Guess of the Case for the ℓ_ρ: We define a new sequence of hybrid games $\mathbf{G}^*_\rho$ for all $\rho = 0, \dots, n$, which is exactly as above, except that a guess for the missing honest-client ciphertext i^* under ℓ_ρ is performed ($i^* = 0$ means that all the honest-client ciphertexts are expected to be generated under ℓ_ρ):

– Initialize: it first makes a guess for $i^* \xleftarrow{\$} \{0, \dots, n\}$, and then does as in $\mathbf{G}_\rho$;
– $\mathsf{QEncrypt}'(i, x, \ell_j)$, $\mathsf{QLeftRight}'(i, x^0, x^1, \ell_j)$, $\mathsf{QDKeyGen}'(f)$, $\mathsf{QCorrupt}'(i)$, as in $\mathbf{G}_\rho$;
– Finalize: as in $\mathbf{G}_\rho$, except if
  • $i^* = 0$, but not all the honest ciphertexts under ℓ_ρ have been asked;
  • $i^* \neq 0$, but client i^* is corrupted;
  • $i^* \neq 0$, but the i^*-th client ciphertext has been asked under ℓ_ρ;

  in which cases a random bit is output.

Since $\mathbf{G}^*_\rho$ and $\mathbf{G}_\rho$ are the same when the guess incorrect, which happens with probability exactly $1/(n+1)$, for any adversary $\mathcal{A}$: $\mathsf{Adv}_{\mathbf{G}_\rho}(\mathcal{A}) = (n+1) \cdot \mathsf{Adv}_{\mathbf{G}^*_\rho}(\mathcal{A})$.

All the Ciphertexts are Generated under ℓ_ρ: Under the condition that $\mathcal{A}$ asks for all the honest ciphertexts under ℓ_ρ, which means the correct guess is $i^* = 0$, we build a PPT adversary $\mathcal{B}_\rho$ against the $\mathtt{IND}^*$ security of MCFE such that $|\mathsf{Adv}_{\mathbf{G}^*_{\rho-1}}(\mathcal{A} \wedge i^* = 0) - \mathsf{Adv}_{\mathbf{G}^*_\rho}(\mathcal{A} \wedge i^* = 0)| \leq \mathsf{Adv}^{\mathtt{IND}^*}_{\mathsf{MCFE}}(\mathcal{B}_\rho)$. $\mathcal{B}_\rho$ simulates the $\mathtt{IND}$-adversary $\mathcal{A}$'s view as follows:

- Initialize: it obtains mpk from its own $\mathtt{IND}^*$-security game for MCFE, samples $(\mathsf{pk}_{\mathsf{sse}}, (\mathsf{ek}_{\mathsf{sse},i})_{i\in[n]}) \leftarrow \mathsf{SSE.SetUp}(\lambda)$ and returns $\mathsf{mpk}' = \mathsf{mpk} \cup \mathsf{pk}_{\mathsf{sse}}$ to the adversary $\mathcal{A}$;
- $\mathsf{QEncrypt}'(i, x, \ell_j)$: it uses its own encryption oracle $\mathsf{QEncrypt}$ to get $C_{\ell_j,i} \leftarrow \mathsf{QEncrypt}(i, x, \ell_j)$. Then, it computes $(D_{\ell_j,i}, K_{\ell_j,i}) \leftarrow \mathsf{SSE.Encaps}(\mathsf{pk}_{\mathsf{sse}}, \ell_j)$, and $S_{\ell_j,i} \leftarrow \mathsf{SSE.Share}(\mathsf{ek}_{\mathsf{sse},i}, \ell_j)$. Eventually, it computes and returns the ciphertext $(E_{\ell_j,i} = \mathsf{SEnc}(K_{\ell_j,i}, C_{\ell_j,i}), D_{\ell_j}, S_{\ell_j,i})$;
- $\mathsf{QLeftRight}'(i, x^0, x^1, \ell_j)$:
  - if $j < \rho$, it uses its own encryption oracle $\mathsf{QEncrypt}$ to get the ciphertext $C_{\ell_j,i} \leftarrow \mathsf{QEncrypt}(i, x^1, \ell_j)$;
  - if $j > \rho$, it uses its own encryption oracle $\mathsf{QEncrypt}$ to get the ciphertext $C_{\ell_j,i} \leftarrow \mathsf{QEncrypt}(i, x^0, \ell_j)$;
  - if $j = \rho$, then it uses its own left-or-right encryption oracle to get the ciphertext $C_{\ell_j,i} \leftarrow \mathsf{QLeftRight}(i, x^0, x^1, \ell_\rho)$.

  Then, it computes the encapsulation $(D_{\ell_j,i}, K_{\ell_j,i}) \leftarrow \mathsf{SSE.Encaps}(\mathsf{pk}_{\mathsf{sse}}, \ell_j)$ and the share $S_{\ell_j,i} \leftarrow \mathsf{SSE.Share}(\mathsf{ek}_{\mathsf{sse},i}, \ell_j)$. Eventually, it returns the ciphertext $(E_{\ell_j,i} = \mathsf{SEnc}(K_{\ell_j,i}, C_{\ell_j,i}), D_{\ell_j,i}, S_{\ell_j,i})$;
- $\mathsf{QCorrupt}'(i)$: it uses its own corruption oracle to get $\mathsf{ek}_i \leftarrow \mathsf{QCorrupt}(i)$, and returns $\mathsf{ek}'_i = \mathsf{ek}_i \cup \mathsf{ek}_{\mathsf{sse},i}$;
- Finalize: $\mathcal{B}_\rho$ checks whether all the honest ciphertexts under ℓ_ρ have been asked. If not, it ignores $\mathcal{A}$'s guess and sends a uniformly random bit $\beta \xleftarrow{\$} \{0,1\}$; Otherwise, it forwards $\mathcal{A}$'s guess to the Finalize procedure of the $\mathtt{IND}^*$-security game.

When the guess $i^* = 0$ is correct, the queries $\mathcal{B}_\rho$ makes to its $\mathsf{QLeftRight}$ oracle are valid, i.e. they don't make the Finalize procedure output a uniformly random bit (independent of $\mathcal{B}_\rho$'s guess). Indeed, if $\mathsf{QLeftRight}(i, \cdot, \cdot, \ell_\rho)$ is queried for some $i \in [n]$, then for all slots $j \in \mathcal{HS}$, $\mathsf{QLeftRight}(j, \cdot, \cdot, \ell_\rho)$ is also queried. Thus, we can use the $\mathtt{IND}^*$ security of MCFE to switch $\mathsf{Encrypt}'(\mathsf{ek}_i, \vec{x}^0, \ell_\rho)$ as in game $\mathbf{G}^*_{\rho-1}$ to $\mathsf{Encrypt}'(\mathsf{ek}_i, \vec{x}^1, \ell_\rho)$ as in game $\mathbf{G}^*_\rho$.

Some Ciphertexts are Missing under ℓ_ρ: For $\beta \in \{0,1\}$, we define the game $\mathbf{H}_{\rho,\beta}$ as $\mathbf{G}^*_\rho$, except that when $i^* \neq 0$, $\mathsf{QEncrypt}'(i, x, \ell_\rho)$ encrypts x and $\mathsf{QLeftRight}'(i, x^0, x^1, \ell_\rho)$ encrypts x^β in $C_{\ell_\rho,i}$, then they both generate the encapsulation $(D_{\ell_\rho,i}, K_{\ell_\rho,i}) \leftarrow \mathsf{SSE.Encaps}(\mathsf{pk}_{\mathsf{sse}}, \ell_\rho)$ and the share $S_{\ell_\rho,i} \leftarrow \mathsf{SSE.Share}(\mathsf{ek}_{\mathsf{sse},i}, \ell_\rho)$, sample a fresh key $K'_{\ell_\rho,i} \xleftarrow{\$} \mathcal{K}$ at random in the key space, and return the ciphertext $(E_{\ell_\rho,i} = \mathsf{SEnc}(K'_{\ell_\rho,i}, C_{\ell_\rho,i}), D_{\ell_\rho,i}, S_{\ell_\rho,i})$.

Now, we build PPT adversaries $\mathcal{B}_{\rho,0}$ and $\mathcal{B}_{\rho,1}$ against the $\mathtt{1\text{-}Label\text{-}IND}$-security of the SSE such that

$$|\mathsf{Adv}_{\mathbf{G}^*_{\rho-1}}(\mathcal{A} \wedge i^* \neq 0) - \mathsf{Adv}_{\mathbf{H}_{\rho,0}}(\mathcal{A} \wedge i^* \neq 0)| \leq \mathsf{Adv}^{\mathtt{1\text{-}Label\text{-}IND}}_{\mathsf{SSE}}(\mathcal{B}_{\rho,0});$$
$$|\mathsf{Adv}_{\mathbf{G}^*_{\rho}}(\mathcal{A} \wedge i^* \neq 0) - \mathsf{Adv}_{\mathbf{H}_{\rho,1}}(\mathcal{A} \wedge i^* \neq 0)| \leq \mathsf{Adv}^{\mathtt{1\text{-}Label\text{-}IND}}_{\mathsf{SSE}}(\mathcal{B}_{\rho,1}).$$

Let $\beta \in \{0,1\}$. We proceed to describe $\mathcal{B}_{\rho,\beta}$. First, $\mathcal{B}_{\rho,\beta}$ samples the guess $i^* \xleftarrow{\$} \{0, \ldots, n\}$. If $i^* = 0$, then $\mathcal{B}_{\rho,\beta}$ behaves exactly as the challenger in the game $\mathbf{G}^*_{\rho-1+\beta}$. Otherwise, it does the following, using the $\mathtt{1\text{-}Label\text{-}IND}$-security game against SSE:

- Initialize: it generates $(\mathsf{mpk}, \mathsf{msk}, (\mathsf{ek}_i)_{i\in[n]}) \leftarrow \mathsf{SetUp}(\lambda)$, and sends i^* to receive $\mathsf{pk}_{\mathsf{sse}}$ from its own $\mathtt{1\text{-}Label\text{-}IND}$ challenger for SSE. It returns $\mathsf{mpk}' = \mathsf{mpk} \cup \mathsf{pk}_{\mathsf{sse}}$ to the adversary $\mathcal{A}$;
- $\mathsf{QEncrypt}'(i, x, \ell_j)$: it can compute $C_{\ell_j,i} \leftarrow \mathsf{Encrypt}(\mathsf{ek}_i, x, \ell_j)$. Then, it call its own oracle to get $S_{\ell_j,i} \leftarrow \mathsf{QShare}(i, \ell_j)$. If $j \neq \rho$, it computes $(D_{\ell_j,i}, K_{\ell_j,i}) \leftarrow \mathsf{SSE.Encaps}(\mathsf{pk}_{\mathsf{sse}}, \ell_j)$; if $j = \rho$, it calls $(D_{\ell_\rho,i}, K_{\ell_\rho,i}) \leftarrow \mathsf{QRealRandom}(\ell_\rho)$. Eventually, it returns the ciphertext $(E_{\ell_j,i} = \mathsf{SEnc}(K_{\ell_j,i}, C_{\ell_j,i}), D_{\ell_j,i}, S_{\ell_j,i})$;
- $\mathsf{QLeftRight}'(i, x^0, x^1, \ell_j)$: if $j < \rho$, it computes $C_{\ell_j,i} = \mathsf{Encrypt}(\mathsf{ek}_i, x^1, \ell_j)$; if $j > \rho$, it computes $C_{\ell_j,i} = \mathsf{Encrypt}(\mathsf{ek}_i, x^0, \ell_j)$; and if $j = \rho$, it computes $C_{\ell_j,i} = \mathsf{Encrypt}(\mathsf{ek}_i, x^\beta, \ell_j)$. Then it calls its own oracle to get $S_{\ell_j,i} = \mathsf{QShare}(i, \ell_j)$. If $j \neq \rho$, it computes $(D_{\ell_j,i}, K_{\ell_j,i}) \leftarrow \mathsf{SSE.Encaps}(\mathsf{pk}_{\mathsf{sse}}, \ell_j)$; if $j = \rho$, it calls $(D_{\ell_\rho,i}, K_{\ell_\rho,i}) \leftarrow \mathsf{QRealRandom}(\ell_\rho)$. Eventually, it returns the ciphertext $(E_{\ell_j,i} = \mathsf{SEnc}(K_{\ell_j,i}, C_{\ell_j,i}), D_{\ell_j,i}, S_{\ell_j,i})$;

– $\mathsf{QDKeyGen}'(f)$: it runs and returns $\mathsf{DKeyGen}(\mathsf{msk}, f)$.
– $\mathsf{QCorrupt}'(i)$: it uses its own corruption oracle to get $\mathsf{ek}_{\mathsf{sse},i} \leftarrow \mathsf{QCorrupt}(i)$, and returns $\mathsf{ek}'_i = \mathsf{ek}_i \cup \mathsf{ek}_{\mathsf{sse},i}$;
– Finalize: $\mathcal{B}_{\rho,\beta}$ checks whether the ciphertext for the i^*-th client has been asked under ℓ_ρ, or corrupted. If so, it ignores $\mathcal{A}$'s guess and sends a uniformly random bit $\beta \xleftarrow{\$} \{0,1\}$; Otherwise, it forwards $\mathcal{A}$'s guess to the Finalize procedure of the `IND`*-security game.

Game $\mathbf{G}_\rho$, which encrypts x^1 under ℓ_ρ, just differs from $\mathbf{H}_{\rho,1}$ with real vs. random keys K_{ℓ_ρ}, as emulated by $\mathcal{B}_{\rho,1}$, according to the real-or-random behavior of the `1-Label-IND` game for SSE. Game $\mathbf{G}_{\rho-1}$, which encrypts x^0 under ℓ_ρ, just differs from $\mathbf{H}_{\rho,0}$ with real vs. random keys K_{ℓ_ρ}, as emulated by $\mathcal{B}_{\rho,0}$, according to the real-or-random behavior of the `1-Label-IND` game for SSE. Note that if adversary $\mathcal{A}$ makes queries that satisfy the conditions required by the Finalize procedure from the game `IND`* of MCFE′, and that the guess $i^* \neq 0$ is correct, then the queries of $\mathcal{B}_{\rho,\beta}$ satisfy the conditions required by the `1-Label-IND` security game for SSE, namely, QRealRandom is only queried on one label ℓ_ρ, QCorrupt is never queried on i^*, and QShare is never queried on (i^*, ℓ_ρ).

Note that in the case the guess $i^* \neq 0$ is correct, in the `IND`-security game, the adversary can ask QLeftRight queries on users $i \in \mathcal{CS}$ for the label ℓ_ρ, which was not allowed in the original `IND`*-security game (and would lead to a random output). The reason is that security here relies on the `1-Label-IND` security of the SSE for the label ℓ_ρ. In the simulation, no matter what is encrypted in $C_{\ell_\rho,i}$ under the label ℓ_ρ, the QRealRandom algorithm provides randomness and makes the ciphertexts $C_{\ell_\rho,i}$ impossible to recover.

Since the encapsulation keys K_{ℓ_ρ} are uniformly random in games $\mathbf{H}_{\rho,0}$ and $\mathbf{H}_{\rho,1}$, we can use the one-time security of SKE, for each ciphertext for the label ℓ_ρ, to obtain a PPT adversary $\mathcal{B}''_\rho$ such that:

$$|\mathsf{Adv}_{\mathbf{H}_{\rho,0}}(\mathcal{A} \wedge i^* \neq 0) - \mathsf{Adv}_{\mathbf{H}_{\rho,1}}(\mathcal{A} \wedge i^* \neq 0)| \leq q_e \cdot \mathsf{Adv}^{\mathsf{OT}}_{\mathsf{SKE}}(\mathcal{B}''_\rho),$$

where q_e denotes maximum number of ciphertexts generated by the QLeftRight oracle for a given label.

Putting everything together, for the case $i^* \neq 0$, we obtain PPT adversaries $\mathcal{B}'_\rho$ and $\mathcal{B}''_\rho$ such that: $|\mathsf{Adv}_{\mathbf{G}^*_{\rho-1}}(\mathcal{A} \wedge i^* \neq 0) - \mathsf{Adv}_{\mathbf{G}^*_\rho}(\mathcal{A} \wedge i^* \neq 0)|$ is upper-bounded by $2 \cdot \mathsf{Adv}^{\mathtt{1\text{-}Label\text{-}IND}}_{\mathsf{SSE}}(\mathcal{B}'_\rho) + q_e \cdot \mathsf{Adv}^{\mathsf{OT}}_{\mathsf{SKE}}(\mathcal{B}''_\rho))$. Since for any game $\mathbf{G}$ and any adversary $\mathcal{A}$, $\mathsf{Adv}_{\mathbf{G}}(\mathcal{A}) = \mathsf{Adv}_{\mathbf{G}}(\mathcal{A} \wedge i^* = 0) + \mathsf{Adv}_{\mathbf{G}}(\mathcal{A} \wedge i^* \neq 0)$, this concludes the proof of Lemma 13.

6 IP-MCFE with Repetitions

In this section, we add a layer of IP-FE on top of the IP-MCFE from [CDG+18], to remove the restriction of having a unique challenge ciphertext per client and per label. Our construction works for any IP-FE that is compatible with the IP-MCFE from [CDG+18], namely, an IP-FE whose message space is the ciphertext space of the IP-MCFE. For correctness, we exploit the fact that decryption of the IP-MCFE computes the inner product of the ciphertext together with the decryption keys. For security, we exploit the fact that the IP-MCFE is linearly homomorphic, in the sense that given an input $\vec{x}$, one can publicly maul an encryption of $\vec{x}'$ into an encryption of $\vec{x}+\vec{x}'$. This is used to bootstrap the security from one to many challenge ciphertexts per (user,label) pair, similarly to [AGRW17, ACF+18] in the context of multi-input IP-FE. In fact, [ACF+18] uses a one-time secure multi-input FE as inner layer, and a single-input IP-FE as outer layer, while we use an IP-MCFE as inner layer, and an IP-FE as outer layer. The main technical challenge is to handle the case of (adaptive) corruptions, which are not considered in [AGRW17, ACF+18] (even in the static case where corruptions are known beforehand).

We first recall the IP-MCFE from [CDG+18] extended to handle vectors as inputs of the encryption algorithm. Also, we make use of the fact that the encryption algorithm can act on vectors of group elements, in $\mathbb{G}^m$, where $\mathbb{G}$ is a prime-order group, as opposed to vectors over

$\mathbb{Z}$. Decryption recovers the inner product in the group $\mathbb{G}$, without any restriction on the size of the input of the encryption and decryption key generation algorithms. Namely, the message space of $\mathsf{IP\text{-}FE}$ is $\mathbb{G}^m$,for some dimension m, its decryption key space is $\mathbb{Z}_p^m$, where p is the order of $\mathbb{G}$, and for any $[\vec{x}] \in \mathbb{G}^m$, $\vec{y} \in \mathbb{Z}_p^m$, $\mathsf{IP.Dec}(\mathsf{ct}, \mathsf{dk}_{\vec{y}}) = [\vec{x}^\top \vec{y}]$ with probability one, where $\mathsf{ct} \leftarrow \mathsf{IP.Encrypt}(\mathsf{IP.msk}, [\vec{x}])$, $\mathsf{dk}_{\vec{y}} \leftarrow \mathsf{IP.DKeyGen}(\mathsf{IP.msk}, \vec{y})$, and $(\mathsf{IP.mpk}, \mathsf{IP.msk}) \leftarrow \mathsf{IP.SetUp}(\lambda)$. for Then we give our generic construction to obtain security with repetitions.

6.1 Reproduction of the IP-MCFE from [CDG+18]

In [CDG+18], Chotard *et al.* proposed an $\mathtt{IND}^*$-secure $\mathsf{IP\text{-}MCFE}$. Roughly speaking, it relies on a private-key variant of Agrawal *et al.* [ALS16] $\mathsf{IP\text{-}FE}$, where a random oracle is used to generate common randomness among the different users, that is used to produce the ciphertexts. We extend it to handle vector-inputs for each client, instead of just scalars, and to correspond to Definition 1 given in Section 2.

- $\mathsf{SetUp}(\lambda)$: samples $\mathcal{G} := (\mathbb{G}, p, P) \overset{\$}{\leftarrow} \mathsf{GGen}(1^\lambda)$, a full-domain hash function $\mathcal{H}$ onto $\mathbb{G}^2$, $\mathbf{S}_i \overset{\$}{\leftarrow} \mathbb{Z}_p^{m\times 2}$, for $i = 1, \ldots, n$. Returns the public key $\mathsf{mpk} := (\mathcal{G}, \mathcal{H})$, encryption keys $\mathsf{ek}_i = \mathbf{S}_i$ for $i = 1, \ldots, n$, and the master secret key $\mathsf{msk} = ((\mathbf{S}_i)_i)$, (in addition to mpk, which is omitted);
- $\mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i, \ell)$: Takes as input the value $\vec{x}_i \in \mathbb{Z}_p^m$ to encrypt, under the key $\mathsf{ek}_i = \mathbf{S}_i$ and the label ℓ. It computes $[\vec{u}_\ell] := \mathcal{H}(\ell) \in \mathbb{G}^2$, and outputs the ciphertext $[\vec{c}_i] = [\mathbf{S}_i\vec{u}_\ell + \vec{x}_i] \in \mathbb{G}^m$;
- $\mathsf{DKeyGen}(\mathsf{msk}, \vec{y})$: Takes as input $\mathsf{msk} = (\mathbf{S}_i)_i$ and an inner-product function defined by $\vec{y} \in \mathbb{Z}_p^{m\cdot n}$ as $f_{\vec{y}}(\vec{x}) = \langle \vec{x}, \vec{y}\rangle$,where $\vec{x} = (\vec{x}_1 \| \cdots \| \vec{x}_n) \in \mathbb{Z}_p^{nm}$, and outputs the functional decryption key $\mathsf{dk}_{\vec{y}} = \big(\vec{y}, \sum_i \mathbf{S}_i^\top \vec{y}_i\big) \in \mathbb{Z}_p^{mn} \times \mathbb{Z}_p^2$;
- $\mathsf{FDecrypt}(\mathsf{dk}_{\vec{y}}, \ell, ([\vec{c}_i])_{i\in[n]})$: Takes as input a functional decryption key $\mathsf{dk}_{\vec{y}} = (\vec{y}, \vec{d})$, a label ℓ, and ciphertexts. It computes $[\vec{u}_\ell] := \mathcal{H}(\ell)$ and returns $[\alpha] = \sum_i [\vec{c}_i]^\top \vec{y}_i - [\vec{u}_\ell]^\top \vec{d}$;
- $\mathsf{Decrypt}(\mathsf{ek}_i, \ell, ([\vec{c}_i])_{i\in[n]})$: Takes as input the encryption key $\mathsf{ek}_i = \mathbf{S}_i$, a label ℓ, and ciphertexts. It computes $[\vec{u}_\ell] := \mathcal{H}(\ell)$ and returns $[\vec{x}_{\ell,i}] = [\vec{c}_i] - [\mathbf{S}_i\vec{u}_\ell]$.

For correctness, one can check that:

$$\begin{aligned}[\alpha] &= \sum_i [\vec{c}_i]^\top \vec{y}_i - [\vec{u}_\ell]^\top \vec{d} = \sum_i [\mathbf{S}_i\vec{u}_\ell + \vec{x}_i]^\top \vec{y}_i - [\vec{u}_\ell]^\top \sum_i \mathbf{S}_i^\top \vec{y}_i \\ &= \sum_i [\mathbf{S}_i\vec{u}_\ell]^\top \vec{y}_i + [\vec{x}_i]^\top \vec{y}_i - \sum_i [\mathbf{S}_i\vec{u}_\ell]^\top \vec{y}_i = \sum_i [\vec{x}_i]^\top \vec{y}_i = [\vec{x}^\top \vec{y}] = [\langle \vec{x}, \vec{y}\rangle].\end{aligned}$$

For security, we will use the two following properties of the $\mathsf{IP\text{-}MCFE}$ from [CDG+18]:

- *Linear Homomorphism of ciphertexts*: for any $i \in [n]$, $\vec{x}_i, \vec{x}'_i \in \mathbb{Z}_p$, and any label ℓ, we have $[\vec{c}_i] + [\vec{x}'_i] = \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i + \vec{x}'_i, \ell)$, where $[\vec{c}_i] = \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i, \ell)$.
- *Deterministic Encryption.* In particular, together with the linear homomorphism of ciphertexts, this implies that for any $\vec{x}_i, \vec{x}'_i \in \mathbb{Z}_p^m$ and any label ℓ, we have: $\mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i, \ell) - \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}'_i, \ell) = [\vec{x}_i - \vec{x}'_i]$.

Security of the IP-MCFE from [CDG+18]. The security notion proven in [CDG+18] slightly differs from the $\mathtt{IND}^*$ security notion we need here, in that it does not give the adversary access to a $\mathsf{QEncrypt}$ oracle, but only $\mathsf{QLeftRight}$ (see Remark 3 on the role of the oracle $\mathsf{QEncrypt}$, and why it cannot be simulated by $\mathsf{QLeftRight}$ in the $\mathtt{IND}^*$ security game). It follows from inspection of the security proof of [CDG+18] that it can actually achieve the $\mathtt{IND}^*$ security notion defined here. Here, we give intuitive arguments of why this holds.s As explained in remark Remark 3, the only difference between $\mathsf{QEncrypt}(i, \vec{x}_i, \ell)$ and $\mathsf{QLeftRight}(i, \vec{x}_i, \vec{x}_i, \ell)$ is that $\mathsf{QEncrypt}$ allows the adversary to make incomplete ciphertext queries, not $\mathsf{QLeftRight}$. In the security proof, the challenge ciphertexts output by $\mathsf{QLeftRight}$ are switched from an encryption of $\vec{x}_i^0$ to encryption of $\vec{x}_i^1$, introducing a delta terms in the functional decryption keys. This delta term cancels out thanks to the conditions given in the Finalize procedure (for which we need complete ciphertexts). For $\mathsf{QEncrypt}$ queries, $\vec{x}_i^0 = \vec{x}_i^1 = \vec{x}_i$, which means the delta term is zero, and doesn't show up in any of the functional decryption key even for incomplete ciphertexts.

6.2 Construction of IND-Secure IP-MCFE with Repetitions

Let $\mathsf{MCFE} = (\mathsf{SetUp}, \mathsf{Encrypt}, \mathsf{DKeyGen}, \mathsf{FDecrypt}, \mathsf{Decrypt})$ be the above $\mathsf{IP\text{-}MCFE}$ scheme, and $\mathsf{IP\text{-}FE} = (\mathsf{IP.SetUp}, \mathsf{IP.Encrypt}, \mathsf{IP.DKeyGen}, \mathsf{IP.Dec})$ be a single-input Inner Product FE (as defined in Section 3.3) whose message space is the ciphertext space of MCFE. We define a new $\mathsf{MCFE}' = (\mathsf{SetUp}', \mathsf{Encrypt}', \mathsf{DKeyGen}', \mathsf{Decrypt}')$ as follows:

– $\mathsf{SetUp}'(\lambda)$: It executes $(\mathsf{mpk}, \mathsf{msk}, (\mathsf{ek}_i)_i) \leftarrow \mathsf{SetUp}(\lambda)$ as well as, for $i = 1, \ldots, n$, $(\mathsf{IP.mpk}_i, \mathsf{IP.msk}_i) \leftarrow \mathsf{IP.SetUp}(\lambda)$. The encryption keys are $\mathsf{ek}'_i = (\mathsf{ek}_i, \mathsf{IP.msk}_i)$ for all $i = 1, \ldots, n$, the public key is $\mathsf{mpk}' := (\mathsf{mpk}, \{\mathsf{IP.mpk}_i\}_i)$, and the master secret key is $\mathsf{msk}' = (\mathsf{msk}, \{\mathsf{IP.msk}_i\}_i)$;
– $\mathsf{Encrypt}'(\mathsf{ek}'_i, \vec{x}_i, \ell)$: It parses the encryption key ek'_i as $(\mathsf{ek}_i, \mathsf{IP.msk}_i)$, runs $[\vec{c}_{i,\ell}] \leftarrow \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i, \ell)$, and returns $C'_{\ell,i} := \mathsf{IP.Encrypt}(\mathsf{IP.msk}_i, [\vec{c}_{i,\ell}])$;
– $\mathsf{DKeyGen}'(\mathsf{msk}', \vec{y})$: on input $\vec{y} := (\vec{y}_1 \| \cdots \| \vec{y}_n) \in \mathbb{Z}_p^{nm}$, it computes $\mathsf{dk}_{\vec{y}} = \mathsf{DKeyGen}(\mathsf{msk}, \vec{y})$, and for all $i \in [n]$: $\mathsf{dk}_{\vec{y}_i} = \mathsf{IP.DKeyGen}(\mathsf{IP.msk}_i, \vec{y}_i)$. It returns $\mathsf{dk}'_{\vec{y}} = \big(\mathsf{dk}_{\vec{y}}, \{\mathsf{dk}_{\vec{y}_i}\}_{i\in[n]}\big)$.

The three above algorithms are enough to show the security (as proven below), which holds with respect to any $\mathsf{IP\text{-}MCFE}$ that satisfies the Linearly Homomorphism of ciphertexts, and deterministic encryption, as defined above. However, correctness only holds for the particular $\mathsf{IP\text{-}MCFE}$ from [CDG+18], where decryption computes the inner product between ciphertexts and decryption keys. That prevents from a generic transformation.

We now prove correctness when using the $\mathsf{IP\text{-}MCFE}$ from [CDG+18] in MCFE':

– $\mathsf{FDecrypt}'(\mathsf{dk}'_{\vec{y}}, \ell, (C'_{\ell,i})_{i\in[n]})$: Takes as input a functional decryption key $\mathsf{dk}'_{\vec{y}} = \big(\mathsf{dk}_{\vec{y}}, \{\mathsf{dk}_{\vec{y}_i}\}_{i\in[n]}\big)$, where $\mathsf{dk}_{\vec{y}} = (\vec{y}, \vec{d} = \sum_i \mathbf{S}_i^\top \vec{y}_i)$, a label ℓ, and ciphertexts $(C'_{\ell,i})_{i\in[n]}$. First, it computes $[d_{i,\ell}] = \mathsf{IP.Dec}(\mathsf{dk}_{\vec{y}_i}, C'_{\ell,i})$ for all $i \in [n]$. Then it computes $[\vec{u}_\ell] = \mathcal{H}(\ell)$, and computes $[\alpha] = [\sum_i d_{i,\ell}] - \vec{d}^\top [\vec{u}_\ell]$. Finally, it returns the discrete logarithm $\alpha \in \mathbb{Z}_p$;
– $\mathsf{Decrypt}'(\mathsf{ek}'_i, \ell, (C'_{\ell,i})_{i\in[n]})$: Takes as input an encryption key $\mathsf{ek}'_i = (\mathsf{ek}_i, \mathsf{IP.msk}_i)$, a label ℓ, and ciphertexts $(C'_{\ell,i})_{i\in[n]}$. For each $k \in [m]$, computes $\mathsf{dk}_k = \mathsf{IP.DKeyGen}(\mathsf{IP.msk}_i, (\delta_{k,j})_j)$, then $[c_i] = \mathsf{IP.Dec}(\mathsf{dk}_k, C'_{\ell,i})$, and finally $\vec{x}_i = \mathsf{Decrypt}(\mathsf{ek}_i, \ell, [\vec{c}_i])$.

Correctness. By correctness of the $\mathsf{IP\text{-}FE}$, we have for all $i \in [n]$, and any label ℓ: $[d_{i,\ell}] = [\langle \vec{y}_i, \vec{x}_i + \mathbf{S}_i \vec{u}_\ell \rangle] = [\langle \vec{y}_i, \vec{x}_i \rangle] + \langle \vec{y}_i, \mathbf{S}_i \rangle \cdot [\vec{u}_\ell]$. Thus, $\sum_i [d_{i,\ell}] = [\langle \vec{y}, \vec{x} \rangle] + (\sum_i \vec{y}_i^\top \mathbf{S}_i) \cdot [\vec{u}_\ell]$. Since $\vec{d} = \sum_i \mathbf{S}_i^\top \vec{y}_i$, we have $\sum_i [d_{i,\ell}] = [\langle \vec{y}, \vec{x} \rangle] + \vec{d}^\top [\vec{u}_\ell]$, hence $\alpha = \langle \vec{x}, \vec{y} \rangle$.

In Appendix B, we provide the proof that the MCFE' described above achieves `1-Label-IND`*-security, using the `wtr-IND`*-security of the MCFE from [CDG+18], assuming the $\mathsf{IP\text{-}FE}$ is `IND`-secure (concrete instances of which are given in [ALS16]).

7 Chosen-Ciphertext Security

In this section we briefly describe how to achieve Chosen-Ciphertext (`CCA`) security in a black-box way, where the adversary may have access to functional decryption queries, without learning the functional decryption key, but just the result. Indeed, while using public-key cryptography techniques, MCFE remains a symmetric-key primitive, where the client encryption key not only allows encryption, but decryption. Using the additional decryption algorithm, we show that adding a signature simply provides `CCA`-security.

More precisely, given an `IND`-secure MCFE scheme $\mathsf{MCFE} = (\mathsf{MCFE.SetUp}, \mathsf{MCFE.Encrypt}, \mathsf{MCFE.KeyGen}, \mathsf{MCFE.FDecrypt}, \mathsf{MCFE.Decrypt})$, a strongly unforgeable signature $(\mathsf{S.SetUp}, \mathsf{S.Sign}, \mathsf{S.Verif})$, we generically build an `IND-CCA`-secure scheme $\widetilde{\mathsf{MCFE}}$, where the changes are as follows:

- $\widetilde{\mathsf{MCFE}}.\mathsf{SetUp}(\lambda)$: Executes $(\mathsf{msk}, (\mathsf{ek}_i)_i, \mathsf{pp}) \leftarrow \mathsf{MCFE}.\mathsf{SetUp}(\lambda)$ and, for any $i \in [n]$, sets the signature parameters $(\mathsf{S.key}_i, \mathsf{S.pp}_i) \leftarrow \mathsf{S.SetUp}(\lambda)$ for each user i. $\widetilde{\mathsf{msk}} = (\mathsf{msk}, (\mathsf{S.key}_i)_i)$, for all i sets $\widetilde{\mathsf{ek}}_i = (\mathsf{ek}_i, \mathsf{S.key}_i)$, and finally $\widetilde{\mathsf{pp}} = (\mathsf{pp}, (\mathsf{S.pp}_i)_i)$;
- $\widetilde{\mathsf{MCFE}}.\mathsf{Encrypt}(\widetilde{\mathsf{ek}}_i, x_i, \ell)$: Computes $A_{\ell,i} \leftarrow \mathsf{MCFE}.\mathsf{Encrypt}(\mathsf{ek}_i, x_i, \ell)$ and the signature $S_{\ell,i} \leftarrow \mathsf{S.Sign}(\mathsf{S.key}_i, (A_{\ell,i}, \ell))$ to return $C_{\ell,i} = (A_{\ell,i}, S_{\ell,i})$;
- $\widetilde{\mathsf{MCFE}}.\mathsf{FDecrypt}(\mathsf{dk}_f, \ell, \vec{C})$: Given a ciphertext $\vec{C} = (A_i, S_i)_i$, computes the values $\mathsf{S.Verif}(\mathsf{S.pp}_i, (A_i, \ell), S_i)$ for any user i. If all the signatures are valid, returns $\alpha \leftarrow \mathsf{MCFE}.\mathsf{FDecrypt}(\mathsf{dk}_f, \ell, (A_i)_i)$, otherwise, returns $\perp$;
- $\widetilde{\mathsf{MCFE}}.\mathsf{Decrypt}(\widetilde{\mathsf{ek}}_i, \ell, \vec{C})$: Given a ciphertext $\vec{C} = (A_i, S_i)_i$, computes the values $\mathsf{S.Verif}(\mathsf{S.pp}_i, (A_i, \ell), S_i)$ for any user i. If all the signatures are valid, returns $x_i \leftarrow \mathsf{MCFE}.\mathsf{Decrypt}(\mathsf{ek}_i, \ell, (A_i)_i)$, otherwise, returns $\perp$;

$\widetilde{\mathsf{MCFE}}.\mathsf{DKeyGen}$ remains the same as $\mathsf{MCFE}.\mathsf{DKeyGen}$. In Appendix C, we provide a full security proof.

8 DMCFE from MCFE without Pairings

8.1 Decentralized Multi-Client Functional Encryption

In [CDG+18], Chotard *et al.* defined the notion of DMCFE, where the generation of the functional decryption keys is distributed among the clients, so that they keep control on these keys. For efficiency reasons, they focused on efficient one-round key generation protocols DKeyGen that can be split in a first step DKeyGenShare that generates partial keys and the combining algorithm DKeyComb that combines partial keys into the functional decryption key. The full definition can be found in [CDG+18], and we briefly recall it here for completeness.

Definition 14 (Decentralized Multi-Client Functional Encryption). *A decentralized multi-client functional encryption on $\mathcal{M}$ between a set of n senders $(\mathcal{S}_i)_i$, for $i = 1, \ldots, n$, and a functional decrypter $\mathcal{FD}$ is defined by the setup protocol and four algorithms:*

- $\mathsf{SetUp}(\lambda)$*: This is a protocol between the senders $(\mathcal{S}_i)_i$ that generate their own secret keys sk_i and encryption keys ek_i, and eventually output the public parameters mpk;*
- $\mathsf{Encrypt}(\mathsf{ek}_i, x_i, \ell)$*: Takes as input a user encryption key ek_i, a value x_i to encrypt, and a label ℓ, and outputs the ciphertext $C_{\ell,i}$;*
- $\mathsf{DKeyGenShare}(\mathsf{sk}_i, \ell_f)$*: Takes as input a user secret key sk_i and a label ℓ_f, and outputs the partial functional decryption key $\mathsf{dk}_{f,i}$ for a function $f : \mathcal{M}^n \to \mathcal{R}$ that is described in ℓ_f;*
- $\mathsf{DKeyComb}((\mathsf{dk}_{f,i})_i, \ell_f)$*: Takes as input the partial functional decryption keys and eventually outputs the functional decryption key dk_f;*
- $\mathsf{FDecrypt}(\mathsf{dk}_f, \ell, \vec{C})$*: Takes as input a functional decryption key dk_f, a label ℓ, and an n-vector ciphertext $\vec{C}$, and outputs $f(\vec{x})$, if $\vec{C}$ is a valid encryption of $\vec{x} = (x_i)_i \in \mathcal{M}^n$ for the label ℓ, or $\perp$ otherwise;*
- $\mathsf{Decrypt}(\mathsf{ek}_i, \ell, \vec{C})$*: Takes as input an encryption key ek_i, a label ℓ, and an n-vector ciphertext $\vec{C}$, and outputs x_i, if $\vec{C}$ is a valid encryption of $x_i \in \mathcal{M}$ for the label ℓ, or $\perp$ otherwise;*

The correctness property essentially states the combined key corresponds to the functional decryption key. The security model is quite similar to the previous one for MCFE (see Definition 2), except that

- for the DKeyGen protocol: the adversary has access to transcripts of the communications, thus modeled by a query $\mathsf{QDKeyGen}(i, f)$ that executes $\mathsf{DKeyGenShare}(\mathsf{sk}_i, \ell_f)$, where ℓ_f is a description of f;
- corruption queries additionally reveal the secret keys sk_i;

– the Finalize procedure ignores incomplete functional decryption keys: for condition (2), only functions f for which all the honest key-shares have been asked are considered.

The critical point is the last one: the distributed key generation must guarantee that without all the shares, no information is known about the functional decryption key. In addition, the protocol must be efficient.

8.2 Distributed Sum

In order to convert an MCFE scheme into a DMCFE, one needs to allow efficient distributed computation of the functional decryption key. In many cases, this can be seen as a particular MCFE for the unique sum function on the contributions of all the clients. As an example, for the IP-MCFE from [CDG+18], $\mathsf{dk}_{\vec{y}} = \left(\vec{y}, \sum_i \mathbf{S}_i^\top \vec{y}_i\right)$, and namely one has to compute $\sum_i x_i = \sum_i \mathbf{S}_i^\top \vec{y}_i$, where the x_i's can be computed by each client.

In this section, we thus focus on the functionality of publishing the sum of individual secrets, in an efficient manner.

Definition 15 (Ideal Protocol DSum). *A DSum on a group $\boldsymbol{G}$ among n senders is defined by three algorithms:*

– $\mathsf{DSum.SetUp}(\lambda)$*: Takes as input the security parameter λ. Generates the public parameters* pp *and the personal secret keys* sk_i *for $i = 1 \cdots n$;*
– $\mathsf{DSum.Encode}(x_i, \ell, \mathsf{sk}_i)$*: Takes the x_i value to encode, a label ℓ and the personal secret key* sk_i *of the user i. Returns the share $M_{\ell,i}$*
– $\mathsf{DSum.Combine}(\vec{M})$*: Takes as input a vector $\vec{M} = (M_{\ell,i})_i$ of shares. Returns the value $\sum_i M_{\ell,i}$;*

Correctness. For any label ℓ, we want $\Pr[\mathsf{DSum.Combine}(\vec{M}_\ell) = \sum_i x_i] = 1$, where the probability is taken over $M_{\ell,i} \leftarrow \mathsf{DSum.Encode}(x_i, \ell, \mathsf{sk}_i)$ for all $i \in [n]$, and $(\mathsf{pp}, (\mathsf{sk}_i)_i) \leftarrow \mathsf{DSum.SetUp}(\lambda)$.

Security Notion. This protocol must guarantee the privacy of the x_i's, their sum possibly excepted when all the shares are known. This is the classical security notion for multi-party computation, where the security proof is performed by simulating the view of the adversary from the output of the result: nothing when not all the shares are asked, and just the sum of the inputs when all the shares are queried. We also have to deal with the corruptions, which give the users' secret keys.

8.3 DSum Protocol in the Random Oracle Model

The protocol below is similar to [KDK11], with a hash function. We provide a new security analysis, which relies on the CDH problem in the Random Oracle Model, given in Appendix D.

– $\mathsf{DSum.SetUp}(\lambda)$: Takes as input the security parameter λ and generates a group $\mathbb{G}$ of prime order p, with a generator g, were the CDH assumption holds. It also generates a hash function $\mathcal{H} : \{0,1\}^* \to \mathbf{G}$, for any group $\mathbf{G}$, denoted additively. Each user i, picks $t_i \xleftarrow{\$} \mathbb{Z}_p$. The public parameters pp are $(\mathbb{G}, p, g, \mathcal{H}, ([t_i])_i)$ and the personal secret keys $\mathsf{sk}_i = t_i$ for $i = 1 \cdots n$ (with the public parameters);
– $\mathsf{DSum.Encode}(x_i, \ell, \mathsf{sk}_i)$: Takes the x_i value to encode, a label ℓ and the personal secret key $\mathsf{sk}_i = t_i$ of the user i, it returns $M_{\ell,i}$ computed as below, where $h_{\ell,i,j} = \mathcal{H}([t_{\min\{i,j\}}], [t_{\max\{i,j\}}], t_i \cdot [t_j], \ell) = h_{\ell,j,i}$: $M_{\ell,i} = x_i - \sum_{j<i} h_{\ell,i,j} + \sum_{j>i} h_{\ell,i,j}$;
– $\mathsf{DSum.Combine}(\vec{M} = (M_{\ell,i})_i)$: Takes as input a vector $\vec{M}$ of shares. Computes and return the value $\sum_i M_{\ell,i}$.

Correctness. The correctness should show that the sum of the shares is equal to the sum of the x_i's: the former is equal to

$$\sum_i \left(x_i - \sum_{j<i} h_{\ell,i,j} + \sum_{j>i} h_{\ell,i,j} \right) = \sum_i x_i - \sum_i \sum_{j<i} h_{\ell,i,j} + \sum_i \sum_{j>i} h_{\ell,j,i}$$
$$= \sum_i x_i - \sum_i \sum_{j<i} h_{\ell,i,j} + \sum_j \sum_{i<j} h_{\ell,j,i} = \sum_i x_i$$

8.4 DSum Protocol in the Standard Model

A variant of this protocol can also be described with a randomness extractor and a PRF. We then provide the security analysis under the DDH assumption and the PRF indistinguishability. More precisely, for the randomness extractor, we can use the Left-over-Hash-Lemma [ILL89, HILL99], with a random seed k in the CRS to extract random keys K for a PRF $(\mathcal{F}_K)_K$, with a universal hash function $(H_k)_k$:

– $\mathsf{DSum.SetUp}(\lambda)$: Takes as input the security parameter λ and generates a group $\mathbb{G}$ of prime order p, with a generator g. From a family of universal hash functions $(H_k)_k$ and a random key k, this define the randomness extractor $\mathcal{E}(\cdot) = H_k(\cdot)$, later used to generate the keys K of a PRF $(\mathcal{F}_K)_K$. Each user i, picks $t_i \overset{\$}{\leftarrow} \mathbb{Z}_p$. The public parameters pp are $(\mathbb{G}, p, g, \mathcal{E}, (\mathcal{F}_K)_K, ([t_i])_i)$ and the personal secret keys $\mathsf{sk}_i = t_i$ for $i = 1 \cdots n$ (with the public parameters);
– $\mathsf{DSum.Encode}(x_i, \ell, \mathsf{sk}_i)$: Takes the x_i value to encode, a label ℓ and the personal secret key $\mathsf{sk}_i = t_i$ of the user i, it returns $M_{\ell,i}$ computed as below, where $h_{\ell,i,j} = \mathcal{F}_{K_{i,j}}(\ell)$ with $K_{i,j} = \mathcal{E}(t_i \cdot [t_j])$: $M_{\ell,i} = x_i - \sum_{j<i} h_{\ell,i,j} + \sum_{j>i} h_{\ell,i,j}$;
– $\mathsf{DSum.Combine}(\vec{M} = (M_{\ell,i})_i)$: Takes as input a vector $\vec{M}$ of shares. Computes and return the value $\sum_i M_{\ell,i}$.

The correctness is the same as above, since it just makes use of $h_{\ell,i,j}$. The security however requires the DDH assumption, in order to guarantee the randomness of all the Diffie-Hellman values $[t_i \cdot t_j]$. The Left-over-Hash Lemma thereafter ensures the uniform and independent distributions of the $K_{i,j}$'s which then make the $h_{\ell,i,j}$'s unpredictable for all the honest i, j. Again, the details of this proof are given in Appendix D.

8.5 Application to IP-DMCFE

One can generically convert an IP-MCFE into an IP-DMCFE, when $\mathsf{dk}_{\vec{y}} = \left(\vec{y}, \vec{d_{\vec{y}}}\right)$, where $\vec{d_{\vec{y}}} = \sum_i x_i$, with the x_i's computed by each client, as $x_i \leftarrow \mathbf{S}_i^\top \vec{y_i}$ in [CDG+18], by letting the clients generating the DSum secret keys at the setup time, and the label is the vector $\vec{y}$:

– $\mathsf{DKeyGenShare}(\mathsf{sk}_i, \vec{y})$: outputs $M_{\vec{y},i} \leftarrow \mathsf{DSum.Encode}(x_i, \vec{y}, \mathsf{sk}_i)$;
– $\mathsf{DKeyComb}((M_{\vec{y},i})_i, \vec{y})$: outputs the functional decryption key $\mathsf{dk}_{\vec{y}} = \left(\vec{y}, \vec{d_{\vec{y}}}\right)$, where $\vec{d_{\vec{y}}}$ is publicly computed as $\mathsf{DSum.Combine}((M_{\vec{y},i})_i)$;

In the last simulated game, we can now show that all the $\mathsf{DKeyGenShare}(\mathsf{sk}_i, \vec{y})$-queries are simulated at random, excepted the last query that requires a DKeyGen-query to the IP-MCFE scheme to get the sum and program the output. Hence, unless all the queries are asked, the functional decryption key is unknown.

9 DDFE

9.1 Definition

Definition 16 (Dynamic Decentralized Functional Encryption). *A dynamic decentralized functional encryption scheme over a set of public keys $\mathcal{PK}$ for functionality $\mathcal{F} : \mathcal{L}(\mathcal{PK} \times \mathcal{K}) \times \mathcal{L}(\mathcal{PK} \times \mathcal{M}) \to \{0,1\}^*$ (where $\mathcal{L}(\mathcal{A})$ denotes the set of finite lists of elements of $\mathcal{A}$ for any set $\mathcal{A}$) consists of four algorithms:*

- $\mathsf{KeyGen}(\lambda)$*: Generates and outputs a party's public key* $\mathsf{pk} \in \mathcal{PK}$ *and the corresponding secret key* sk*;*
- $\mathsf{Encrypt}(\mathsf{sk}, m)$*: Takes as input a party's secret key* sk*, a value $m \in \mathcal{M}$ to encrypt and outputs a ciphertext c_{pk};*
- $\mathsf{DKeyGen}(\mathsf{sk}, k)$*: Takes as input a party's secret key* sk*, a key space object k and outputs a functional decryption key* $\mathsf{dk}_{\mathsf{pk},k}$*;*
- $\mathsf{Decrypt}((\mathsf{dk}_{\mathsf{pk}_i,k_i})_{i\in\mathcal{I}}, (c_{\mathsf{pk}_j})_{j\in\mathcal{J}})$*: Takes as input a set of functional decryption keys* $(\mathsf{dk}_{\mathsf{pk}_i,k_i})_{i\in\mathcal{I}}$*, a set of ciphertexts $(c_{\mathsf{pk}_j})_{j\in\mathcal{J}})$, and outputs a value $y \in \{0,1\}^*$.*

Correctness: We require that

- *for all λ;*
- *for all polynomially-sized sets $\mathcal{I} \subset \mathbb{N}$ and $\mathcal{J} \subset \mathbb{N}$;*
- *for all maps* $(\mathsf{pk}_i, \mathsf{sk}_i)_{i\in\mathcal{I}\cup\mathcal{J}}$ *such that* $\mathsf{pk}_i, \mathsf{sk}_i \leftarrow \mathsf{KeyGen}(\lambda)$ *for all i;*
- *for all* $\mathsf{dk}_{\mathsf{pk}_i,k_i} \leftarrow \mathsf{DKeyGen}(\mathsf{sk}_i, k_i)$ *for all k_i for all $i \in \mathcal{I}$;*
- *for all* $c_{\mathsf{pk}_j} \leftarrow \mathsf{Encrypt}(\mathsf{sk}_j, m_j)$ *for all m_j for all $j \in \mathcal{J}$;*

it holds that

$$\mathsf{Decrypt}((\mathsf{dk}_{\mathsf{pk}_i,k_i})_{i\in\mathcal{I}}, (c_{\mathsf{pk}_j})_{j\in\mathcal{J}}) = F((\mathsf{pk}_i, k_i)_{i\in\mathcal{I}}, (\mathsf{pk}_j, m_j)_{j\in\mathcal{J}})$$

Note that, unlike with single input Functional Encryption, we needn't require that the empty key ϵ be in $\mathcal{K}$ because we operate over lists of elements of $\mathcal{PK} \times \mathcal{K}$, so we simply define ϵ as the empty list.

9.2 Security

(Edouard) TODO

9.3 Functionalities

We define the following functionalities:

Definition 17 (Distributed Sum).

- *$\mathcal{K} = \emptyset$;*
- *$\mathcal{M} = \{x \in \mathbb{Z}_p\} \times \{(\mathsf{pk}_i)_{i\in\mathcal{I}}, \mathcal{I} \subset \mathbb{N}\} \times \{\ell \in \{0,1\}^*\}$;*
- *$F(\epsilon, (x, (\mathsf{pk}_i)_{i\in\mathcal{I}}, \ell)) = (\mathsf{pk}_i)_{i\in\mathcal{I}}, \ell$ and*

$$F(\epsilon, (\mathsf{pk}_i, m_i)_{i\in\mathcal{I}}) = \begin{cases} \sum_i x_i & \text{if, for some } \ell \in \{0,1\}^*, \\ & m_i = (x_i, (\mathsf{pk}_j)_{j\in\mathcal{I}}, \ell) \text{ for all } i \in \mathcal{I}; \\ \perp & \text{otherwise.} \end{cases}$$

Definition 18 (Strict Inner-Product DDFE). *Sets must match perfectly.*

- *$\mathcal{K} = \{(y_j, \mathsf{pk}_j)_{j\in\mathcal{J}}, \mathcal{J} \subset \mathbb{N}\}$;*
- *$\mathcal{M} = \{x \in \mathbb{Z}_p\} \times \{(\mathsf{pk}_i)_{i\in\mathcal{I}}, \mathcal{I} \subset \mathbb{N}\} \times \{\ell \in \{0,1\}^*\}$;*

– $F(\epsilon, (x, (\mathsf{pk}_i)_{i\in\mathcal{I}}, \ell)) = (\mathsf{pk}_i)_{i\in\mathcal{I}}, \ell$ *and*

$$F((\mathsf{pk}_i, k_i)_{i\in\mathcal{I}}, (\mathsf{pk}_i, m_i)_{i\in\mathcal{I}}) = \begin{cases} \sum_i x_i y_i & \text{if } k_i = (y_i, (\mathsf{pk}_t)_{t\in\mathcal{I}}) \text{ and, for some } \ell \in \{0,1\}^*, \\ & m_i = (x_i, (\mathsf{pk}_j)_{j\in\mathcal{I}}, \ell) \text{ for all } i \in \mathcal{I}; \\ \perp & \text{otherwise.} \end{cases}$$

Definition 19 (Ciphertext-policy Inner-Product DDFE). *Only functionality changes. Key can be any subset of ciphertext.*

- $\mathcal{K} = \{(y_j, \mathsf{pk}_j)_{j\in\mathcal{J}}, \mathcal{J} \subset \mathbb{N}\}$;
- $\mathcal{M} = \{x \in \mathbb{Z}_p\} \times \{(\mathsf{pk}_i)_{i\in\mathcal{I}}, \mathcal{I} \subset \mathbb{N}\} \times \{\ell \in \{0,1\}^*\}$;
- $F(\epsilon, (x, (\mathsf{pk}_i)_{i\in\mathcal{I}}, \ell)) = (\mathsf{pk}_i)_{i\in\mathcal{I}}, \ell$ *and*

$$F((\mathsf{pk}_j, k_j)_{j\in\mathcal{J}}, (\mathsf{pk}_i, m_i)_{i\in\mathcal{I}}) = \begin{cases} \sum_j x_j y_j & \text{if } \mathcal{I} \supset \mathcal{J} \text{ and} \\ & k_j = (y_j, (\mathsf{pk}_t)_{t\in\mathcal{J}}) \text{ for all } j \in \mathcal{J} \text{ and,} \\ & \text{for some } \ell \in \{0,1\}^*, \\ & m_i = (x_i, (\mathsf{pk}_j)_{j\in\mathcal{I}}, \ell) \text{ for all } i \in \mathcal{I}; \\ \perp & \text{otherwise.} \end{cases}$$

Definition 20 (Key-policy Inner-Product DDFE). *Message space and functionality both change. Encryption no longer speicifies group.*

- $\mathcal{K} = \{(y_j, \mathsf{pk}_j)_{j\in\mathcal{J}}, \mathcal{J} \subset \mathbb{N}\}$;
- $\mathcal{M} = \{x \in \mathbb{Z}_p\} \times \{\ell \in \{0,1\}^*\}$;
- $F(\epsilon, (x, \ell)) = \ell$ *and*

$$F((\mathsf{pk}_i, k_i)_{i\in\mathcal{I}}, (\mathsf{pk}_i, m_i)_{i\in\mathcal{I}}) = \begin{cases} \sum_i x_i y_i & \text{if } k_i = (y_i, (\mathsf{pk}_t)_{t\in\mathcal{I}}) \text{ and, for some } \ell \in \{0,1\}^*, \\ & m_i = (x_i, \ell) \text{ for all } i \in \mathcal{I}; \\ \perp & \text{otherwise.} \end{cases}$$

F should always be understood to be equal to $\perp$ on inputs on which it was not defined above. (Edouard) TODO: change functionalities to encrypt (unbounded?) vectors? also add more functionalities, some with identity based style identities to partition the space, some ciphertext-policy with the other direction of inclusion?

9.4 A Distributed Sum DDFE

Assume a NIKE $NIKE$, a PRF family $(\mathcal{F}_K)_K$ that outputs in $\mathbb{Z}_p$.

- $\mathsf{KeyGen}(\lambda)$: Return $\mathsf{pk}, \mathsf{sk} = NIKE.\mathsf{KeyGen}(\lambda)$;
- $\mathsf{Encrypt}(\mathsf{sk}, m)$: Parse m as $x \in \mathbb{Z}_p, (\mathsf{pk}_j)_{j\in\mathcal{I}}, \ell \in \{0,1\}^*$. Let i be such that pk_i is our encryptor's public key (Edouard) TODO, include public key in secret key to guarantee that check can be performed?. For all $j \in \mathcal{I}$ such that $i \neq j$, compute $K_j = NIKE.SharedKey(\mathsf{sk}, \mathsf{pk}_i)$ and $r_j = \mathcal{F}_{K_j}(\ell)$. Output $x + \sum_{j<i} r_j - \sum_{j>i} r_j$;
- $\mathsf{DKeyGen}(\mathsf{sk}, k)$: There are no keys in this functionality, so $\mathsf{DKeyGen}$ does not exist;
- $\mathsf{Decrypt}((\mathsf{dk}_{\mathsf{pk}_i, k_i})_{i\in\mathcal{I}}, (c_{\mathsf{pk}_j})_{j\in\mathcal{J}})$: Return $\sum_{i\in\mathcal{J}} c_{\mathsf{pk}_i}$.

Correctness:

$$\begin{aligned} \sum_{i\in\mathcal{J}} c_{\mathsf{pk}_i} &= \sum_i \left(x_i + \sum_{j<i} r_j - \sum_{j>i} r_j \right) \\ &= \sum_i x_i + \sum_{\substack{i,j\in\mathcal{I} \\ i<j}} NIKE.SharedKey(\mathsf{sk}_j, \mathsf{pk}_i) - NIKE.SharedKey(\mathsf{sk}_i, \mathsf{pk}_j) \\ &= \sum_i x_i \end{aligned}$$

by correctness of the NIKE.

Security: (Edouard) TODO

9.5 A Strict IP DDFE

Assume a group $\mathbb{G}$, a random oracle $\mathcal{H}$, an SSL SSL(Edouard) TODO: rename SSL, I'm not in favor of using threshold IBE here because that requires calling both keygen and encrypt. We can do simpler, with only encrypts (although we can show this can be built from TIBE), a Distributed Sum $DSum$, and a PRF family $(\mathcal{F}_K)_K$ that outputs in $\mathbb{Z}_p$.

- $\mathsf{KeyGen}(\lambda)$: Sample a PRF Key K.
  Sample $SSL.\mathsf{pk}, SSL.\mathsf{sk} = SSL.\mathsf{KeyGen}()$. Sample $DSum.\mathsf{pk}, DSum.\mathsf{sk} = DSum.\mathsf{KeyGen}()$.
  Return $(\mathsf{pk}, \mathsf{sk}) = ((SSL.\mathsf{pk}, DSum.\mathsf{pk}), (K, SSL.\mathsf{sk}, DSum.\mathsf{sk}))$;
- $\mathsf{Encrypt}(\mathsf{sk}, m)$: Parse m as $x \in \mathbb{Z}_p, (\mathsf{pk}_i)_{i\in\mathcal{I}}, \ell \in \{0,1\}^*$. We write $G = (\mathsf{pk}_i)_{i\in\mathcal{I}}$. Compute s as $\mathcal{F}_K(G)$. Compute $[h_\ell] = \mathcal{H}(\ell)$. Return $SSL.\mathsf{Encrypt}(SSL.\mathsf{sk}, [x]\cdot[h_\ell]^s)$;
- $\mathsf{DKeyGen}(\mathsf{sk}, k)$: Parse k as $(y_j, \mathsf{pk}_j)_{j\in\mathcal{J}}$. We write $G = (\mathsf{pk}_j)_{j\in\mathcal{J}}$. Compute s as $\mathcal{F}_K(G)$. Let i be such that $\mathsf{pk}_i = \mathsf{pk}$.
  Return $DSum.\mathsf{Encrypt}(DSum.\mathsf{sk}, (sy_i, (DSum.\mathsf{pk}_j)_{j\in\mathcal{J}}, (y_j, \mathsf{pk}_j)_{j\in\mathcal{J}}))$;
- $\mathsf{Decrypt}((\mathsf{dk}_{\mathsf{pk}_j,k_j})_{i\in\mathcal{J}}, (c_{\mathsf{pk}_i})_{i\in\mathcal{I}})$: If $\mathcal{J} \neq \mathcal{I}$ return $\perp$.
  Call $SSL.\mathsf{Decrypt}(\epsilon, (c_{\mathsf{pk}_i})_{i\in\mathcal{I}}) = (c_i)_{i\in\mathcal{I}}$.
  Call $DSum.\mathsf{Decrypt}(\epsilon, \mathsf{dk}_{\mathsf{pk}_i,k_i})_{i\in\mathcal{I}}) = \mathsf{dk}_y$.
  Compute $[h_\ell] = \mathcal{H}(\ell)$.
  Compute
  $$\left(\prod_{j\in\mathcal{J}} c_j^{y_j}\right) / [h_\ell]^{\mathsf{dk}_y}$$
  and return its discrete logarithm in base $[1]$.

Correctness: We write $s_i = \mathcal{F}_{K_i}(G)$ where $G = (\mathsf{pk}_i)_{i\in\mathcal{I}}$. By correctness of SSL, $c_i = [x_i]\cdot[h_\ell]^{s_i}$. By correctness of $DSum$, $\mathsf{dk}_y = \sum_i s_i y_i$. Thus we compute

$$\begin{aligned}\left(\prod_{i\in\mathcal{I}} c_i^{y_i}\right) / [h_\ell]^{\mathsf{dk}_y} &= \left(\prod_i [x_i y_i + s_i y_i h_\ell]\right) / \left[h_\ell\left(\sum_i s_i y_i\right)\right] \\ &= \left(\prod_i [x_i y_i]\right)\cdot\left[h_\ell\left(\sum_i s_i y_i - s_i y_i\right)\right] \\ &= [\sum_i x_i y_i]\end{aligned}$$

as expected.

Security: (Edouard) TODO

9.6 A Ciphertext-policy IP DDFE

Assume a group $\mathbb{G}$, a random oracle $\mathcal{H}$, an SSL SSL and a Distributed Sum $DSum$.

- $\mathsf{KeyGen}(\lambda)$: Sample $s \xleftarrow{\$} \mathbb{Z}_p$.
  Sample $SSL.\mathsf{pk}, SSL.\mathsf{sk} = SSL.\mathsf{KeyGen}()$. Sample $DSum.\mathsf{pk}, DSum.\mathsf{sk} = DSum.\mathsf{KeyGen}()$.
  Return $(\mathsf{pk}, \mathsf{sk}) = ((SSL.\mathsf{pk}, DSum.\mathsf{pk}), (s, SSL.\mathsf{sk}, DSum.\mathsf{sk}))$;
- $\mathsf{Encrypt}(\mathsf{sk}, m)$: Parse m as $x \in \mathbb{Z}_p, (\mathsf{pk}_j)_{j\in\mathcal{I}}, \ell \in \{0,1\}^*$. We write $G = (\mathsf{pk}_j)_{j\in\mathcal{I}}$. Compute $[h_{G||\ell}] = \mathcal{H}(G||\ell)$. Return $SSL.\mathsf{Encrypt}(SSL.\mathsf{sk}, [x]\cdot[h_{G||\ell}]^s)$;
- $\mathsf{DKeyGen}(\mathsf{sk}, k)$: Parse k as $(y_j, \mathsf{pk}_j)_{j\in\mathcal{J}}$. Let i be such that $\mathsf{pk}_i = \mathsf{pk}$.
  Return $DSum.\mathsf{Encrypt}(DSum.\mathsf{sk}, (sy_i, (DSum.\mathsf{pk}_j)_{j\in\mathcal{J}}, (y_j, \mathsf{pk}_j)_{j\in\mathcal{J}}))$;

- $\mathsf{Decrypt}((\mathsf{dk}_{\mathsf{pk}_j,k_j})_{i\in\mathcal{J}},(c_{\mathsf{pk}_i})_{i\in\mathcal{I}})$: If $\mathcal{J}\not\subset\mathcal{I}$ return $\perp$.
  Call $SSL.\mathsf{Decrypt}(\epsilon,(c_{\mathsf{pk}_i})_{i\in\mathcal{I}})=(c_i)_{i\in\mathcal{I}}$.
  Call $DSum.\mathsf{Decrypt}(\epsilon,\mathsf{dk}_{\mathsf{pk}_j,k_j})_{j\in\mathcal{I}})=\mathsf{dk}_y$.
  Compute $[h_{G||\ell}]=\mathcal{H}(G||\ell)$.
  Compute
  $$\left(\prod_{j\in\mathcal{J}}c_j^{y_j}\right)/[h_{G||\ell}]^{\mathsf{dk}_y}$$
  and return its discrete logarithm in base $[1]$.

Correctness: By correctness of SSL, $c_i=[x_i]\cdot[h_{G||\ell}]^{s_i}$. By correctness of $DSum$, $\mathsf{dk}_y=\sum_j s_jy_j$. Thus we compute

$$\begin{aligned}\left(\prod_{j\in\mathcal{J}}c_j^{y_j}\right)/[h_{G||\ell}]^{\mathsf{dk}_y}&=\left(\prod_i[x_jy_j+s_jy_jh_{G||\ell}]\right)/\left[h_{G||\ell}\left(\sum_j s_jy_j\right)\right]\\&=\left(\prod_j[x_jy_j]\right)\cdot\left[h_{G||\ell}\left(\sum_j s_jy_j-s_jy_j\right)\right]\\&=[\sum_j x_jy_j]\end{aligned}$$

as expected.

Security: (Edouard) TODO

9.7 A Key-policy IP DDFE

Assume a pairing group $e:\mathbb{G}_1\times\mathbb{G}_2\to\mathbb{G}_T$, random oracles $\mathcal{H}_1:\{0,1\}^*\to\mathbb{G}_1$ and $\mathcal{H}_2:\{0,1\}^*\to\mathbb{G}_2$, and an (exponential) Distributed Sum $DSum$ into $\mathbb{G}_2$ (Edouard) TODO: define.

- $\mathsf{KeyGen}(\lambda)$: Sample $s\xleftarrow{\$}\mathbb{Z}_p$.
  Sample $t\xleftarrow{\$}\mathbb{Z}_p$.
  Sample $DSum.\mathsf{pk},DSum.\mathsf{sk}=DSum.\mathsf{KeyGen}()$.
  Return $(\mathsf{pk},\mathsf{sk})=((DSum.\mathsf{pk}),(s,t,DSum.\mathsf{sk}))$;
- $\mathsf{Encrypt}(\mathsf{sk},m)$: Parse m as $x\in\mathbb{Z}_p,\ell\in\{0,1\}^*$. Compute $[h_\ell]_1=\mathcal{H}_1(\ell)$. Return $[x]_1\cdot[h_\ell]_1^s,[h_\ell]_1^t$;
- $\mathsf{DKeyGen}(\mathsf{sk},k)$: Parse k as $(y_j,\mathsf{pk}_j)_{j\in\mathcal{J}}$. Let i be such that $\mathsf{pk}_i=\mathsf{pk}$.
  We write $G=(\mathsf{pk}_j)_{j\in\mathcal{J}}$. Compute $[h_{G||y}]_2=\mathcal{H}_2(G||y)$.
  Return $DSum.\mathsf{Encrypt}(DSum.\mathsf{sk},([h_{G||y}]_2^{sy_i+t},(DSum.\mathsf{pk}_j)_{j\in\mathcal{J}},(y_j,\mathsf{pk}_j)_{j\in\mathcal{J}}))$;
- $\mathsf{Decrypt}((\mathsf{dk}_{\mathsf{pk}_j,k_j})_{i\in\mathcal{J}},(c_{\mathsf{pk}_i})_{i\in\mathcal{I}})$: If $\mathcal{J}\neq\mathcal{I}$ return $\perp$.
  Call $DSum.\mathsf{Decrypt}(\epsilon,\mathsf{dk}_{\mathsf{pk}_i,k_i})_{i\in\mathcal{I}})=\mathsf{dk}_y$.
  Write $c_i,d_i=c_{\mathsf{pk}_i}$.
  Compute $[h_\ell]_1=\mathcal{H}_1(\ell)$.
  Compute $[h_{G||y}]_2=\mathcal{H}_2(G||y)$.
  Compute
  $$\left(\prod_{i\in\mathcal{I}}e(c_i^{y_i},[h_{G||y}]_2)\right)\cdot e(\prod_i d_i,[h_{G||y}]_2)/e([h_\ell]_1,\mathsf{dk}_y)$$
  and return its discrete logarithm in base $e([1]_1,[h_{G||y}]_2)=[h_{G||y}]_T$.

Correctness: By correctness of $DSum$, $\mathsf{dk}_y = \left[h_{G||y}\left(\sum_j s_j y_j + t_j\right)\right]_2$. By definition of encrypt $c_i = [x_i]_1 \cdot [h_\ell]_1^{s_i}$ and $d_i = [h_\ell]_1^{t_i}$. Thus we compute

$$\begin{aligned}
&\left(\prod_{i\in\mathcal{I}} e\left(c_i^{y_i}, [h_{G||y}]_2\right)\right) \cdot e\left(\prod_i d_i, [h_{G||y}]_2\right) / e\left([h_\ell]_1, \mathsf{dk}_y\right) \\
&= \frac{\left(\prod_{i\in\mathcal{I}} e\left(([x_i]_1 \cdot [h_\ell]_1^{s_i})^{y_i}, [h_{G||y}]_2\right)\right) \cdot e\left(\prod_i [h_\ell]_1^{t_i}, [h_{G||y}]_2\right)}{e\left([h_\ell]_1, \left[h_{G||y}\left(\sum_i s_i y_i + t_i\right)\right]_2\right)} \\
&= \frac{\left[\sum_{i\in\mathcal{I}} x_i y_i h_{G||y} + s_i y_i h_\ell h_{G||y}\right]_T \cdot \left[\sum_{i\in\mathcal{I}} t_i h_\ell h_{G||y}\right]_T}{\left[\sum_i (s_i y_i + t_i) h_\ell h_{G||y}\right]_T} \\
&= \left(\sum_{i\in\mathcal{I}} x_i y_i\right) [h_{G||y}]_T
\end{aligned}$$

as expected.

Security: (Edouard) TODO

Acknowledgments.

This work was supported in part by the European Community's Seventh Framework Programme (FP7/2007-2013 Grant Agreement no. 339563 – CryptoCloud), the European Community's Horizon 2020 Project FENTEC (Grant Agreement no. 780108), the Google PhD fellowship, and the French FUI ANBLIC Project.

References

ABDP15. Michel Abdalla, Florian Bourse, Angelo De Caro, and David Pointcheval. Simple functional encryption schemes for inner products. In Jonathan Katz, editor, *PKC 2015*, volume 9020 of *LNCS*, pages 733–751. Springer, Heidelberg, March / April 2015.

ABKW19. Michel Abdalla, Fabrice Benhamouda, Markulf Kohlweiss, and Hendrik Waldner. Decentralizing inner-product functional encryption. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part II*, volume 11443 of *LNCS*, pages 128–157. Springer, Heidelberg, April 2019.

ABSV15. Prabhanjan Ananth, Zvika Brakerski, Gil Segev, and Vinod Vaikuntanathan. From selective to adaptive security in functional encryption. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 657–677. Springer, Heidelberg, August 2015.

ACF$^+$18. Michel Abdalla, Dario Catalano, Dario Fiore, Romain Gay, and Bogdan Ursu. Multi-input functional encryption for inner products: Function-hiding realizations and constructions without pairings. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 597–627. Springer, Heidelberg, August 2018.

ACF$^+$19. Shweta Agrawal, Michael Clear, Ophir Frieder, Sanjam Garg, Adam O'Neill, and Justin Thaler. Ad hoc multi-input functional encryption. Cryptology ePrint Archive, Report 2019/356, 2019. `https://eprint.iacr.org/2019/356`.

AGRW17. Michel Abdalla, Romain Gay, Mariana Raykova, and Hoeteck Wee. Multi-input inner-product functional encryption from pairings. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 601–626. Springer, Heidelberg, April / May 2017.

ALS16. Shweta Agrawal, Benoît Libert, and Damien Stehlé. Fully secure functional encryption for inner products, from standard assumptions. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 333–362. Springer, Heidelberg, August 2016.

BCFG17. Carmen Elisabetta Zaira Baltico, Dario Catalano, Dario Fiore, and Romain Gay. Practical functional encryption for quadratic functions with applications to predicate encryption. Cryptology ePrint Archive, Report 2017/151, 2017. `http://eprint.iacr.org/2017/151`.

BDJR97. Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *38th FOCS*, pages 394–403. IEEE Computer Society Press, October 1997.

BGJS16. Saikrishna Badrinarayanan, Vipul Goyal, Aayush Jain, and Amit Sahai. Verifiable functional encryption. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 557–587. Springer, Heidelberg, December 2016.

BKS16. Zvika Brakerski, Ilan Komargodski, and Gil Segev. Multi-input functional encryption in the private-key setting: Stronger security from weaker assumptions. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 852–880. Springer, Heidelberg, May 2016.

Boy99. Victor Boyko. On the security properties of OAEP as an all-or-nothing transform. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 503–518. Springer, Heidelberg, August 1999.

BSW11. Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 253–273. Springer, Heidelberg, March 2011.

CDG$^+$18. Jérémy Chotard, Edouard Dufour Sans, Romain Gay, Duong Hieu Phan, and David Pointcheval. Decentralized multi-client functional encryption for inner product. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 703–732. Springer, Heidelberg, December 2018.

CDH$^+$00. Ran Canetti, Yevgeniy Dodis, Shai Halevi, Eyal Kushilevitz, and Amit Sahai. Exposure-resilient functions and all-or-nothing transforms. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 453–469. Springer, Heidelberg, May 2000.

CPP05. Hervé Chabanne, Duong Hieu Phan, and David Pointcheval. Public traceability in traitor tracing schemes. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 542–558. Springer, Heidelberg, May 2005.

DGP18. Edouard Dufour Sans, Romain Gay, and David Pointcheval. Reading in the dark: Classifying encrypted digits with functional encryption. Cryptology ePrint Archive, Report 2018/206, 2018. https://eprint.iacr.org/2018/206.

EHK$^+$13. Alex Escala, Gottfried Herold, Eike Kiltz, Carla Ràfols, and Jorge Villar. An algebraic framework for Diffie-Hellman assumptions. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 129–147. Springer, Heidelberg, August 2013.

FO99. Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 537–554. Springer, Heidelberg, August 1999.

Gay16. Romain Gay. Functional encryption for quadratic functions, and applications to predicate encryption. Cryptology ePrint Archive, Report 2016/1106, 2016. http://eprint.iacr.org/2016/1106.

GGG$^+$14. Shafi Goldwasser, S. Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 578–602. Springer, Heidelberg, May 2014.

GGH$^+$13. Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE Computer Society Press, October 2013.

GGJS13. Shafi Goldwasser, Vipul Goyal, Abhishek Jain, and Amit Sahai. Multi-input functional encryption. Cryptology ePrint Archive, Report 2013/727, 2013. http://eprint.iacr.org/2013/727.

GKL$^+$13. S. Dov Gordon, Jonathan Katz, Feng-Hao Liu, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. Cryptology ePrint Archive, Report 2013/774, 2013. http://eprint.iacr.org/2013/774.

GKP$^+$13a. Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run turing machines on encrypted data. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 536–553. Springer, Heidelberg, August 2013.

GKP$^+$13b. Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th ACM STOC*, pages 555–564. ACM Press, June 2013.

GVW12. Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption with bounded collusions via multi-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 162–179. Springer, Heidelberg, August 2012.

HILL99. Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

ILL89. Russell Impagliazzo, Leonid A. Levin, and Michael Luby. Pseudo-random generation from one-way functions (extended abstracts). In *21st ACM STOC*, pages 12–24. ACM Press, May 1989.

KDK11. Klaus Kursawe, George Danezis, and Markulf Kohlweiss. Privacy-friendly aggregation for the smart-grid. In Simone Fischer-Hübner and Nicholas Hopper, editors, *Privacy Enhancing Technologies - 11th International Symposium, PETS '11*, volume 6794 of *LNCS*, pages 175–191. Springer, 2011.

KY02. Aggelos Kiayias and Moti Yung. Traitor tracing with constant transmission rate. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 450–465. Springer, Heidelberg, April / May 2002.

NY90. Moni Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *22nd ACM STOC*, pages 427–437. ACM Press, May 1990.

O'N10. Adam O'Neill. Definitional issues in functional encryption. Cryptology ePrint Archive, Report 2010/556, 2010. https://eprint.iacr.org/2010/556.

Riv97. Ronald L. Rivest. All-or-nothing encryption and the package transform. In Eli Biham, editor, *FSE'97*, volume 1267 of *LNCS*, pages 210–218. Springer, Heidelberg, January 1997.

SS10. Amit Sahai and Hakan Seyalioglu. Worry-free encryption: functional encryption with public keys. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 2010*, pages 463–472. ACM Press, October 2010.

SW05. Amit Sahai and Brent R. Waters. Fuzzy identity-based encryption. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 457–473. Springer, Heidelberg, May 2005.

Wat15. Brent Waters. A punctured programming approach to adaptively secure functional encryption. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 678–697. Springer, Heidelberg, August 2015.

A Definition of Symmetric Key Encryption

A symmetric key encryption $(\mathsf{SEnc}, \mathsf{SDec})$ with key space $\mathcal{K}$ is defined as:

- $\mathsf{SEnc}(K, m)$: given a key K and a message m, outputs a ciphertext ct;
- $\mathsf{SDec}(K, \mathsf{ct})$: given a key K and a ciphertext ct, output a plaintext.

The following must hold:

Correctness. For all m in the message space, $\Pr[\mathsf{SDec}(K, \mathsf{SEnc}(K, m)) = m] = 1$, where the probability is taken over $K \xleftarrow{\$} \mathcal{K}$.

One Time Security. For any PPT adversary $\mathcal{A}$, the following advantage is negligible:

$$\mathsf{Adv}^{\mathsf{OT}}_{\mathsf{SKE}}(\mathcal{A}) = \left| 2 \times \Pr\left[b' = b : \begin{array}{l} (m_0, m_1) \leftarrow \mathcal{A}(1^\lambda) \\ K \xleftarrow{\$} \mathcal{K}, b \xleftarrow{\$} \{0,1\}, \mathsf{ct} = \mathsf{SEnc}(K, m_b) \\ b' \leftarrow \mathcal{A}(\mathsf{ct}) \end{array} \right] - 1 \right| .$$

If the key space is larger than the message space, on can simply use the one-time pad to build a one-time secure symmetric encryption. Otherwise, a pseudo-random generator can stretch the key to the right length.

B MCFE with Repetitions: Proofs

In this section, we describe the `1-Label-IND`*-security and prove how the scheme from 6.2 achieves it. The `1-Label-IND`* security is exactly the same security notion as `IND`* where the challenge $\mathsf{QLeftRight}$ oracle can only be queried with the same label. Hence, as above, the index ρ of the target label ℓ^* is provided by the adversary, at the beginning, and so we can assume that all the encryption queries for $\ell^* = \ell_\rho$ are asked to the $\mathsf{QLeftRight}$ oracle, while the other encryption queries are asked to the $\mathsf{QEncrypt}$ oracle (contrarily to the encapsulation in the SSL scheme, encryption uses a secret key). It is well-known that `1-Label-IND`* and `IND`* are equivalent [BDJR97], but the former is more convenient in our security proof.

`1-Label-IND`* Security for MCFE : As just explained, the `1-Label-IND`*-security game for MCFE is exactly the `IND`*-security game for MCFE from Definition 2 where only one label ℓ^* is allowed in the challenge $\mathsf{QLeftRight}$ oracle, defined by its index ρ, at the initialization step. We assume that all the other encryption queries are asked to the $\mathsf{QEncrypt}$ oracle.

Definition 21 (`1-Label-IND`*-Security Game for MCFE). *Let us consider MCFE, a scheme over a set of n senders. No adversary $\mathcal{A}$ should be able to win the following security game against a challenger $\mathcal{C}$:*

- *Initialize(ρ): the adversary announces the index of the unique label $\ell^* = \ell_\rho$ that will be involved in challenge queries. The challenger $\mathcal{C}$ runs the setup algorithm* $(\mathsf{mpk}, \mathsf{msk}, (\mathsf{ek}_i)_i) \leftarrow \mathsf{SetUp}(\lambda)$ *and chooses a random bit $b \xleftarrow{\$} \{0,1\}$. It provides* mpk *to the adversary $\mathcal{A}$;*
- *Encryption queries* $\mathsf{QEncrypt}(i, x, \ell_j)$*: $\mathcal{A}$ has unlimited and adaptive access to the encryption oracle (for $j \neq \rho$), and receives the ciphertext $C_{\ell_j,i} \leftarrow$* $\mathsf{Encrypt}(\mathsf{ek}_i, x, \ell_j)$*;*
- *Challenge queries* $\mathsf{QLeftRight}(i, x^0, x^1, \ell_\rho)$*: $\mathcal{A}$ has unlimited and adaptive access to a Left-or-Right encryption oracle (for the label ℓ_ρ only), and receives the ciphertext $C_{\ell_\rho,i} \leftarrow$* $\mathsf{Encrypt}(\mathsf{ek}_i, x^b, \ell_\rho)$*;*
- *Functional decryption key queries* $\mathsf{QDKeyGen}(f)$*: $\mathcal{A}$ has unlimited and adaptive access to the* $\mathsf{DKeyGen}(\mathsf{msk}, f)$ *algorithm for any input function f of its choice. It is given back the functional decryption key* dk_f*;*
- *Corruption queries* $\mathsf{QCorrupt}(i)$*: $\mathcal{A}$ can make an unlimited number of adaptive corruption queries on input index i, to get the encryption key* ek_i *of any sender i of its choice;*
- *Finalize: $\mathcal{A}$ provides its guess b' on the bit b, and this procedure outputs the result β of the security game, according to the analysis given below, where $\ell^* = \ell_\rho$.*

The output β of the game depends on some conditions, where $\mathcal{CS}$ is the set of corrupted senders (the set of indexes i input to $\mathsf{QCorrupt}$ *during the whole game), and $\mathcal{HS}$ the set of honest (non-corrupted) senders. We set the output to $\beta \leftarrow b'$, unless one of the cases below is true, in which case we set $\beta \xleftarrow{\$} \{0,1\}$:*

1. *some* $\mathsf{QLeftRight}(i, x_i^0, x_i^1, \ell^*)$*-query has been asked for an index $i \in \mathcal{CS}$ with $x_i^0 \neq x_i^1$;*
2. *for some function f asked to* $\mathsf{QDKeyGen}$*, there exists a pair of vectors $(\vec{x}^0 = (x_i^0)_i, \vec{x}^1 = (x_i^1)_i)$ such that $f(\vec{x}^0) \neq f(\vec{x}^1)$, when*
   - *$x_i^0 = x_i^1$, for all $i \in \mathcal{CS}$;*
   - $\mathsf{QLeftRight}(i, x_i^0, x_i^1, \ell^*)$*-queries have been asked for all $i \in \mathcal{HS}$.*
3. *a challenge query* $\mathsf{QLeftRight}(i, x_i^0, x_i^1, \ell^*)$ *has been asked for some $i \in \mathcal{HS}$, but challenge queries* $\mathsf{QLeftRight}(j, x_j^0, x_j^1, \ell^*)$ *have not all been asked for all $j \in \mathcal{HS}$.*

We say MCFE is `1-Label-IND`**-secure if for any adversary $\mathcal{A}$, its advantage $\mathsf{Adv}_{\mathit{MCFE}}^{\texttt{1-Label-IND}^*}(\mathcal{A}) = |\Pr[\beta = 1|b = 1] - \Pr[\beta = 1|b = 0]|$ is negligible.*

We can also define the weaker static, selective, and/or without-repetition variants.

We can state the following security result:

Theorem 22. *For any adversary $\mathcal{A}$, against the* `1-Label-IND`**-security of the above MCFE′,*

$$\mathsf{Adv}_{\mathit{MCFE}'}^{\texttt{1-Label-IND}^*}(\mathcal{A}) \leq \mathsf{Adv}_{\mathit{MCFE}}^{\texttt{wtr-IND}^*}(t') + n \cdot \mathsf{Adv}_{\mathsf{IP\text{-}FE}}^{\texttt{IND}}(t''),$$

where both t' and t'' are close to the running time t of $\mathcal{A}$.

As a consequence, using the IP-FE from [ALS16], the IP-MCFE from [CDG$^+$18], and adding the above SSE scheme, one gets an IP-MCFE that is `IND`-secure, with repetitions and with adaptive corruptions.

The proof uses a series of hybrid games, defined below. For any game $\mathbf{G}$, we denote $\mathsf{Adv}_{\mathbf{G}}(\mathcal{A})$ the advantage of $\mathcal{A}$ in the game $\mathbf{G}$, that is, the probability that the procedure Finalize in the game $\mathbf{G}$ outputs 1. For any user $i \in [n]$, we denote by Q_i the number of queries to the oracle $\mathsf{QLeftRight}'$ containing the user i, that is, of the form: $\mathsf{QLeftRight}'(i, \vec{x}_i^{k,0}, \vec{x}_i^{k,1}, \ell)$, for $k \in \{1, \ldots, Q_i\}$. When all the Q_i's are 1, there is no repetition, but here we are dealing with repetitions. The counter k numbers the repetitions.

Game $\mathbf{G}_\beta$: For any $\beta \in \{0,1\}$, we define the following game, where multiple plaintexts can be queried for the same user i and the same label. We use a counter k, which starts at 1 to number the queries $(\vec{x}_i^{k,0}, \vec{x}_i^{k,1})$, under the label $\ell^* = \ell_\rho$. We do not keep track of the queries under other labels (as in previous definitions).

– Initialize(ρ): it generates $(\mathsf{mpk}, \mathsf{msk}, (\mathsf{ek}_i)_{i\in[n]}) \leftarrow \mathsf{SetUp}(\lambda)$, and for all $i \in [n]$, $(\mathsf{IP.mpk}_i, \mathsf{IP.msk}_i) \leftarrow \mathsf{IP.SetUp}(\lambda)$. It returns $\mathsf{mpk}' := (\mathsf{mpk}, (\mathsf{IP.mpk}_i)_{i\in[n]})$ to the adversary $\mathcal{A}$;
– $\mathsf{QEncrypt}'(i, \vec{x}_i, \ell_j)$: it first computes $[\vec{c}_i] \leftarrow \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i, \ell_j)$, and returns $\mathsf{IP.Enc}(\mathsf{msk}_i, [\vec{c}_i])$;
– $\mathsf{QLeftRight}'(i, \vec{x}_i^{k,0}, \vec{x}_i^{k,1}, \ell_\rho)$: it computes $[\vec{c}_i^{\,k}] \leftarrow \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i^{k,\beta}, \ell_\rho)$, and returns $\mathsf{IP.Enc}(\mathsf{msk}_i, [\vec{c}_i^{\,k}])$;
– $\mathsf{QDKeyGen}'(\vec{y})$: on input $\vec{y} := (\vec{y}_1\|\cdots\|\vec{y}_n) \in \mathbb{Z}_p^{nm}$, it first computes $\mathsf{dk}_{\vec{y}} = \mathsf{DKeyGen}(\mathsf{msk}, \vec{y})$, and for all $i \in [n]$: $\mathsf{dk}_{\vec{y}_i} = \mathsf{IP.DKeyGen}(\mathsf{msk}_i, \vec{y}_i)$. It returns $\mathsf{dk}'_{\vec{y}} = \big(\mathsf{dk}_{\vec{y}}, \{\mathsf{dk}_{\vec{y}_i}\}_{i\in[n]}\big)$.
– $\mathsf{QCorrupt}'(i)$: on input a user $i \in [n]$, it returns $(\mathsf{ek}_i, \mathsf{IP.msk}_i)$.
– Finalize: as in Definition 21.

Note that:
$$\mathsf{Adv}^{\texttt{1-Label-IND}^*}_{\mathsf{MCFE}'}(\mathcal{A}) = |\mathsf{Adv}_{\mathbf{G}_0}(\mathcal{A}) - \mathsf{Adv}_{\mathbf{G}_1}(\mathcal{A})|.$$

Game $\mathbf{H}_0$: Now we consider the game $\mathbf{H}_0$ defined exactly as $\mathbf{G}_0$, except in $\mathsf{QLeftRight}'(i, \vec{x}_i^{k,0}, \vec{x}_i^{k,1}, \ell_\rho)$, one computes $[\vec{c}_i^{\,k}] \leftarrow \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i^{k,0} + \vec{x}_i^{1,1} - \vec{x}_i^{1,0}, \ell_\rho)$. Then it returns $\mathsf{IP.Enc}(\mathsf{IP.msk}_i, [\vec{c}_i^{\,k}])$. The transition from $\mathbf{G}_0$ and $\mathbf{H}_0$ uses $\texttt{1-Label-IND}^*$ security and the linear homomorphism of the ciphertexts of MCFE. Namely, we build a PPT adversary $\mathcal{B}$ against the $\texttt{1-Label-IND}^*$ security of MCFE such that:
$$|\mathsf{Adv}_{\mathbf{G}_0}(\mathcal{A}) - \mathsf{Adv}_{\mathbf{H}_0}(\mathcal{A})| \leq \mathsf{Adv}^{\texttt{1-Label-IND}^*}_{\mathsf{MCFE}}(\mathcal{B}).$$

$\mathcal{B}$ simulates the view of the $\texttt{1-Label-IND}^*$-adversary $\mathcal{A}$ against MCFE' as follows:

– Initialize(ρ): after having sent ρ, it gets mpk from its $\texttt{1-Label-IND}^*$ challenger. For all $i \in [n]$, $(\mathsf{IP.mpk}_i, \mathsf{IP.msk}_i) \leftarrow \mathsf{IP.SetUp}(\lambda)$, and it returns $\mathsf{mpk}' := (\mathsf{mpk}, (\mathsf{IP.mpk}_i)_{i\in[n]})$ to the adversary $\mathcal{A}$;
– $\mathsf{QEncrypt}'(i, \vec{x}_i, \ell_j)$: it first computes $[\vec{c}_i] \leftarrow \mathsf{QEncrypt}(i, \vec{x}_i, \ell_j)$, and returns $\mathsf{IP.Enc}(\mathsf{msk}_i, [\vec{c}_i])$;
– $\mathsf{QLeftRight}'(i, \vec{x}_i^{k,0}, \vec{x}_i^{k,1}, \ell_\rho)$: for $k = 1$, i.e. the first query for user i, $\mathcal{B}$ queries its own $\mathsf{QLeftRight}$ oracle to get $[\vec{c}_i^{\,1}] = \mathsf{QLeftRight}(i, \vec{x}_i^{k,0}, \vec{x}_i^{k,1}, \ell_\rho)$; otherwise it computes $[\vec{c}_i^{\,k}] := [\vec{c}_i^{\,1}] + [\vec{x}_i^{k,0} - \vec{x}_i^{1,0}]$. It then returns $\mathsf{IP.Encrypt}(\mathsf{IP.msk}_i, [\vec{c}_i^{\,k}])$ to $\mathcal{A}$;
– $\mathsf{QDKeyGen}'(\vec{y})$: on input $\vec{y} := (\vec{y}_1\|\cdots\|\vec{y}_n) \in \mathbb{Z}_p^{nm}$, it first computes $\mathsf{dk}_{\vec{y}} = \mathsf{DKeyGen}(\mathsf{msk}, \vec{y})$, and for all $i \in [n]$: $\mathsf{dk}_{\vec{y}_i} = \mathsf{IP.DKeyGen}(\mathsf{msk}_i, \vec{y}_i)$. It returns $\mathsf{dk}'_{\vec{y}} = \big(\mathsf{dk}_{\vec{y}}, \{\mathsf{dk}_{\vec{y}_i}\}_{i\in[n]}\big)$.
– $\mathsf{QCorrupt}'(i)$: $\mathcal{B}$ queries its own oracle to obtain $\mathsf{ek}_i \leftarrow \mathsf{QCorrupt}(i)$, and returns $(\mathsf{ek}_i, \mathsf{IP.msk}_i)$ to $\mathcal{A}$.
– Finalize: $\mathcal{B}$ verifies that the conditions in Definition 2 are satisfied; if they are, it forwards the guess b' of $\mathcal{A}$, otherwise, it sends a random bit to its own Finalize oracle.

Note that the constraints $\mathcal{B}$ has to verify in the finalize procedure, and namely for condition (2), might look exponential for general functionalities. But in the case of inner-product, one just has to look at spanned vector sub-spaces. Namely, all queries $(i, \vec{x}_i^{k_i,0}, \vec{x}_i^{k_i,1}, \ell_\rho)_{i\in[n],k_i\in[Q_i]}$ to $\mathsf{QLeftRight}'$ and all queries $\vec{y} := (\vec{y}_1\|\cdots\|\vec{y}_n)$ to $\mathsf{QDKeyGen}'$ must satisfy: $\sum_i\langle\vec{x}_i^{k_i,0}, \vec{y}_i\rangle = \sum_i\langle\vec{x}_i^{k_i,1}, \vec{y}_i\rangle$. This is an exponential number of linear equations, but, as noted in [AGRW17], it suffices to verify the linearly independent equations, of which there can be at most $n \cdot m$. This can be done efficiently given the queries.

One can note that, for the label $\ell_\rho = \ell^*$, $[\vec{c}_i^{\,1}]$ received by $\mathcal{B}$ is actually $[\vec{c}_i^{\,1}] = \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i^{1,b}, \ell^*)$, where b is the random bit chosen by the $\texttt{1-Label-IND}^*$ security game for MCFE that $\mathcal{B}$ is interacting with. By linear homomorphism of the ciphertexts of MCFE, for all $k \in [Q_i]$, we have: $[\vec{c}_i^{\,k}] = \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i^{1,b}, \ell^*) + [\vec{x}_i^{k,0} - \vec{x}_i^{1,0}] = \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i^{k,0} + \vec{x}_i^{1,b} - \vec{x}_i^{1,0}, \ell^*)$. So, when $b = 0$, $\mathcal{B}$ simulates $\mathbf{G}_0$, while it simulates $\mathbf{H}_0$ when $b = 1$, which proves $|\mathsf{Adv}_{\mathbf{G}_0}(\mathcal{A}) - \mathsf{Adv}_{\mathbf{H}_0}(\mathcal{A})| \leq \mathsf{Adv}^{\texttt{1-Label-IND}^*}_{\mathsf{MCFE}}(\mathcal{B})$.

We define the following hybrid games $\mathbf{H}_r$, for all $r \in [n]$, as $\mathbf{H}_0$, except for $\mathsf{QLeftRight}'(i, \vec{x}_i^{k,0}, \vec{x}_i^{k,1}, \ell_\rho)$: for all $i \leq r$, it sets $[\vec{c}_i^{\,k}] \leftarrow \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i^{k,1}, \ell_\rho)$, instead of $[\vec{c}_i^{\,k}] \leftarrow \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i^{k,0} + \vec{x}_i^{1,1} - \vec{x}_i^{1,0}, \ell_k)$, and returns $\mathsf{IP.Enc}(\mathsf{msk}_i, [\vec{c}_i^{\,k}])$. Note that this definition is compatible with $\mathbf{H}_0$

defined previously, and $\mathbf{H}_n$ is $\mathbf{G}_1$. Thus, it suffices to build a PPT adversary $\mathcal{B}_r$ for all $r \in [n]$, against the $\mathtt{IND}$-security of the $\mathsf{IP\text{-}FE}$, such that:

$$|\mathsf{Adv}_{\mathbf{H}_{r-1}}(\mathcal{A}) - \mathsf{Adv}_{\mathbf{H}_r}(\mathcal{A})| \leq \mathsf{Adv}^{\mathtt{IND}}_{\mathsf{IP\text{-}FE}}(\mathcal{B}_r).$$

We distinguish two cases. The first case occurs when $\mathcal{A}$ queries the user r to its oracle $\mathsf{QCorrupt}'$. Then, conditioned on the event that Finalize doesn't output a random bit, it must be the case that for all $k \in [Q_r]$, $\vec{x}_r^{k,0} = \vec{x}_r^{k,1}$. If we call E this first case, we have: $\mathsf{Adv}_{\mathbf{H}_{r-1}}(\mathcal{A} \wedge E) = \mathsf{Adv}_{\mathbf{H}_r}(\mathcal{A} \wedge E)$. The second case corresponds to the event $\neg E$: $\mathcal{A}$ does not query $\mathsf{QCorrupt}'$ on r. We build a PPT adversary $\mathcal{B}$ such that $|\mathsf{Adv}_{\mathbf{H}_{r-1}}(\mathcal{A}|\neg E) - \mathsf{Adv}_{\mathbf{H}_r}(\mathcal{A}|\neg E) \leq \mathsf{Adv}^{\mathtt{IND}}_{\mathsf{IP\text{-}FE}}(\mathcal{B}_r)$, which implies that $|\mathsf{Adv}_{\mathbf{H}_{r-1}}(\mathcal{A} \wedge \neg E) - \mathsf{Adv}_{\mathbf{H}_r}(\mathcal{A} \wedge \neg E) \leq \mathsf{Adv}^{\mathtt{IND}}_{\mathsf{IP\text{-}FE}}(\mathcal{B}_r)$. We conclude using the fact that for any game $\mathbf{G}$ and event E: $\mathsf{Adv}_{\mathbf{G}}(\mathcal{A}) = \mathsf{Adv}_{\mathbf{G}}(\mathcal{A} \wedge E) + \mathsf{Adv}_{\mathbf{G}}(\mathcal{A} \wedge \neg E)$. We now proceed to describe $\mathcal{B}_r$, which simulates the view of the $\mathtt{1\text{-}Label\text{-}IND}^*$-adversary $\mathcal{A}$ against MCFE' as follows:

- Initialize(ρ): $\mathcal{B}_r$ obtains $\mathsf{IP.mpk}_r$ from its own Initialize oracle, and generates $(\mathsf{IP.mpk}_i, \mathsf{IP.msk}_i) \leftarrow \mathsf{IP.SetUp}(\lambda)$ for all $i \neq r$, $(\mathsf{mpk}, \mathsf{msk}, (\mathsf{ek}_i)_i) \leftarrow \mathsf{SetUp}(\lambda)$ and returns $\mathsf{mpk}' := (\mathsf{mpk}, (\mathsf{IP.mpk}_i)_i)$ to $\mathcal{A}$.
- $\mathsf{QEncrypt}'(i, \vec{x}_i, \ell_j)$: it computes $[\vec{c}_i] \leftarrow \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i, \ell_j)$. If $i \neq r$, it returns $\mathsf{IP.Enc}(\mathsf{msk}_i, [\vec{c}_i])$; if $i = r$, it returns $\mathsf{QLeftRight}([\vec{c}_i], [\vec{c}_i])$.
- $\mathsf{QLeftRight}'(i, \vec{x}_i^{k,0}, \vec{x}_i^{k,1}, \ell_\rho)$: $\mathcal{B}$ computes $[\vec{c}_i^{k,0}] = \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i^{k,1}, \ell_\rho)$ and $[\vec{c}_i^{k,1}] = \mathsf{Encrypt}(\mathsf{ek}_i, \vec{x}_i^{k,0} + \vec{x}_i^{1,1} - \vec{x}_i^{1,0}, \ell_\rho)$, and uses its own $\mathsf{QLeftRight}$ oracle to output the ciphertext to $\mathcal{A}$
  - if $i < r$, it outputs $\mathsf{IP.Enc}(\mathsf{msk}_i, [\vec{c}_i^{k,0}])$;
  - if $i > r$, it outputs $\mathsf{IP.Enc}(\mathsf{msk}_i, [\vec{c}_i^{k,1}])$;
  - if $i = r$, it outputs $\mathsf{QLeftRight}([\vec{c}_i^{k,0}], [\vec{c}_i^{k,1}])$.
- $\mathsf{QDKeyGen}'(\vec{y})$: on input $\vec{y} := (\vec{y}_1 \| \cdots \| \vec{y}_n) \in \mathbb{Z}_p^{nm}$, $\mathcal{B}_r$ computes $\mathsf{dk}_{\vec{y}} = \mathsf{DKeyGen}(\mathsf{msk}, \vec{y})$, for all $i \neq r$: it computes $\mathsf{dk}_{\vec{y}_i} = \mathsf{IP.DKeyGen}(\mathsf{msk}_i, \vec{y}_i)$, and it queries its $\mathsf{QDKeyGen}$ oracle to obtain $\mathsf{QDKeyGen}(\vec{y}_r)$. It returns $\mathsf{dk}'_{\vec{y}} = \big(\mathsf{dk}_{\vec{y}}, \{\mathsf{dk}_{\vec{y}_i}\}_{i \in [n]}\big)$ to $\mathcal{A}$.
- $\mathsf{QCorrupt}(i)$: if $i = r$, $\mathcal{B}$ aborts the simulation and sends a random bit to its Finalize oracle. Otherwise, it returns $\mathsf{IP.msk}_i$.
- Finalize: $\mathcal{B}_r$ verifies that the conditions in Definition 21 are satisfied; if they are, it forwards the guess b' of $\mathcal{A}$, otherwise, it sends a random bit to its own Finalize oracle.

Note that when the random bit b used by the $\mathtt{IND}$-security game of $\mathsf{IP\text{-}FE}$ that $\mathcal{B}_r$ is interacting with is equal to 0, then, $\mathcal{B}_r$ simulates the game $\mathbf{H}_r$ to $\mathcal{A}$; otherwise, it simulates the game $\mathbf{H}_{r-1}$. In particular, the condition of the Finalize from Definition 21 implies that for all queries $(i, \vec{x}_i^{k_i,0}, \vec{x}_i^{k_i,1}, \ell_\rho)$ to $\mathsf{QLeftRight}'$, we have: $\sum_i \langle \vec{x}_i^{k_i,0}, \vec{y}_i \rangle = \sum_i \langle \vec{x}_i^{k_i,1}, \vec{y}_i \rangle$ for all $k_i \in [Q_i]$. Thus, we have in particular, for all $k \in [Q_r]$:

$$\begin{aligned}
&\langle \vec{x}_r^{k,0} - \vec{x}_\rho^{1,0}, \vec{y}_r \rangle = \langle \vec{x}_r^{k,1} - \vec{x}_\rho^{1,1}, \vec{y}_r \rangle \Rightarrow \\
&\langle \vec{x}_r^{k,0} + \vec{x}_r^{1,1} - \vec{x}_\rho^{1,0}, \vec{y}_r \rangle = \langle \vec{x}_r^{k,1} + \vec{x}_r^{1,1} - \vec{x}_\rho^{1,1}, \vec{y}_r \rangle \Rightarrow \\
&\langle \vec{c}_r^{k,0}, \vec{y}_r \rangle = \langle \vec{c}_r^{k,1}, \vec{y}_r \rangle,
\end{aligned}$$

where $[\vec{c}_r^{k,0}] = \mathsf{Encrypt}(\mathsf{ek}_r, (\vec{x}_r^{k,0} + \vec{x}_r^{1,1} - \vec{x}_r^{1,0}), \ell_\rho)$ and $[\vec{c}_r^{k,1}] = \mathsf{Encrypt}(\mathsf{ek}_r, (\vec{x}_r^{k,1} + \vec{x}_r^{1,1} - \vec{x}_r^{1,1}), \ell_\rho)$. The last implication uses the structural properties of the $\mathsf{IP\text{-}MCFE}$ scheme, namely, the property of linear homomorphism, and deterministic encryption. The last equality corresponds exactly to the condition to prevent the Finalize oracle from the $\mathtt{IND}$ security game of the $\mathsf{IP\text{-}FE}$ from outputting a random bit (see Definition 9).

This proves $|\mathsf{Adv}_{\mathbf{H}_{r-1}}(\mathcal{A}) - \mathsf{Adv}_{\mathbf{H}_r}(\mathcal{A})| \leq \mathsf{Adv}^{\mathtt{IND}}_{\mathsf{IP\text{-}FE}}(\mathcal{B}_r)$, and concludes the security proof.

C Chosen-Ciphertext Security: Proof

In this section, we formally prove the scheme described in 7 to achieve the $\mathtt{IND} - \mathtt{CCA}$ security level. The proof consists of a sequence of games starting from $\mathbf{G}_0$, playing the real security game described as above, to $\mathbf{G}_2$ where we are back to the basic $\mathtt{IND}$-security game, without decryption queries.

C.1 Signature

Before proving the scheme, as we will use strong unforgeability of signatures, let us recall the definition of a signature scheme, and the security notion:

Definition 23 (Signature Scheme). *A signature scheme on $\mathcal{M}$ is defined by four algorithms:*

- $\mathsf{SetUp}(\lambda)$*: Takes as input the security parameter λ, and outputs the public parameters* pp*;*
- $\mathsf{KeyGen}(\mathsf{pp})$*: Takes as input the public parameters and outputs a pair of signing and verification keys* $(\mathsf{sk}, \mathsf{vk})$*;*
- $\mathsf{Sign}(\mathsf{sk}, m)$*: Takes as input a signing key* sk *and a message $m \in \mathcal{M}$, and outputs a signature s;*
- $\mathsf{Verif}(\mathsf{vk}, m, s)$*: Takes as input the verification key* vk*, a message m and a signature s, and returns either 1 or 0, whether the signature is valid or not.*

Definition 24 (Strong Unforgeability). *Let us consider a signature scheme, no adversary $\mathcal{A}$ should be able to win the following security game against a challenger $\mathcal{C}$:*

- *Initialize: the challenger $\mathcal{C}$ runs the setup algorithm* $\mathsf{pp} \leftarrow \mathsf{SetUp}(\lambda)$ *as well as the key generation algorithm* $(\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$*, and provides* vk *to the adversary $\mathcal{A}$;*
- *Signature queries* $\mathsf{QSign}(m)$*: $\mathcal{A}$ has unlimited and adaptive access to the signing oracle, and receives the signature* $s \leftarrow \mathsf{Sign}(\mathsf{sk}, m)$*. One appends (m, s) to the list* $\mathsf{Queries}$*;*
- *Finalize: $\mathcal{A}$ provides a pair (m, s). $\mathcal{A}$ wins if* $\mathsf{Verif}(\mathsf{vk}, m, s)$ *when* $(m, s) \notin \mathsf{Queries}$.

We say Sign *is strongly unforgeable if for any adversary $\mathcal{A}$, the following value is negligible*

$$\mathsf{Adv}^{\mathit{SUF}}_{\mathsf{Sign}}(\mathcal{A}) = \Pr\begin{bmatrix} (m, s) \leftarrow \mathcal{A}^{\mathsf{QSign}}, \\ (m, s) \notin \mathsf{Queries}, \\ \mathsf{Verif}(\mathsf{pp}, m, s) = 1 \end{bmatrix}$$

C.2 Security Proof

Game $\mathbf{G}_0$: The real security game.

- Initialize: $\mathcal{C}$ runs both the algorithms $\mathsf{MCFE.SetUp}(\lambda)$ and $\mathsf{S.SetUp}(\lambda)$, to get respectively $(\mathsf{msk}, (\mathsf{ek}_i)_i, \mathsf{pp})$ and $(\mathsf{S.key}_i, \mathsf{S.pp}_i)_i$. Finally, $\mathcal{C}$ chooses a bit $b \xleftarrow{\$} \{0, 1\}$ and provides $\mathcal{A}$ the public parameters $\mathsf{mpk} = (\mathsf{pp}, (\mathsf{S.pp}_i)_i)$;
- Encryption queries $\mathsf{QEncrypt}(i, x_i, \ell)$: for each encryption query, $\mathcal{C}$ computes $A_{\ell,i} \leftarrow \mathsf{MCFE.Encrypt}(\mathsf{ek}_i, x_i, \ell)$ and $S_{\ell,i} \leftarrow \mathsf{S.Sign}(\mathsf{S.key}_i, (A_{\ell,i}, \ell))$ to return $\mathcal{A}$ the value $C_{\ell,i} = (A_{\ell,i}, S_{\ell,i})$;
- Challenge queries $\mathsf{QLeftRight}(i, x_i^0, x_i^1, \ell)$: for each encryption query, $\mathcal{C}$ computes $A_{\ell,i} \leftarrow \mathsf{MCFE.Encrypt}(\mathsf{ek}_i, x_i^b, \ell)$ and $S_{\ell,i} \leftarrow \mathsf{S.Sign}(\mathsf{S.key}_i, (A_{\ell,i}, \ell))$ to return $\mathcal{A}$ the value $C_{\ell,i} = (A_{\ell,i}, S_{\ell,i})$;
- Functional decryption key queries $\mathsf{QDKeyGen}(f)$: for each functional key query, $\mathcal{C}$ returns $\mathsf{dk}_f \leftarrow \mathsf{MCFE.QDKeyGen}(\mathsf{msk}, f)$;
- Corruption queries $\mathsf{QCorrupt}(i)$: for each corruption query, $\mathcal{C}$ returns $\widetilde{\mathsf{ek}_i} = (\mathsf{ek}_i, \mathsf{S.key}_i)$;
- Functional decryption queries $\mathsf{QFDecrypt}(\vec{C}, \ell, f)$: for each functional decryption query, $\mathcal{C}$ first checks the validity of the signatures parts S_i of C_i. If all the signatures are valid, recovers the decryption function key dk_f and runs $\mathsf{MCFE.FDecrypt}(\mathsf{dk}_f, \ell, \vec{C})$.

Game $\mathbf{G}_1$: In this game, $\mathcal{C}$ does as above, but rejects (outputting $\perp$) upon a query $\mathsf{QFDecrypt}(f, \ell, \vec{C})$ for which there is $i \in \mathcal{HS}$, and C_i is not the output of $\mathsf{QEncrypt}$ or $\mathsf{QLeftRight}$ on previous queries. This makes a difference with game $\mathbf{G}_0$ only if such C_i contains a valid signature, which would be a forgery. By simply guessing which client will be impersonated (which is correct with probability greater than $1/n$), one can output a forgery as soon as a difference happens:

$$|\mathsf{Adv}_{\mathbf{G}_1}(\mathcal{A}) - \mathsf{Adv}_{\mathbf{G}_0}(\mathcal{A})| \leq n \times \mathsf{Adv}_{\mathsf{S}}^{\mathtt{SUF}}(t).$$

Now we build a PPT adversary $\mathcal{B}$ such that: $\mathsf{Adv}_{\mathbf{G}_1}(\mathcal{A}) \leq \mathsf{Adv}_{\mathsf{MCFE}}^{\mathtt{IND}}(\mathcal{B})$, that is, we rely on the IND-CPA security of the underlying MCFE to conclude the CCA security proof. $\mathcal{B}$ samples the signature parameters $(\mathsf{S.key}_i, \mathsf{S.pp}_i)_i \leftarrow \mathsf{S.SetUp}(\lambda)$ for all $i \in [n]$, and uses its own oracle to simulate the oracles $\mathsf{QEncrypt}$, $\mathsf{QLeftRight}$, $\mathsf{QDKeyGen}$ and $\mathsf{QCorrupt}$. It answers queries $\mathsf{QFDecrypt}(f, \ell, \vec{C})$ as follows. If for some $i \in \mathcal{HS}$, C_i is not the output of $\mathsf{QEncrypt}$ or $\mathsf{QLeftRight}$ on previous queries, then output $\perp$. Otherwise, that means that for all $i \in \mathcal{HS}$, either C_i is the output of $\mathsf{QEncrypt}(i, x_i, \ell)$, in which case $\mathcal{B}$ sets $\widetilde{x_i} := x_i$; either C_i is the output of $\mathsf{QLeftRight}(i, x_i^0, x_i^1, \ell)$, in which case $\mathcal{B}$ sets $\widetilde{x_i} := x_i^0$. For all corrupted slots $i \in [n] \setminus \mathcal{HS}$, $\mathcal{B}$ computes $\widetilde{x_i} \leftarrow \mathsf{Decrypt}(\mathsf{ek}_i, \ell, \vec{C})$, which it can do since it knows ek_i, and $\vec{C}$ is a complete valid ciphertext. It finally outputs $f(\widetilde{\vec{x}} = (\widetilde{x_i})_i)$ to $\mathcal{A}$, which is correctly simulated, since the Finalize procedure imposes that the function f doesn't distinguish between x_i^0 and x_i^1 queried to $\mathsf{QLeftRight}$, that is, it doesn't matter whether we choose $\widetilde{x_i} := x_i^0$ or $\widetilde{x_i} := x_i^1$. This allows $\mathcal{B}$ to properly simulate the oracle $\mathsf{QFDecrypt}$ without knowing which bit b is chosen by $\mathsf{QLeftRight}$ in its own experiment.

D DSum: Proofs

In this section we provide the proofs of security for the two variants described in Section 8. Respectively under CDH assumption in the Random Oracle Model, and under DDH assumption in the standard model.

D.1 Security Analysis (in the Random Oracle Model)

We will prove that there exists a simulator that generates the view of the adversary from the output only. In this proof, we will assume static corruptions (the set $\mathcal{CS}$ of the corrupted clients is known from the beginning) and the hardness of the CDH problem. However, this construction will only tolerate up to $n-2$ corruptions, so that there are at least 2 honest users. But this is also the case for the MCFE.

W.l.o.g., we can assume that $\mathcal{HS} = \{1, \ldots, n-c\}$ and $\mathcal{CS} = \{n-c+1, \ldots, n\}$, by simply reordering the clients, when $\mathcal{CS}$ is known. We will gradually modify the behavior of the simulator, with less and less powerful queries. At the beginning, the $\mathsf{DSum.Encode}$-query takes all the same inputs as in the real game, including the secret keys. At the end, it should just take the sum (when all the queries have been asked), as well as the corrupted x_j's.

Game $\mathbf{G}_0$: The simulator runs as in the real game, with known $\mathcal{CS}$.

Game $\mathbf{G}_1$: The simulator is given a group $\mathbb{G}$ with a generator g and a random pair $(X = [t]; Y = [t^2])$.

– $\mathsf{DSum.SetUp}$: the simulator randomly chooses $\alpha_i \overset{\$}{\leftarrow} \mathbb{Z}_p$, for $i = 1, \ldots, n-c$, and defines $X_i \leftarrow X + [\alpha_i]$. This sets $t_i = t + \alpha_i$. It can also set $Y_{i,j} = \mathsf{CDH}(X_i, X_j) = Y + (\alpha_i + \alpha_j) \cdot X + [\alpha_i \alpha_j]$, for $i, j \leq n-c$. It then randomly choses $t_i \leftarrow \mathbb{Z}_p$ for $i > n-c$ and sets $X_i = [t_i]$. It can also generate all the other $Y_{i,j} = \mathsf{CDH}(X_i, X_j)$'s, using the known t_i's. It sends the X_i's as the pp, and the secret keys t_i of the corrupted users;
– $\mathsf{DSum.Encode}(x_i, \ell)$: the simulator generates all the required $h_{\ell,i,j}$ using the X_j's and $Y_{i,j}$'s, querying the hash function, and returns $M_{\ell,i} = x_i - \sum_{j<i} h_{\ell,i,j} + \sum_{j>i} h_{\ell,i,j}$.

Game $\mathbf{G}_2$: The simulator does as above, but just uses a random $Y' \overset{\$}{\leftarrow} \mathbb{G}$ instead of Y, to answer the $\mathsf{DSum.Encode}$-queries.

This can make a difference for the adversary if the latter asks for the hash function on some tuple $(X_{\min\{i,j\}}, X_{\max\{i,j\}}, \mathsf{CDH}(X_i, X_j), \ell)$, for $i, j \leq n - c$, as this will not be the value $h_{\ell,i,j}$, which has been computed using $Y_{i,j} \neq \mathsf{CDH}(X_i, X_j)$. In such a case, one can find $\mathsf{CDH}(X_i, X_j) = Y + [\alpha_i + \alpha_j] \cdot X + [\alpha_i \alpha_j]$ in the list of the hash queries, and thus extract $Y = \mathsf{CDH}(X, X)$. As a consequence, under the hardness of the square Diffie-Hellman problem (which is equivalent to the CDH problem), this simulation is indistinguishable from the previous one.

Game $\mathbf{G}_3$: The simulator does as above except for the $\mathsf{DSum.Encode}$-queries. If this is not the last-honest query under label ℓ, the simulator returns $M_{\ell,i} = -\sum_{j<i} h_{\ell,i,j} + \sum_{j>i} h_{\ell,i,j}$; for the last honest query, it returns $M_{\ell,i} = S_H - \sum_{j<i} h_{\ell,i,j} + \sum_{j>i} h_{\ell,i,j}$, where $S_H = \sum_{j \in \mathcal{HS}} x_j$.

Actually, for a label ℓ, if we denote i_ℓ the index of the honest player involved in the last query, the view of the adversary is exactly the same as if, for every $i \neq i_\ell$, we have replaced h_{ℓ,i,i_ℓ} by $h_{\ell,i,i_\ell} + x_i$ (if $i_\ell > i$) or by $h_{\ell,i,i_\ell} - x_i$ (if $i_\ell < i$). We thus replace uniformly distributed variables by other uniformly distributed variables: this simulation is perfectly indistinguishable from the previous one.

Game $\mathbf{G}_4$: The simulator now ignores the values $h_{\ell,i,j}$ for honest i, j. But for each label, it knows the corrupted x_j's, and can thus compute the values $M_{\ell,j}$ for the corrupted users, using the corrupted x_j's and secret keys. If this is not the last honest query, it returns a random $M_{\ell,i}$. For the last honest query, knowing $S = \sum_j x_j$, it outputs $M_{\ell,i} = S - \sum_{j \neq i} M_{\ell,j}$.

As in the previous analysis, if one first sets all the $h_{\ell,i,j}$, for $j \neq i_\ell$, this corresponds to define h_{ℓ,i,i_ℓ} from $M_{\ell,i}$, for $i \neq i_\ell$.

D.2 Security Analysis (in the Standard Model)

In the previous section, we observe that we do not exploit programmability of the random oracle, and can actually use the Decisional Diffie-Hellman assumption to prove it in the standard model. The key used $K_{i,j}$ for $\mathcal{F}$ is $\mathcal{E}([t_i t_j])$, where $\mathcal{E}$ is a randomness extractor, and the input is ℓ. We still assume that $\mathcal{HS} = \{1, \ldots, n - c\}$.

Game $\mathbf{G}_0$: The simulator runs as in the real game, with known $\mathcal{CS}$ (assumed to be $\{n - c + 1, \ldots, n\}$, without loss of generality, since we are in the static corruption setting).

Game $\mathbf{G}_1$: The simulator does as above, but just uses a random value $Y_{i,j} \stackrel{\$}{\leftarrow} \mathbb{G}$ instead of the key $[t_i t_j]$, when both $i \neq j \in \mathcal{HS}$, to generate the $K_{i,j}$'s to answer the $\mathsf{DSum.Encode}$-queries. After the hybrid sequence described below, the advantage for the adversary is:

$$|\mathsf{Adv}_{\mathbf{G}_0}(\mathcal{A}) - \mathsf{Adv}_{\mathbf{G}_1}(\mathcal{A})| \leq \frac{(n-c)^2}{2} \cdot \mathsf{Adv}^{\mathsf{ddh}}(\mathcal{B}),$$

for some adversary $\mathcal{B}$ running with a similar time as $\mathcal{A}$.

Game $\mathbf{G}_2$: The simulator now uses random keys $K_{i,j}$'s in the cases $i < j$ are both honest, and $K_{j,i} = K_{i,j}$. Because of the entropy on the $Y_{i,j}$'s, the Left-over-Hash Lemma guarantees a statistical indistinguishability with the previous game.

Game $\mathbf{G}_3$: The simulator now chooses random $h_{\ell,i,j}$ for any ℓ, in the cases $i < j$ are both honest, and $h_{\ell,j,i} = h_{\ell,i,j}$. Under the indistinguishability of the PRF with random keys, this game is indistinguishable from $\mathbf{G}_2$.

Now, the rest of the proof is similar to the previous one, with a final simulation as in above game $\mathbf{G}_4$.

Hybrid Sequence: Here we present the hybrid games $\mathbf{H}_{i,j,k}$ between $\mathbf{G}_0$ and $\mathbf{G}_1$. An iteration of this sequence describes how to replace the value $[t_{i^*} t_{j^*}]$ used in the setup phasis, for honest $i^* < j^*$, by random $Y_{i^*,j^*} \stackrel{\$}{\leftarrow} \mathbb{G}$. The progression follows the lexicographical order on the pairs

$(i,j) \in \mathcal{HS}$ where $i < j$, and $\mathsf{Succ}(i,j)$ denotes the next pair. It will be clear that $\mathbf{G}_0 = \mathbf{H}_{1,2,0}$ and $\mathbf{G}_1 = \mathbf{H}_{n-c-1,n-c,3}$. In addition, for all $(1,2) \leq (i^*,j^*) < (n-c-1,n-c)$, $\mathbf{H}_{i^*,j^*,3} = \mathbf{H}_{\mathsf{Succ}(i^*,j^*),0}$. We indeed insist that $K_{i,i}$ is never used, so only Diffie-Hellman values for two different keys are used.

Game $\mathbf{H}_{i^*,j^*,0}$: The simulator runs the real game, except that it additionally initializes $Y_{i,j}$ in the $\mathsf{DSum.SetUp}$, used for the extracted keys $K_{i,j} = \mathcal{E}(Y_{i,j})$ during the $\mathsf{DSum.Encode}$, either correctly as $[t_i t_j]$ or at random:

– $\mathsf{DSum.SetUp}$: after having generated the group $\mathbb{G}$ of prime order p, with a generator g, the randomness extractor $\mathcal{E}(\cdot)$, and the PRF $(\mathcal{F}_K)_K$, the simulator generates the secret keys $t_i \overset{\$}{\leftarrow} \mathbb{Z}_p$ and sets $X_i \leftarrow [t_i]$, for all i. Then it defines:
  - for $(i,j) < (i^*,j^*)$, where $i < j$ are both honest, pick a random element $Y_{i,j} \overset{\$}{\leftarrow} \mathbb{G}$
  - for $(i,j) \geq (i^*,j^*)$, where $i < j$ are both honest, set $Y_{i,j} \leftarrow [t_i t_j]$
  - for (i,j) where $i < j$ and some of them is corrupted, set $Y_{i,j} \leftarrow [t_i t_j]$
  - for (i,j) where $i > j$, set $Y_{i,j} \leftarrow Y_{j,i}$

It sends the X_i's as the pp, and the secret keys t_i of the corrupted users;

Game $\mathbf{H}_{i^*,j^*,1}$: for $i^* < j^*$, the simulator is given a group $\mathbb{G}$ with a generator g and a random Diffie-Hellman tuple $(X = [x], Y = [y], Z = [xy])$.

– $\mathsf{DSum.SetUp}$: it uses the above group $\mathbb{G}$ and generator g, and generates $\mathcal{E}$ and $(\mathcal{F}_K)_K$. For the indices i^*, j^*, the simulator defines $X_{i^*} \leftarrow X$ and $X_{j^*} \leftarrow Y$. This sets $t_{i^*} \leftarrow x$ and $t_{j^*} \leftarrow y$. It can also set $Y_{i^*,j^*} = \mathsf{CDH}(X_{i^*}, X_{j^*}) = Z$. It then randomly chooses $t_i \overset{\$}{\leftarrow} \mathbb{Z}_p$ for $i \neq i^*, j^*$ and sets $X_i \leftarrow [t_i]$. It can also generate $Y_{i,j} = \mathsf{CDH}(X_i, X_j)$, using the known t_i, for $(i,j) > (i^*,j^*)$ and $i < j$. The cases $(i,j) < (i^*,j^*)$ for $i < j$ and the cases $i > j$ remain unchanged. It sends the X_i's as the pp, and the secret keys t_i of the corrupted users;

The view of the adversary remains the same.

Game $\mathbf{H}_{i^*,j^*,2}$: for $i^* < j^*$, the simulator is given a random tuple $(X = [x], Y = [y], Z \overset{\$}{\leftarrow} \mathbb{G})$, and does as above. Under the hardness of the Decisional Diffie-Hellman problem, this simulation is indistinguishable from the previous one.

Game $\mathbf{H}_{i^*,j^*,3}$: this is quite similar to game $\mathbf{H}_{i^*,j^*,0}$, but with difference for $(i,j) = (i^*,j^*)$:

– $\mathsf{DSum.SetUp}$: after having generated the group $\mathbb{G}$ of prime order p, with a generator g, the randomness extractor $\mathcal{E}(\cdot)$, and the PRF $(\mathcal{F}_K)_K$, the simulator generates the secret keys $t_i \overset{\$}{\leftarrow} \mathbb{Z}_p$ and sets $X_i \leftarrow [t_i]$, for all i. Then it defines:
  - for $(i,j) \leq (i^*,j^*)$, where $i < j$ are both honest, pick a random element $Y_{i,j} \overset{\$}{\leftarrow} \mathbb{G}$
  - for $(i,j) > (i^*,j^*)$, where $i < j$ are both honest, set $Y_{i,j} \leftarrow [t_i t_j]$
  - for (i,j) where $i < j$ and some of them is corrupted, set $Y_{i,j} \leftarrow [t_i t_j]$
  - for (i,j) where $i > j$, set $Y_{i,j} \leftarrow Y_{j,i}$

The view of the adversary does not change.

Starting from $(1,2)$ up to $(n-c-1, n-c)$, there are $(n-c)(n-c-1)/2$ cases with $i^* < j^*$ which involve the DDH assumption, hence the conclusion.