Registered (Inner-Product) Functional Encryption

Danilo Francati[1], Daniele Friolo[2], Monosij Maitra[3,5], Giulio Malavolta[4,5(✉)], Ahmadreza Rahimi[5], and Daniele Venturi[2]

[1] Aarhus University, Aarhus, Denmark
dfrancati@cs.au.dk
[2] Sapienza University of Rome, Rome, Italy
{friolo,venturi}@di.uniroma1.it
[3] Ruhr-Universität Bochum, Bochum, Germany
monosij.maitra@rub.de
[4] Bocconi University, Milan, Italy
[5] Max-Planck Institute for Security and Privacy, Bochum, Germany
giulio.malavolta@unibocconi.it, ahmadrezar@pm.me

Abstract. Registered encryption (Garg *et al.*, TCC'18) is an emerging paradigm that tackles the key-escrow problem associated with identity-based encryption by replacing the private-key generator with a much weaker entity known as the key curator. The key curator holds no secret information, and is responsible to: (i) update the master public key whenever a new user registers its own public key to the system; (ii) provide helper decryption keys to the users already registered in the system, in order to still enable them to decrypt after new users join the system. For practical purposes, tasks (i) and (ii) need to be efficient, in the sense that the size of the public parameters, of the master public key, and of the helper decryption keys, as well as the running times for key generation and user registration, and the number of updates, must be small.

In this paper, we generalize the notion of registered encryption to the setting of functional encryption (FE).

As our main contribution, we show an efficient construction of registered FE for the special case of (*attribute hiding*) inner-product predicates, built over asymmetric bilinear groups of prime order. Our scheme supports a *large* attribute universe and is proven secure in the bilinear generic group model. We also implement our scheme and experimentally demonstrate the efficiency requirements of the registered settings. Our second contribution is a feasibility result where we build registered FE for P/poly based on indistinguishability obfuscation and somewhere statistically binding hash functions.

Keywords: Registered encryption · functional encryption · inner-product predicate encryption

J. Guo and R. Steinfeld (Eds.): ASIACRYPT 2023, LNCS 14442, pp. 98–133, 2023.
https://doi.org/10.1007/978-981-99-8733-7_4

1 Introduction

Functional encryption (FE) [18,51,53] enriches standard public-key encryption with fine-grained access control over encrypted data. This added feature is made possible by having a so-called master secret key msk that can be used (by an authority) to generate decryption keys sk_f associated with functions f, in such a way that decrypting any ciphertext c, corresponding to a plaintext m, reveals $f(m)$ and nothing more. Recent years have seen a flourish of works exploring FE constructions in various settings and from different assumptions [1–3,5,6,8,9,17,19,20,27,28,32–34,36,37,43–45,48,52,56], and its applications to building powerful cryptographic tools such as reusable garbled circuits [36], adaptive garbling [40], multi-party non-interactive key exchange [31], universal samplers [31], verifiable random functions [15,38], and indistinguishability obfuscation (iO) [7,16] (which, in turn, implies a plethora of other cryptographic primitives [52]).

An important limitation of FE is the well-known *key escrow* problem: the authority holding the master secret key (sometimes referred to as the private key generator – PKG) can generate secret keys for any function, allowing it to arbitrarily decrypt messages intended for specific recipients. This requires a fully trusted PKG which severely restricts the applicability of FE in many scenarios.

Registered Encryption. A recent line of research proposes to tackle the key-escrow problem in the much simpler case of identity-based encryption[1] (IBE) [54]. This led to the notion of *registered* IBE (RIBE) [29][2], where the main idea is to replace the PKG with a much weaker entity called the key curator (KC), whose role is to register the public keys of the users (without possessing any secret key). In particular, in a RIBE scheme there is an initial setup phase in which a common reference string (CRS) is sampled. The CRS is given to the KC which publishes an (initially empty) master public key. Each user now can also use the CRS and sample its own public and secret key, and can register its identity and the chosen public key to the KC; the KC is required to generate a new master public key, which includes the newly registered public keys, and which will permit encrypting messages to any of the registered users. Moreover, since the master public key is updated over time, the KC is responsible for providing any decrypting party with a so-called helper decryption key, i.e., auxiliary information connecting its public key with the updated master public key.

Recently, the notion of RIBE has been extended to the setting of attribute-based encryption (ABE) [41], where one can encrypt messages with respect to policies, and where decryptors can recover the message if their attributes satisfy the policy embedded in the ciphertext. However, their registered ABE (RABE)

[1] IBE can be seen as a special case of FE for equality predicates f_y such that $f_y(x, m) = m$ if and only if $y = x$ (and $\perp$ otherwise). Here, x and y have the role of the parties' identities (which do not need to be secret), and m is the encrypted message.

[2] The original paper define the primitive as registration based encryption. However, we choose to call it as registered IBE, in line with the more recent work in [41].

schemes [41] are required to hide only messages in the ciphertext. In particular, they do not hide the policies embedded in the ciphertexts, since they are required in the clear for decryption to work. This restricts using RABE in scenarios where hiding the policy is also important.

More generally, the current state of affairs leaves open the question of building registered FE (RFE), where any user can sample its own key pair $(\mathsf{pk}, \mathsf{sk})$ as before, along with fixing a function of its choice (say f, from a class of functions), and register (pk, f) with the KC. In such a setting, one can then encrypt messages m that the registered user can decrypt with sk and a helper secret key to learn only $f(m)$. Overall, this would achieve the analogous functionality to that of the celebrated notion of FE, without suffering from the key escrow issue. The focus of our work is to make progress on this problem.

1.1 Our Contributions

We initiate the study of RFE in this paper by providing two constructions – one for a special class of FE, and another for the general class of all functions.

In particular, as our first contribution, we provide the *first* RFE scheme for the class of inner-product predicates (a.k.a. (attribute hiding) inner-product predicate encryption), i.e., a registered IPE (RIPE) from asymmetric bilinear maps on prime-order groups. More concretely, our scheme supports the function class $\mathcal{F} = \{f_{\mathbf{x}}(\cdot,\cdot)\}_{\mathbf{x}\in\mathbb{Z}_q^{n^+}}$ defined as:

$$f_{\mathbf{x}}(m, \mathbf{y}) = \begin{cases} m & \text{if } \langle \mathbf{x}, \mathbf{y} \rangle = 0 \\ \perp & \text{otherwise} \end{cases} \tag{1}$$

where $\mathbf{x}$ and $\mathbf{y}$ are n-size vectors over $\mathbb{Z}_q^{n^+} = \mathbb{Z}_q^n \backslash \{\mathbf{0}^n\}$, and q is a prime. Below we summarize our result informally in Theorem 1 and also later in Table 1 (Sect. 3) when we discuss related works to compare it with existing registered encryption schemes.

Theorem 1 (Informal). *Let λ be a security parameter, n be the length of supported vectors, and L be a bound on the maximum number of users. There is a (black-box) construction of RIPE supporting a large universe and up to L users in the generic bilinear group model, satisfying the following properties:*

- *The CRS is of size $n \cdot L^2 \cdot \mathsf{poly}(\lambda, \log L)$.*
- *The master public key and each helper decryption key is of size $n \cdot \mathsf{poly}(\lambda, \log L)$.*
- *Key-generation and registration runs in time $L \cdot \mathsf{poly}(\lambda, \log L)$ and $n \cdot L^2 \cdot \mathsf{poly}(\lambda, \log L)$, respectively.*
- *Each registered user receives at most $O(\log L)$ updates from the KC over the entire lifetime of the system.*

Moreover, both encryption and decryption runs in time $n \cdot \mathsf{poly}(\lambda, \log L)$.

Our scheme is proven secure in the bilinear generic group model [12,14]. We emphasize that our scheme supports *attribute-hiding* and a *large universe* unlike [41]. In particular, our scheme satisfies the strong notion of *two-sided* security[3] [26,46], where no information on the attribute vector $\mathbf{y}$ is revealed (besides the orthogonality test) even if decryption succeeds, akin to what [46] achieved.[4]

Somewhat interestingly, our proof strategy and construction template are substantially different from the typical inner-product predicate encryption schemes in the literature (e.g., [46]). Roughly speaking, traditional proof strategies work by "programming" the function output (for the challenge ciphertext) in the key given by the adversary, and then arguing that this new key is indistinguishable from the original distribution. In the registered setting, the adversary can sample its own key, so the reduction has no control over it and cannot modify its distribution. Hence, we see RIPE as the main technical contribution of this work.

We also implemented our scheme and describe the results in Sect. 7. The benchmarks are achieved with a set of $L = 100$ to $L = 1000$ users with attribute vectors of length varying between $n = 10$ and $n = 100$. Our results demonstrate concrete, practical efficiency of our scheme beyond the realms of only feasibility. Further, following the *generic* and *non-cryptographic* transformations described in [46, Section 5], our RIPE scheme can also support constant-degree polynomial evaluations, disjunctions, conjunctions, and evaluating CNF and DNF formulas.

As our second contribution, we build RFE for all circuits from indistinguishability obfuscation (iO). This is a feasibility result extending the iO-based RABE schemes in [41] to the setting of RFE. In more detail, we achieve the following:

Theorem 2 (Informal). *Let λ be the security parameter. Assuming somewhere statistically binding hash functions [42,50] and iO [13], there is a (non black-box) construction of RFE supporting arbitrary functions and an arbitrary number of users, satisfying the following properties:*

- *The CRS, master public key, and each helper decryption key is of size* $\mathsf{poly}(\lambda)$.
- *Key-generation and registration runs in time* $\mathsf{poly}(\lambda)$ *and* $L \cdot \mathsf{poly}(\lambda)$*, respectively, where L stands for the* current *number of registered users.*
- *Each registered user receives at most $O(\log L)$ updates from the KC over the entire lifetime of the system, where L is as defined in the previous item.*

Moreover, both encryption and decryption runs in time $\mathsf{poly}(\lambda)$*. Further, the above scheme achieves the same efficiency as that of iO-based RABE from [41].*

[3] Two-sided security in PE allows an adversary to obtain secret keys for predicates that *can* decrypt a challenge ciphertext, provided the challenge message pair consists of the same message.

[4] Generic compilers from any ABE for LSSS (or equivalently, monotone span programs) to (hierarchical) IPE are known (e.g., [11]). However, such compilers *do not ensure* attribute-privacy which we crucially require from our (registered) IPE scheme.

2 Technical Overview

In the following, we first describe the notion of registered FE and its properties of interest. Next, we provide a brief overview of the techniques behind our schemes.

RFE Definition. We discuss the notion of RFE at a high level. Fundamentally, RFE allows users to generate their own keys (associated to functions of their choice) without the need of a trusted authority, which is replaced with a KC that does not hold any secret. The KC is simply responsible of managing a data structure containing the public keys (plus the corresponding functions) of registered users. Roughly, the RFE syntax goes as follows: For some security parameter λ and a function class $\mathcal{F}$, the algorithm $\mathsf{Setup}(1^n, |\mathcal{F}|)$ initializes the system to output a common reference string crs.[5] Given crs, the KC initializes a state $\alpha = \perp$ (i.e., the data structure) and the master public key $\mathsf{mpk} = \perp$. A user can now register its own (pk, f) pair as follows: it samples $(\mathsf{pk}, \mathsf{sk}) \leftarrow\!\!\$\ \mathsf{KGen}(\mathsf{crs}, \alpha)$ and submits a registration request (pk, f) to the KC, where $f \in \mathcal{F}$ is a function it wishes to associate with pk. The KC updates its state as $\alpha = \alpha'$ and $\mathsf{mpk} = \mathsf{mpk}'$ where (mpk', α') are output by the *deterministic* registration algorithm $\mathsf{RegPK}(\mathsf{crs}, \alpha, \mathsf{pk}, f)$. Intuitively, a ciphertext $c \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{mpk}, m)$ computed with mpk can be later decrypted by the users registered before or during mpk was generated. The registered user uses sk to decrypt c. However, mpk is updated periodically (after each registration) – so the user issues an update request to the KC that, in turn, *deterministically* returns a helper secret key $\mathsf{hsk} = \mathsf{Update}(\mathsf{crs}, \alpha, \mathsf{pk})$. The hsk provides necessary information to make a (previously registered) user's secret key sk valid with respect to a new mpk. With hsk, the user can decrypt to learn $f(m) = \mathsf{Dec}(\mathsf{sk}, \mathsf{hsk}, c)$. For optimal efficiency, an RFE system with L registered users should satisfy the following properties:

(1) *Compact parameters:* The sizes of $\mathsf{crs}, \mathsf{mpk}, \mathsf{hsk}$ must be small, e.g., $\mathsf{poly}(\lambda, \log L)$.
(2) *Efficiency:* This measures key-generation and registration runtimes, and the number of updates as described below.
   (a) Each execution of KGen and RegPK should run in time $\mathsf{poly}(\lambda, \log L)$.
   (b) Each registered user receives at most $O(\log L)$ number of new updates (i.e., new hsks) over the lifetime of the system.

RFE can support an unbounded or a bounded number of users. In particular, for the unbounded case, the setup is independent of the number of users. (In this case, the parameter L in efficiency conditions refer to the *current* number of registered users.) For the bounded case, the setup depends on a bound L (fixed a-priori). Security of RFE is analogous to that of RIBE [29] and RABE [41]. In particular, an adversary A corrupting a subset of k registered users (i.e., A knows the

[5] Although the common reference string is generated by a trusted setup, the important difference is that there is no long-term secret that needs to be stored throughout the lifetime of the system. Furthermore, in some cases, the setup algorithm could be "transparent", and therefore computable using just a hash function.

set $\{(\mathsf{sk}_i, (\mathsf{pk}_i, f_i))\}_{i\in[k]}$) cannot distinguish $\mathsf{Enc}(\mathsf{mpk}, m_0)$ from $\mathsf{Enc}(\mathsf{mpk}, m_1)$, as long as $f_i(m_0) = f_i(m_1), \forall i \in [k]$. This should hold even if A registers *malformed* public keys. We refer to the full version [25] for more details.

Slotted RFE. Following Hohenberger *et al.* [41], we first define and use *slotted* RFE as a stepping stone towards building full-fledged RFE. Differently to RFE, there is only a single update (referred to as *aggregation*) in slotted RFE, where users are assigned to "slots" and the master public key is only computed once all slots are filled. In more detail, initialization and key generation work as before, except now that the Setup (resp. KGen) takes as an extra input the maximum number of slots/keys L that can be aggregated (resp. a user index $i \in [L]$). The KC takes all L pairs $\{(\mathsf{pk}_i, f_i)\}_{i\in[L]}$ *together*, aggregrates (i.e. updates) it to compute a short mpk and L helper secret keys $\{\mathsf{hsk}_i\}_{i\in[L]}$ for each user. Encryption and decryption again works as before.

Akin to RFE, slotted RFE security requires that, for an aggregated mpk w.r.t. to all L slots, $\mathsf{Enc}(\mathsf{mpk}, m_0)$ and $\mathsf{Enc}(\mathsf{mpk}, m_1)$ are computationally indistinguishable, so long as $f_j(m_0) = f_j(m_1)$ for all *corrupted* slots $j \in [L]$. We refer to the full version [25] for more details.

Hohenberger *et al.*[41] lifted slotted RABE to a standard RABE via a generic compiler, and the same holds for slotted RFE (with minor syntactic changes). Loosely speaking, they use a "powers-of-two" approach, where users are assigned to different slotted schemes with increasing capacities, and they are moved forward as new users join the system. The same idea yields a fully-fledged RFE that supports $O(\log L)$ number of updates and incurs a multiplicative $O(\log L)$ overhead on the size of $\mathsf{crs}, \mathsf{mpk}, \mathsf{hsk}$, and the key-generation and encryption runtimes compared to that of the underlying slotted RFE scheme. The registration runtime is dominated by $O(t_{\mathsf{Aggr}} + L \cdot t_{\mathsf{hsk}})$, where t_{Aggr} and t_{hsk} are the aggregation runtime and the helper decryption key size of the slotted RFE respectively. For completeness, we present the transformation in our full version [25].

2.1 (Bounded Users) Slotted RIPE from Pairings

We begin with an overview of our scheme for inner-product predicates. This is a special case of FE, where vectors $\mathbf{x} \in \mathbb{Z}_q^{n^+} (= \mathbb{Z}_q^n \setminus \{\mathbf{0}^n\})$ denote functions $f_{\mathbf{x}}$ (associated to keys), and messages consist of a tuple $(\mathbf{y}, m)$. The function $f_{\mathbf{x}}$ can be recast as:

$$f_{\mathbf{x}}(\mathbf{y}, m) = \begin{cases} m & \text{if } \langle \mathbf{x}, \mathbf{y} \rangle = 0 \\ \perp & \text{otherwise} \end{cases}$$

where we denote the length of vectors by $n = n(\lambda)$, and assume the attribute space to be $\mathcal{U} = \mathbb{Z}_q^{n^+}$ (i.e., domain of vectors). Our scheme follows the blueprint of [41]. However, unlike [41], that reveals the policy in clear, achieving attribute-hiding security in this setting of predicate encryption requires us to introduce

crucial modifications, which we highlight after the overview of our scheme below. Furthermore, the security analysis is completely different.

Single-Slot Scheme. We begin by discussing a simplified scheme with $L = 1$ (i.e., there is a single slot). Below is a description of each algorithm in the scheme.

- **Generating the CRS:** We first describe the CRS generation. The CRS can be split into three different parts, a general part, a slot-specific part, and a key-specific part. We will describe how each part is generated individually.
  - *General part*: First, we generate an asymmetric pairing group of prime order q, denoted as $\mathcal{G} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathsf{T}, q, g_1, g_2, e)$. Then, we sample $\alpha, \beta, \gamma \leftarrow_\$ \mathbb{Z}_q$ and set $h = g_1^\beta, Z = e(g_1, g_2)^\alpha$. (We will need γ for the multi-slot scheme, which we describe later.)
  - *Slot-specific part*: We associate each slot with a set of group elements, for this case we sample $t \leftarrow_\$ \mathbb{Z}_q$ and set $A = g_2^t$ and $B = g_2^\alpha A^\beta = g_2^{\alpha+\beta t}$.
  - *Key-specific part*: We also associate a group element to each component of the key vector, plus the secret key. To do this, for each $w \in [n+1]$, we sample $u_w \leftarrow \mathbb{Z}_q$ and set $U_w = g_1^{u_w}$.

  In the end, we set the CRS to be:

$$\mathsf{crs} = \left(\mathcal{G}, Z, h, A, B, \{U_w\}_{w\in[n+1]}\right).$$

- **Generating keys:** To compute a new pair of public/secret keys, we sample a non-zero secret key $\mathsf{sk} \leftarrow_\$ \mathbb{Z}_q$ and set $\mathsf{pk} = U_{n+1}^{-\mathsf{sk}}$. Note that we are conceptually treating the secret key as one more element of the predicate vector. This is an important structural difference with respect to [41].
- **Key Aggregation:** Since we only have one slot, given pk and crs, and a predicate vector (or key) $\mathbf{x} = (x_1, \ldots, x_n)$, we set the master public key as:

$$\mathsf{mpk} = \left(\mathcal{G}, h, Z, \{U_w\}_{w\in[n+1]}, \mathsf{pk} \cdot \prod_{w=1}^{n} U_w^{-x_w}\right).$$

- **Encryption:** To encrypt a message $m \in \mathbb{G}_\mathsf{T}$ with respect to a non-zero attribute vector $\mathbf{y} = (y_1, \ldots, y_n) \in \mathbb{Z}_q^{n^+}$, and the master public key mpk, we create a ciphertext that has two components, a message-embedding component, and a key-slot-embedding component.
  - *Message embedding:* We sample $s \leftarrow_\$ \mathbb{Z}_q^*$, and set $C_1 = m \cdot Z^s, C_2 = g_1^s$.
  - *Key-slot embedding:* First, we sample $r, z \leftarrow_\$ \mathbb{Z}_q \setminus \{0\}$. Then, we set

$$C_{3,w} = h^{y_w \cdot r + s} \cdot U_w^{-z} \ (\forall w \in [n]), \quad C_{3,n+1} = h^s \cdot U_{n+1}^{-z}, \quad \text{and}$$

$$C_{3,n+2} = h^s \cdot \mathsf{pk}^{-z} \prod_{w=1}^{n} U_w^{z \cdot x_w}.$$

  The final ciphertext will be $(C_1, C_2, \{C_{3,w}\}_{w\in[n+1]})$.

– **Decryption:** Before describing the actual decryption, let us check the intuition behind each element of the ciphertext. The first component $C_1 = m \cdot Z^s$ is just a masking of the message with a random power of Z from the CRS. Consider B from crs, and the ciphertext components C_1 and C_2, and observe:

$$\frac{C_1}{e(C_2, B)} = \frac{m \cdot e\left(g_1, g_2\right)^{\alpha \cdot s}}{e\left(g_1, g_2\right)^{\alpha \cdot s} \cdot e\left(g_1, g_2\right)^{s\beta t}} = \frac{m}{e\left(h^s, A\right)}.$$

Thus, to recover the message, it suffices to recompute $e(h^s, A)$. Note that h^s is already present in some form in the $C_{3,*}$ components. We can partition $C_{3,*}$ terms into three different groups, and see how h^s appears in each one:

1. For all $w \in [n]$, we have $C_{3,w} = h^s \cdot h^{y_w \cdot r} \cdot U_w^{-z}$. In this case, there are extra terms $y_w \cdot r$ as well as U_w present in the ciphertext. However, since $\mathbf{x}$ and $\mathbf{y}$ are orthogonal (otherwise decryption fails), we can eliminate these extra terms by raising each $C_{3,w}$ to the power of x_w for $w \in [n]$ and compute their product. Thus, we will have:

$$\begin{aligned}\prod_{w=1}^{n} C_{3,w}^{x_w} &= \prod_{w=1}^{n} h^{x_w \cdot s} \cdot h^{x_w \cdot y_w \cdot r} \cdot \prod_{w=1}^{n} U_w^{-z \cdot x_w} \\ &= h^{s \cdot \sum_{w=1}^{n} x_w} \cdot \underbrace{h^{r \cdot \sum_{w=1}^{n} x_w \cdot y_w}}_{=1} \cdot \prod_{w=1}^{n} U_w^{-z \cdot x_w}.\end{aligned}$$

Therefore, we are left with two terms $h^{s \cdot \sum_{w=1}^{n} x_w}$ and $\prod_{w=1}^{n} U_w^{-z \cdot x_w}$.
2. For $w = n+1$, we have $C_{3,n+1} = h^s \cdot U_{n+1}^{-z}$, where the term h^s is masked with U_{n+1}^{-z}.
3. For $w = n+2$, we have $C_{3,n+2} = h^s \cdot \mathsf{pk}^{-z} \prod_{w=1}^{n} U_w^{z \cdot x_w} = h^s \cdot U_{n+1}^{z \cdot \mathsf{sk}} \cdot \prod_{w=1}^{n} U_w^{z \cdot x_w}$.
Multiplying together the remaining components we obtain:

$$C_{3,n+2} \cdot C_{3,n+1}^{\mathsf{sk}} \cdot \prod_{w=1}^{n} C_{3,w}^{x_w} = h^s \cdot h^{s \cdot \mathsf{sk}} \cdot h^{s \cdot \sum_{w=1}^{n} x_w} = h^{s \cdot \left(1 + \mathsf{sk} + \sum_{w=1}^{n} x_w\right)}.$$

The decryptor can now raise $h^{s \cdot \left(1 + \mathsf{sk} + \sum_{w=1}^{n} x_w\right)}$ to the power of $(1 + \mathsf{sk} + \sum_{w=1}^{n} x_w)^{-1}$ to get h^s. Once h^s is obtained, it can be paired with A, available from crs, to decrypt the message.

Multi-slot Scheme. To gain an intuition on how our scheme handles multiple slots, we describe a toy example where $L = 2$, i.e., we are in the two-slot setting. Notice that one trivial generalization is to individually generate public keys as before, and concatenate them into the master public key. However, this approach will not work, since we want the master public key size to be independent of the number of slots. Instead, we expand the slot-specific components in the CRS to A_1, B_1 (for slot 1) and A_2, B_2 (for slot 2), which are generated in the same way as A, B in the one-slot setting, but using independent random elements $t_1, t_2 \leftarrow_{\$} \mathbb{Z}_q$

in generating A_1, A_2. We will also need to link the slots to the keys, so that we can use the slot in the key-generation algorithm. For this, instead of generating only one set of $\{U_w\}_{w\in[n]}$, we generate them with respect to both slots

$$\{U_{w,1} = g_1^{u_{w,1}}\}_{w\in[n+1]} \qquad \text{and} \qquad \{U_{w,2} = g_1^{u_{w,2}}\}_{w\in[n+1]}$$

where the elements $\{u_{w,i}\}_{i\in\{1,2\}}$ are chosen independently and uniformly at random. Accordingly, in the key generation we can set

$$\mathsf{pk}_1 = U_{n+1,1}^{-\mathsf{sk}_1} \qquad \text{and} \qquad \mathsf{pk}_2 = U_{n+1,1}^{-\mathsf{sk}_2}$$

and we aggregate the keys as

$$\{\widehat{U}_w = U_{w,1} \cdot U_{w,2}\}_{w\in[n+1]} \quad \text{and} \quad \widehat{U}_{n+2} = \mathsf{pk}_1 \cdot \mathsf{pk}_2 \cdot \prod_{w=1}^{n} U_{w,1}^{-x_{w,1}} \prod_{w=1}^{n} U_{w,2}^{-x_{w,2}}$$

where $\mathbf{x}_1$ and $\mathbf{x}_2$ are the chosen keys. One can encrypt using the new $\widehat{U}$ values instead of U, however, once we try to decrypt and expand the corresponding equations, we realize that many terms will not cancel out as before. For example, if a message is encrypted for slot 1, during decryption we will have,

$$\begin{aligned}
\prod_{w\in[n]} C_{3,w}^{x_{w,1}} &= \prod_{w\in[n]} h^{(y_w\cdot r+s)\cdot x_{w,1}} \cdot \prod_{w=1}^{n} U_{w,1}^{-z\cdot x_{w,1}} \cdot \prod_{w=1}^{n} U_{w,2}^{-z\cdot x_{w,1}} \\
C_{3,n+1}^{\mathsf{sk}_1} &= h^{s\cdot\mathsf{sk}_1} \cdot U_{n+1,1}^{-z\cdot\mathsf{sk}_1} \cdot U_{n+1,2}^{-z\cdot\mathsf{sk}_1} \\
C_{3,n+2} &= h^{s} \cdot U_{n+1,1}^{z\cdot\mathsf{sk}_1} \cdot U_{n+1,2}^{z\cdot\mathsf{sk}_2} \cdot \prod_{w=1}^{n} U_{w,1}^{z\cdot x_{w,1}} \prod_{w=1}^{n} U_{w,2}^{z\cdot x_{w,2}}
\end{aligned}$$

where the terms in blue can be canceled out using a similar multiplication trick as before. However, the terms $U_{n+1,2}^{-z\cdot\mathsf{sk}_1}, U_{n+1,2}^{z\cdot\mathsf{sk}_2}, \prod_{w\in[n]} U_{w,2}^{-z\cdot x_{w,1}}$ and $\prod_{w=1}^{n} U_{w,2}^{z\cdot x_{w,2}}$ *cannot* be canceled as they do not appear anywhere else, and further we assume the decryptor only knows sk_1, but not sk_2. We can circumvent this issue by introducing some "cross-terms" into the CRS, and use them in the aggregation to compute helper secret keys that enables the decryptor (holding sk_1 and $\mathbf{x}_1$) to cancel such terms. We create these terms such that they include both slot-specific and key-specific parts. Intuitively, they bind each slot to other slots and keys together. For slots $i, j \in [2]$ where $i \neq j$ and key indices $w \in [n+1]$, we define these terms as:

$$W_{i,j,w} = A_i^{u_{j,w}}.$$

We add $\{W_{i,j,w}\}_{i\neq j\in[2],w\in[n+1]}$ to the CRS as:

$$\mathsf{crs} = \left(\mathcal{G}, Z, h, \{A_i, B_i\}_{i\in[2]}, \left\{\{U_{w,i}\}, \{W_{i,j,w}\}_{i\neq j}\right\}_{i,j\in[2],w\in[n+1]}\right).$$

In addition, we will let the user publish $\{W_{j,i,n+1}^{\mathsf{sk}_i}\}_{i\in\{1,2\},j\neq i}$ in their respective public keys, to enable the other users to cancel out the desired cross terms, and

publish in the ciphertext an additional element $C_4 = g_1^z$, to be paired with the W's in order to compute the correct terms.

The above scheme is correct but unfortunately *insecure*. At a high level, the problem is that the adversary can pair C_4 with wrong elements and generate unintended relations between z and other components, in the exponent. To prevent this, instead of putting g_1^z directly in the ciphertext, we introduce an extra component $\Gamma = g_1^\gamma, \gamma \leftarrow\!\!\$\, \mathbb{Z}_q$ in the CRS, and set $C_4 = \Gamma^z$. The only other modification that we must apply is the generation of the CRS itself, where for slots $i, j \in \{1, 2\}$ with $i \neq j$, and key indices $w \in [n + 1]$, we define:

$$W_{i,j,w} = A_i^{u_{j,w}/\gamma}.$$

This forces a (possibly malicious) decryptor to pair C_4 *only* with the elements $W_{i,j,w}$ and remove the additional cross-terms described above. The rest of the construction remains the same. See Sect. 6 for more details.

Proof Sketch. We prove the above slotted RIPE scheme secure in the generic bilinear group model (GGM). Recall that in the GGM, the adversary is supplied with handles to the corresponding group elements from the scheme. Further, it can also learn handles to arbitrary linear combinations of existing and new elements (in the same group $\mathbb{G}_\mathfrak{t}, \mathfrak{t} \in \{1, 2, \mathsf{T}\}$) via the group oracles it is provided with. Additionally, since we are in the bilinear setting, the adversary also gets access to the pairing oracle that allows it to learn handles referring to the product of any two terms from the source groups $\mathbb{G}_1$ and $\mathbb{G}_2$. However, the only crucial information it can actually learn in this whole interaction is via the zero-tests that work again only in $\mathbb{G}_\mathsf{T}$.

Our formal multi-slot RIPE scheme in Sect. 6 introduces several variables with different combinations of indices. To argue indistinguishability in a convenient way between subsequent hybrids in the proof, we first switch from the GGM to the symbolic group model (SGM) via the Schwarz-Zippel lemma. In particular, the SGM allows us to represent all the terms, that the adversary can learn in the security game, as multivariate polynomials (in respective groups) from a ring of variables. The heart of the proof relies on arguing properties of the *coefficients* of these polynomials that correspond to *successful* zero-tests, which aids in proving indistinguishability directly. In particular, these claims set in while proving attribute hiding by switching the challenge attribute from $\mathbf{y}_0$ to $\mathbf{y}_1$ in the ciphertext elements $C_{3,w}$ $\forall w \in [n + 2]$, and helps in arguing the following:

1. Coefficients of such polynomials formed by pairing terms $C_{3,w} \in \mathbb{G}_1$ with *any* element in $\mathbb{G}_2$, except $A_i, i \in [2]$, must be *all zero*.
2. Such a coefficient vector must be *orthogonal* to $\mathbf{y}_b$ for $b \in \{0, 1\}$, and in particular, either be a *constant multiple* of the vector $\widetilde{\mathbf{x}}_i = (\mathbf{x}_i, \mathsf{sk}_i), i \in [2]$ or be *all zero*.

The claim in Item 1 follows from observing that the monomials formed symbolically (in the exponent) when pairing $C_{3,w}$ with *anything* in $\mathbb{G}_2$ (except A_1

or A_2) are all linearly independent and do not cancel out. Item 2 follows from two observations. The first one is that the randomness r (appearing as an independent symbolic term, but only in the components $C_{3,w}$'s) can only cancel out in zero-tests when the coefficients are orthogonal to $\mathbf{y}_b$. The second one follows additionally from linear independence of some specific symbolic terms and observing further that the vector of first $n+1$ coefficients can be expressed as a constant multiple of $\widetilde{\mathbf{x}}_i$. Overall, these claims ensure that the only non-trivial adversarial queries can be for vectors lying in the span of both *registered and valid* predicates. The rest of the proof follows from the admissibility of the adversary, and by reusing these claims. We refer to Theorem 6 for more details.

Comparison with the Slotted RABE of [41]**.** Our slotted RIPE scheme from prime-order pairings (in Sect. 6) shares some similarities at a high level with the slotted RABE from composite-order pairings by Hohenberger *et al.*[41]. For instance, the message-embedding mechanism in both schemes are same, which is by masking the message with the randomness in the term $e\left(h^s, A_i\right)$. (This is also a standard technique in many other pairing-based schemes.) The use of "slot"-based framework to embed users' keys is also similar, but only at the level of a blueprint. In particular, that is where the similarity ends. More specifically, the way slots and attributes are "glued" together in our scheme is fundamentally different: in [41], the ciphertext has two specific components, an attribute-specific component and a slot-specific one, where one party can decrypt a message if it manages to succeed to decrypt the slot-specific component and the attribute-specific component simultaneously. But in our scheme, the slot and attribute elements are entwined in the same ciphertext component. In essence, we conceptually treat the secret key as "one more dimension" in the predicate vector, whereas the scheme in [41] uses a separate machinery that takes care of the key component. Further, unlike [41] which reveals the policy in the ciphertext, we carefully ensure attribute hiding by multiplying a randomizer $r \in \mathbb{Z}_q^+$ to the attribute $\mathbf{y}$. As a result, we achieve totally different functionalities and stronger security notions. Finally, our scheme supports vectors from $\mathbb{Z}_q^{n^+}$ where q is a λ-bit prime and n denotes supported the vector length. As stated in [41, Section 7.2], this enables our scheme to support a large attribute universe in contrast to the pairing-based RABE in [41], that only supports a small attribute universe.

2.2 (Unbounded Users) Slotted RFE from iO

As a feasibility result, we show (slotted) RFE for all circuits based on indistinguishability obfuscation (iO) [13] and (succinct) somewhere statistically binding hash functions (SSB) [42,50]. In particular, we generalize the techniques from Hohenberger *et al.* [41] to get a slotted RFE from iO (which can be lifted to RFE with the powers-of-two trick). Below is a brief overview of this slotted RFE.

The CRS is set as the SSB hash key hk, and users' keys are generated through a PRG PRG and a seed s (i.e., $(\mathsf{pk}, \mathsf{sk}) = (\mathsf{PRG}(s), s)$). To aggregate $((\mathsf{pk}_i, f_i))_{i\in[L]}$, the KC computes a Merkle tree hash $h = \mathsf{Hash}(\mathsf{hk}, ((\mathsf{pk}_i, f_i))_{i\in[L]})$ and sets $\mathsf{mpk} = (\mathsf{hk}, h)$. The helper secret key hsk_i (of the i-th slot) is essen-

tially the SSB opening π_i for the i-th (hashed) block (pk_i, f_i). A ciphertext c (encrypting m) is simply the obfuscation $\widetilde{C}$ of a circuit $C_{h,m}$ that, on input $(i, \mathsf{pk}_i, f_i, \pi_i, \mathsf{sk}_i)$, returns $f_i(m)$ if the following two conditions are satisfied: π_i is a *valid opening* for the i-th block (pk_i, f_i) *and* $(\mathsf{pk}_i, \mathsf{sk}_i)$ is a *valid* key-pair. Decryption works using sk_i and $\mathsf{hsk}_i = \pi_i$ to evaluate $\widetilde{C}$ on input $(i, \mathsf{pk}_i, f_i, \pi_i, \mathsf{sk}_i)$. The scheme supports the function class $\mathsf{P/poly}$. Compactness of parameters is evident from SSB succinctness. Due to a poly-logarithmic overhead from the powers-of-two trick, the final RFE can support an arbitrary number of users by setting $L = 2^\lambda$. The registration runtime remains linear in the *current/effective* number of registered users at the time of registration. We provide more details in our full version [25].

2.3 On Function Privacy in (Slotted) RFE

By definition, RFE allows users to sample their own keys and functions. Thus, the notion of function-privacy, that is typically considered in the setting of (secret-key) FE [21,55], does not make much sense from this perspective. However, one can still define function-privacy w.r.t. any other registered or unregistered party. In more detail, in the case of RFE, a user choosing its own keys and functions may want to hide its function from any party including the KC. Capturing this requires a mild change in the RFE syntax, where the function can be input to the KGen algorithm instead of RegPK and also require that the generated user key-pair is tied to this function. The KC gets access of only the users' public

Table 1. Comparing known registered encryption schemes in terms of efficiency and assumptions. We only consider worst-case time complexity. For schemes supporting an unbounded (resp. bounded) number of users, L denotes the *current* number of registered (resp. the maximum number of supported) users. We omit λ to simplify the table, e.g. for $k \in \mathbb{N}$, $O(k)$ and $\mathsf{poly}(\log k)$ respectively denote $k \cdot \mathsf{poly}(\lambda)$ and $\mathsf{poly}(\lambda, \log k)$ etc. $\mathcal{U}$ (from [41]) denotes the attribute space supported by the corresponding scheme. $\mathcal{F}$ denotes the function space supported by our schemes (each function $f \in \mathcal{F}$ of our RIPE is an n-length vector from $\mathbb{Z}_q^{n^+}$). Above, **BB** is an abbreviation for "black-box".

Reference	Type	CRS size	Keygen runtime	Registration key runtime	Master public key size	Helper dec. key size	# Updates	Unbounded users	BB	Assumptions
[29]	IBE	$O(1)$	$O(1)$	$\mathsf{poly}(\log L)$	$\mathsf{poly}(\log L)$	$\mathsf{poly}(\log L)$	$O(\log L)$	✓	✗	iO + SSB
[29]	IBE	$O(1)$	$O(1)$	$O(L)$	$\mathsf{poly}(\log L)$	$\mathsf{poly}(\log L)$	$O(\log L)$	✓	✗	CDH/LWE
[30]	Anon. IBE	$O(1)$	$O(1)$	$\mathsf{poly}(\log L)$	$\mathsf{poly}(\log L)$	$\mathsf{poly}(\log L)$	$O(\log L)$	✓	✗	CDH/LWE
[39]	IBE	$O(1)$	$O(1)$	$\mathsf{poly}(\log L)$	$\mathsf{poly}(\log L)$	$\mathsf{poly}(\log L)$	$O(\log L)$	✓	✗	CDH/LWE
[23]	IBE	$O(1)$	$O(1)$	$\mathsf{poly}(\log L)$	$O(\sqrt{L})$	$\mathsf{poly}(\log L)$	$O(\log L)$	✓	✗	CDH/LWE
[35]	IBE $O(1)$-size ciphertexts	$O(\sqrt{L})$	$O(\sqrt{L})$	$O(\sqrt{L})$	$O(\sqrt{L})$	$O(\sqrt{L})$	$O(\sqrt{L})$	✗	✓	Pairings of Prime Order
[35]	IBE $O(\log L)$-size ciphertexts	$O(\sqrt{L})$	$O(\sqrt{L})$	$O(\sqrt{L}\log L)$	$O(\sqrt{L}\log L)$	$O(\log L)$	$O(\log L)$	✗	✓	Pairings of Prime Order
[24]	IBE	$\mathsf{poly}(\log L)$	$\mathsf{poly}(\log L)$	$O(L)$	$\mathsf{poly}(\log L)$	$\mathsf{poly}(\log L)$	$O(\log L)$	✓	✓	LWE
[41]	ABE small attribute space $\mathcal{U}$ LSSS policies	$L^2 \cdot \mathsf{poly}(\lvert\mathcal{U}\rvert, \log L)$	$L \cdot \mathsf{poly}(\lvert\mathcal{U}\rvert, \log L)$	$L \cdot \mathsf{poly}(\lvert\mathcal{U}\rvert, \log L)$	$\lvert\mathcal{U}\rvert \cdot \mathsf{poly}(\log L)$	$\lvert\mathcal{U}\rvert \cdot \mathsf{poly}(\log L)$	$O(\log L)$	✗	✓	Pairings of Composite Order
[41]	ABE large attribute space $\mathcal{U}$ arbitrary policies	$O(1)$	$O(1)$	$O(L)$	$O(1)$	$O(1)$	$O(\log L)$	✓	✗	iO + SSB
Ours §6	Inner-Product PE large function space $\mathcal{F}$ n-size vectors	$n \cdot L^2 \cdot \mathsf{poly}(\log L)$	$L \cdot \mathsf{poly}(\log L)$	$n \cdot L^2 \cdot \mathsf{poly}(\log L)$	$n \cdot \mathsf{poly}(\log L)$	$n \cdot \mathsf{poly}(\log L)$	$O(\log L)$	✗	✓	Pairings of Prime Order + GGM
Ours [25]	FE large function space $\mathcal{F}$ arbitrary functions	$O(1)$	$O(1)$	$O(L)$	$O(1)$	$O(1)$	$O(\log L)$	✓	✗	iO + SSB

keys to aggregate and generate $\mathsf{mpk}, \mathsf{hsk}$.[6] The security definition would need to change accordingly. In particular, it would now additionally require each public key to computationally hide the function tied to it.

All our schemes can be modified to satisfy this syntax. For example, our slotted RIPE from pairings can be easily adapted to this notion since the *extended* key $\widetilde{\mathbf{x}}_i = (\mathbf{x}_i, \mathsf{sk}_i, 1)$ is embedded in the public-key pk_i for slot $i \in [2]$ as $\mathsf{pk}_i = \prod_{w=1}^{n+1} U_{w,i}^{-\widetilde{x}_{w,i}}$. This holds similarly for the cross-terms as well. Using a NIZK, the users can prove that they always choose a non-zero vector as its predicate. It is also easy to verify the same for our slotted RFE from iO. However, for simplicity, we avoid formalizing this in our definitions and schemes. Both our formal constructions from Sect. 6 and the one based on iO are thus in the standard registered setting (i.e., without function-privacy). Building more efficient function-private RFE for specific functions is left as a future work.

3 Related Work

The first paper [29] defined and built RIBE from iO and SSB hashes; this was later improved by Garg *et al.* [30] building RIBE (with the same level of efficiency) from standard assumptions (e.g., from CDH/LWE) even for *anonymous* IBE. Subsequent work on RIBE focused on adding verifiability [39], proving lower bounds on the number of decryption updates [49], improving on practical efficiency of the garbled circuit construction [23], providing effcient black-box construction from pairings with $O(\sqrt{L})$ mpk [35]. More recently, Döttling *et al.* [24] obtain a lattice-based RIBE with the sizes of $\mathsf{crs}, \mathsf{mpk}, \mathsf{hsk}$ as well as key generation runtime growing as $\mathsf{poly}(\log L)$, with a $O(L)$ registration runtime and $O(\log L)$ number of updates. Very recently, [41] extended RIBE to the setting of ABE. They built a (black-box) registered ABE (RABE) scheme supporting a *bounded* number of users and linear secret sharing schemes as access policies from assumptions on composite-order pairing groups. However, their (pairing-based) scheme, the size of CRS and runtime of aggregate and keygen depend linearly on the size of attribute space $|\mathcal{U}|$. The dependence on $|\mathcal{U}|$ allows their scheme to only support a small attribute space (e.g., $|\mathcal{U}| \in \mathsf{poly}(n)$). Notably, our (paring-based) RIPE does not suffer from this limitation since our parameters depend only on the vector length $n = n(\lambda)$ (see Table 1); so we can support a exponential size function class $\mathcal{F}$.

In [39], the authors further introduced an RABE extension to more general access structures. Specifically, they proposed a universal definition of registration-based encryption in which the algorithms take as an additional input the description of an FE scheme (although no construction was presented). Such algorithms compile the standard algorithmic behavior of the FE scheme into

[6] In such a setting (rogue) users can try to register arbitrary functions of their choice which would allow them to learn arbitrary information about encrypted messages. To prevent this, one can restrict the function class at setup meaningfully (e.g., excluding trivial functions like identity). Any user wanting to register its public key would then need to prove the validity of its chosen function w.r.t. this class of functions.

a (verifiable) registration-based one. However, our tailored notion for the functional encryption setting is more natural and follows directly from the RABE definition.

Finally, we also mention a related work on dynamic decentralized FE [22] (DDFE), where there is no trusted authority and users sample their own keys. DDFE, as a notion, posits other general (and albeit unrelated) requirements like (conditional) aggregation of labelled data which comes from different users using seperate FE instances. However, a crucial difference from the registered setting, is that in DDFE there is no requirement on the master public key size, which can be as large as the number of registered users. This is a major challenge (and arguably the defining feature) of all registered settings. Chotard *et al.*[22] also built IP-DDFE, that outputs the inner-product value $\langle \mathbf{x}, \mathbf{y} \rangle$, while our scheme is for the more challenging orthogonality-test predicate (with two-sided security).

Open Problems. We view our work as an initial first step in the world of registered FE, however many open problems remain. For example, a natural question is if registered FE can be obtained generically from any compact, polynomially-hard FE. Another interesting direction is to design schemes for specialized function classes from weaker assumptions. Finally, a technical open problem is to prove our pairing-based RIPE scheme (or some modification thereof) secure in the standard model.

4 Organization

We organize the rest of the paper as follows. The formal definitions of both RFE and slotted RFE extend the same for the RABE setting from [41] in a straighforward way. Hence, we provide the RFE definitions in our full version [25]. Our main focus in this paper is on building (slotted) registered IPE. Thus, we first define slotted RIPE formally in Sect. 5.1 and extend it to slotted RFE for the case of general functions in our full version [25]. Our slotted RIPE scheme from bilinear pairings is provided in Sect. 6. We demonstrate our implementation results of the above slotted RIPE scheme in Sect. 7. Our slotted RFE for general functions and unbounded users, built on iO (plus an SSB hash and a PRG), generalizes a construction from [41] and is presented in [25]. Further, the transformation from slotted RFE to RFE extending the generic compiler from [41] is again provided in our full version [25].

5 Preliminaries

Notations. We write $[n] = \{1, 2, \ldots, n\}$ and $[0, n] = \{0\} \cup [n]$. Capital bold-face letters (such as $\mathbf{X}$) are used to denote random variables, small bold-face letters (such as $\mathbf{x}$) to denote vectors, small letters (such as x) to denote concrete values, calligraphic letters (such as $\mathcal{X}$) to denote sets, serif letters (such as A) to denote algorithms. All of our algorithms are modeled as (possibly interactive) Turing

machines. For a string $x \in \{0,1\}^*$, we let $|x|$ be its length; if $\mathcal{X}$ is a set or a list, $|\mathcal{X}|$ represents the cardinality of $\mathcal{X}$. When x is chosen uniformly in $\mathcal{X}$, we write $x \leftarrow\!\!\$\ \mathcal{X}$. If A is an algorithm, we write $y \leftarrow\!\!\$\ \mathsf{A}(x)$ to denote a run of A on input x and output y; if A is randomized, y is a random variable and $\mathsf{A}(x;r)$ denotes a run of A on input x and (uniform) randomness r. An algorithm A is *probabilistic polynomial-time* (PPT) if A is randomized and for any input $x, r \in \{0,1\}^*$ the computation of $\mathsf{A}(x;r)$ terminates in a polynomial number of steps (in the input size). We write $C(x) = y$ to denote the evaluation of the circuit C on input x and output y. For any integer $k \in \mathbb{N}$, we denote $\mathbb{Z}_q^{k^+} = \mathbb{Z}_q^k \setminus \{\mathbf{0}^k\}$ as the set of all non-zero k-size vectors over $\mathbb{Z}_q$, and $\mathbb{Z}_q^+ = \mathbb{Z}_q \setminus \{0\}$.

Negligible Functions. Throughout the paper, we denote the security parameter by $\lambda \in \mathbb{N}$ and we implicitly assume that every algorithm takes λ as input. A function $\nu(\lambda)$ is called negligible in $\lambda \in \mathbb{N}$ if it vanishes faster than the inverse of any polynomial in λ, i.e. $\nu(\lambda) \in O(1/p(\lambda))$ for all positive polynomials $p(\lambda)$.

5.1 Slotted Registered Inner-Product Encryption

We now present the slotted RIPE definitions below. Let $n = n(\lambda)$ be a polynomial in λ and q be a prime. A slotted RIPE with message space $\mathcal{M}$ and attribute space $\mathcal{U}$ is composed of the following polynomial-time algorithms:

$\mathsf{Setup}(1^\lambda, 1^n, 1^L)$: On input the security parameter 1^n, the vector length n, and the number of slots L, the randomized setup algorithm outputs a common reference string crs.

$\mathsf{KGen}(\mathsf{crs}, i)$: On input the common reference string crs and a slot index $i \in [L]$, the randomized key-generation algorithm outputs a public key pk_i and a secret key sk_i.

$\mathsf{IsValid}(\mathsf{crs}, i, \mathsf{pk}_i)$: On input the common reference string crs, a slot index $i \in [L]$, and a public key pk_i, the deterministic key validation algorithm outputs a decision bit $b \in \{0,1\}$.

$\mathsf{Aggr}(\mathsf{crs}, ((\mathsf{pk}_i, \mathbf{x}_i))_{i\in[L]})$: On input the common reference string crs and a L pairs $(\mathsf{pk}_1, \mathbf{x}_1), \ldots, (\mathsf{pk}_L, \mathbf{x}_L)$ each composed of a public key pk_i and its corresponding (non-zero) vector $\mathbf{x}_i \in \mathcal{U}$, the deterministic aggregation algorithm outputs the master public key mpk and a L helper decryption keys $\mathsf{hsk}_1, \ldots, \mathsf{hsk}_L$.

$\mathsf{Enc}(\mathsf{mpk}, \mathbf{y}, m)$: On input the master public key mpk, a (non-zero) attribute vector $\mathbf{y} \in \mathcal{U}$, and a message $m \in \mathcal{M}$, the randomized encryption algorithm outputs a ciphertext c.

$\mathsf{Dec}(\mathsf{sk}, \mathsf{hsk}, c)$: On input a secret key sk, an helper decryption key hsk, and a ciphertext c, the deterministic decryption algorithm outputs a message $m \in \mathcal{M} \cup \{\perp\}$.

Completeness, Correctness, and Efficiency. Completeness of slotted RIPE says that honestly generated public keys for a slot index $i \in [L]$ are valid with

respect to the same slot i, i.e., $\mathsf{IsValid}(\mathsf{crs}, i, \mathsf{pk}_i) = 1$. Similarly, correctness says that honest ciphertexts correctly decrypt (to functions of the plaintext) under honestly generated and aggregated keys. For compactness and efficiency, we extend the requirements of RFE to the slotted RIPE setting. The formal definitions are provided in our full version [25]. Below we define the security of slotted RIPE formally.

Definition 1 (Security of slotted RIPE). *Let* $\Pi_{\mathsf{sRIPE}} = (\mathsf{Setup}, \mathsf{KGen}, \mathsf{IsValid}, \mathsf{Aggr}, \mathsf{Enc}, \mathsf{Dec})$ *be a slotted RIPE scheme with message space* $\mathcal{M}$ *and attribute space* $\mathcal{U}$*. For any adversary* A*, define the following security game* $\mathbf{Game}^{\mathsf{sRIPE}}_{\Pi_{\mathsf{sRIPE}},\mathsf{A}}(\lambda, b)$ *with respect to a bit* $b \in \{0,1\}$ *between* A *and a challenger.*

- **Setup phase:** *Upon getting an attribute length* n *and a slot count* L *from the adversary* A*, the challenger samples* $\mathsf{crs} \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, 1^n, 1^L)$ *and gives* crs *to* A*. The challenger also initializes a counter* $\mathsf{ctr} = 0$*, a dictionary* D*, and a set of slot indices* $\mathcal{C}_L = \emptyset$ *to account for corrupted slots.*
- **Pre-challenge query phase:** A *can issue the following queries.*
  - **Key-generation query:** A *specifies a slot index* $i \in [L]$*. As a response, the challenger increments* $\mathsf{ctr} = \mathsf{ctr} + 1$*, samples* $(\mathsf{pk}_{\mathsf{ctr}}, \mathsf{sk}_{\mathsf{ctr}}) \leftarrow\!\!\$\ \mathsf{KGen}(\mathsf{crs}, i)$*, updates the dictionary as* $\mathsf{D}[\mathsf{ctr}] = (i, \mathsf{pk}_{\mathsf{ctr}}, \mathsf{sk}_{\mathsf{ctr}})$ *and replies with* $(\mathsf{ctr}, \mathsf{pk}_{\mathsf{ctr}})$ *to* A*.*
  - **Corruption query:** A *specifies an index* $c \in [\mathsf{ctr}]$*. In response, the challenger looks up the tuple* $\mathsf{D}[c] = (i', \mathsf{pk}', \mathsf{sk}')$ *and replies with* sk' *to* A*.*
- **Challenge phase:** *For each* $i \in [L]$*,* A *specifies a tuple* $(c_i, \mathbf{x}_i, \mathsf{pk}^*_i)$ *where:*
  - *either* $c_i \in [\mathsf{ctr}]$ *that refers to a challenger-generated key from before which it associates with a non-zero predicate* $\mathbf{x}_i \in \mathcal{U}$*: in this case, the challenger looks up* $\mathsf{D}[c_i] = (i', \mathsf{pk}', \mathsf{sk}')$ *and halts if* $i \neq i'$*. Else, the challenger sets* $\mathsf{pk}^*_i = \mathsf{pk}'$*. Further, if* A *issued a corrupt query before on* c_i*, the challenger adds* i *to* $\mathcal{C}_L$*.*
  - *or* $c_i = \perp$ *that refers to a self-generated (and corrupt) secret key for an arbitrary non-zero predicate* $\mathbf{x}_i \in \mathcal{U}$*: in this case, the challenger aborts if* $\mathsf{IsValid}(\mathsf{crs}, i, \mathsf{pk}^*_i) = 0$*. Else if* pk^*_i *is valid, it adds the index* i *to* $\mathcal{C}_L$*.*

  Additionally, A *sends a challenge pair* $(\mathbf{y}_0, m_0), (\mathbf{y}_1, m_1) \in \mathcal{U} \times \mathcal{M}$*. In response, the challenger computes* $\left(\mathsf{mpk}, (\mathsf{hsk}_i)_{i\in[L]}\right) = \mathsf{Aggr}\left(\mathsf{crs}, (\mathsf{pk}^*_i, \mathbf{x}_i)_{i\in[L]}\right)$ *and* $c^* \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{mpk}, \mathbf{y}_b, m_b)$*, and replies with* c^* *to* A*.*
- **Output phase:** A *returns a bit* $b' \in \{0,1\}$ *which is also the output of the experiment.*

A *is called admissible if the challenge pair* $(\mathbf{y}_0, m_0), (\mathbf{y}_1, m_1)$ *satisfy the following:*

- $\forall \mathbf{x}_i \in \mathcal{U}$ *with* $i \in \mathcal{C}_L$*, it holds that:*

$$\textit{either } \langle \mathbf{x}_i, \mathbf{y}_0 \rangle = \langle \mathbf{x}_i, \mathbf{y}_1 \rangle = 0 \quad \textit{or} \quad \textit{both} \langle \mathbf{x}_i, \mathbf{y}_0 \rangle, \langle \mathbf{x}_i, \mathbf{y}_1 \rangle \neq 0, \textit{ and}$$

- *if* $\exists \mathbf{x}_i \in \mathcal{U}$ *with* $i \in \mathcal{C}_L$ *such that* $\langle \mathbf{x}_i, \mathbf{y}_0 \rangle = \langle \mathbf{x}_i, \mathbf{y}_1 \rangle = 0$*, then* $m_0 = m_1$*.*

We say that Π_{sRIPE} *is secure if for all polynomials* $n = n(\lambda), L = L(\lambda)$ *and for all PPT and admissible* A *in the above security hybrid, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for all* $\lambda \in \mathbb{N}$,

$$\left|\Pr[\mathbf{Game}^{\mathsf{sRIPE}}_{\Pi_{\mathsf{sRIPE}},\mathsf{A}}(\lambda,0)=1] - \Pr[\mathbf{Game}^{\mathsf{sRIPE}}_{\Pi_{\mathsf{sRIPE}},\mathsf{A}}(\lambda,1)=1]\right| = \mathsf{negl}(\lambda).$$

Remark 1. We argue in our full version [25] that for general RFE, security without post-challenge queries imply security with post-challenge queries in the slotted setting as well. This is because Aggr is deterministic and does not require any secret. Hence, an adversary can simulate the post-challenge queries itself.

6 Slotted Registered IPE from Prime-Order Pairings

Bilinear Groups. Our slotted RIPE is based on asymmetric bilinear groups. We use cyclic groups of prime order q with an asymmetric bilinear map endowed on them. We assume a PPT algorithm $\mathsf{GroupGen}$ that takes a security parameter λ as input and outputs $\mathcal{G} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathsf{T}, q, g_1, g_2, e)$, where $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathsf{T}$ are cyclic groups of prime order q, g_1 (resp. g_2) is random generator in $\mathbb{G}_1$ (resp. $\mathbb{G}_2$) and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_\mathsf{T}$ is a non-degenerate bilinear map.

We assume the message space $\mathcal{M} = \mathbb{G}_\mathsf{T}$ for our scheme. Our slotted RIPE supports an a-priori fixed number of slots $L = L(\lambda)$, i.e., the scheme supports a bounded number of slots. Below, we describe our formal scheme.

Construction 1. *The slotted RIPE scheme* $\Pi_{\mathsf{sRIPE}} = (\mathsf{Setup}, \mathsf{KGen}, \mathsf{IsValid}, \mathsf{Aggr}, \mathsf{Enc}, \mathsf{Dec})$ *with message space* $\mathcal{M} = \mathbb{G}_\mathsf{T}$ *and attribute space* $\mathcal{U} = \mathbb{Z}_q^{n^+}$ *is as follows:*

$\mathsf{Setup}(1^\lambda, 1^n, 1^L)$*: On input the security parameter* λ*, the attribute size* n *and the number of slots* L*, compute* $\mathcal{G} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathsf{T}, q, g_1, g_2, e) \leftarrow\!\!\$\ \mathsf{GroupGen}(1^\lambda)$ *and generate the common reference string as follows.*

1. *Sample* $\alpha, \beta, \gamma \leftarrow\!\!\$\ \mathbb{Z}_q^+$ *and set* $h = g_1^\beta, Z = e(g_1, g_2)^\alpha, \Gamma = g_1^\gamma, n' = n + 1$.
2. *For each index* $i \in [0, L]$*, do the following:*
   1. *for each* $w \in [n']$*, sample* $u_{w,i} \leftarrow\!\!\$\ \mathbb{Z}_q$ *and set* $U_{w,i} = g_1^{u_{w,i}}$.
   2. *for a slot index* $i > 0$*, sample* $t_i \leftarrow\!\!\$\ \mathbb{Z}_q$ *and set* $A_i = g_2^{t_i}, B_i = g_2^\alpha \cdot A_i^\beta$.
   3. *for a slot index* $i > 0$*,* $\forall w \in [n'], j \in [0, L] \setminus \{i\}$*, set* $W_{i,j,w} = A_i^{u_{w,j}/\gamma}$.
3. *Sample* $\widetilde{\mathbf{x}}_0 = (\widetilde{x}_{1,0}, \ldots, \widetilde{x}_{n,0}, \widetilde{r}_0) \leftarrow\!\!\$\ \mathbb{Z}_q^{n'^+}$*. Set* $\mathsf{sk}_0 = \widetilde{\mathbf{x}}_0$ *and*

$$T_0 = \left(\prod_{w=1}^{n} U_{w,0}^{-\widetilde{x}_{w,0}}\right) \cdot U_{n',0}^{-\widetilde{r}_0} \quad , \quad \widetilde{W}_{i,0} = \left(\prod_{w=1}^{n} W_{i,0,w}^{\widetilde{x}_{w,0}}\right) \cdot W_{i,0,n'}^{\widetilde{r}_0}, \quad \forall i \in [L].$$

Also, set $\mathsf{pk}_0 = \left(T_0, \left\{\widetilde{W}_{i,0}\right\}_{i\in[L]}\right)$.

Finally, output the common reference string

$$\mathsf{crs} = \left(\mathcal{G}, Z, h, \Gamma, \{A_i, B_i\}_{i\in[L]}, \{\{U_{w,i}\}_{i\in[0,L]}, \{W_{i,j,w}\}_{i\in[L],j\in[0,L]\setminus\{i\}}\}_{w\in[n']}, \mathsf{pk}_0\right)$$

$\mathsf{KGen}(\mathsf{crs}, i)$: *On input the common reference string* crs *and a slot index* $i \in [L]$, *do the following.*

1. *Parse the common reference string*

$$\mathsf{crs} = \Big(\mathcal{G}, Z, h, \Gamma, \{A_i, B_i\}_{i\in[L]}\,, \Big\{\{U_{w,i}\}_{i\in[0,L]}\,, \{W_{i,j,w}\}_{i\in[L],j\in[0,L]\setminus\{i\}}\Big\}_{w\in[n']}\,, \mathsf{pk}_0\Big)\,.$$

2. *Sample* $\widetilde{r}_i \leftarrow\!\!\$\, \mathbb{Z}_q^+$ *and pick elements* $U_{n',i}$ *and* $\{W_{j,i,n'}\}_{j\in[L]\setminus\{i\}}$ *from* crs.
3. *Compute* $T_i = U_{n',i}^{-\widetilde{r}_i}$ *and* $\widetilde{W}_{j,i} = W_{j,i,n'}^{\widetilde{r}_i}, \forall j \in [L] \setminus \{i\}$.
4. *Output* $\mathsf{pk}_i = \Big(T_i, \{\widetilde{W}_{j,i}\}_{j\in[L]\setminus\{i\}}\Big)$ *and* $\mathsf{sk}_i = \widetilde{r}_i$.

$\mathsf{IsValid}(\mathsf{crs}, i, \mathsf{pk}_i)$: *On input the common reference string* crs, *a slot index* $i \in [L]$ *and a purported public key* $\mathsf{pk}_i = \Big(T_i, \{\widetilde{W}_{j,i}\}_{j\in[L]\setminus\{i\}}\Big)$, *the key-validation algorithm first affirms that each of the components in* pk_i *is a valid group element, namely:* $\Big(T_i \stackrel{?}{\in} \mathbb{G}_1 \setminus \{1_{\mathbb{G}_1}\} \quad \wedge \quad \widetilde{W}_{j,i} \stackrel{?}{\in} G_2 \setminus \{1_{\mathbb{G}_2}\}, \quad \forall j \in [L] \setminus \{i\}\Big)$ *where* $1_{\mathbb{G}_\mathfrak{t}}$ *denotes the identity in* $\mathbb{G}_\mathfrak{t}$ *for* $\mathfrak{t} \in [2]$. *If the checks pass, it picks the elements* $U_{n',i}$ *and* $\{W_{j,i,n'}\}_{j\in[L]\setminus\{i\}}$ *from* crs *and checks further that*

$$e\left(T_i^{-1}, W_{j,i,n'}\right) \stackrel{?}{=} e\left(U_{n',i}, \widetilde{W}_{j,i}\right), \forall j \in [L] \setminus \{i\}.$$

If all checks pass, it outputs 1. *Else, it outputs* 0.

$\mathsf{Aggr}(\mathsf{crs}, ((\mathsf{pk}_i, \mathbf{x}_i))_{i\in[L]})$: *On input the common reference string* crs *and a set of* L *public keys* $\mathsf{pk}_i = \Big(T_i, \{\widetilde{W}_{j,i}\}_{j\in[L]\setminus\{i\}}\Big)$ *together with vectors* $\mathbf{x}_i = (x_{1,i}, \ldots, x_{n,i}) \in \mathbb{Z}_q^{n^+}$ *(representing predicates* $f_{\mathbf{x}_i}$*), compute the following.*

1. *Parse the common reference string*

$$\mathsf{crs} = \Big(\mathcal{G}, Z, h, \Gamma, \{A_i, B_i\}_{i\in[L]}\,, \Big\{\{U_{w,i}\}_{i\in[0,L]}\,, \{W_{i,j,w}\}_{i\in[L],j\in[0,L]\setminus\{i\}}\Big\}_{w\in[n']}\,, \mathsf{pk}_0\Big)\,.$$

2. *Fuse the predicate vector* $\mathbf{x}_i$ *into* pk_i *by updating each of its components as*

$$T_i = \left(\prod_{w=1}^{n} U_{w,i}^{-x_{w,i}}\right) \cdot T_i \quad , \quad \widetilde{W}_{j,i} = \left(\prod_{w=1}^{n} W_{j,i,w}^{x_{w,i}}\right) \cdot \widetilde{W}_{j,i}, \forall j \in [L] \setminus \{i\}$$

and set $\mathsf{pk}_i = \left(T_i, \left\{\widetilde{W}_{j,i}\right\}_{j\in[L]\setminus\{i\}}\right)$. *Further, parse* pk_0 *as follows:*

$$\mathsf{pk}_0 = \left(T_0, \left\{\widetilde{W}_{j,0}\right\}_{j\in[0,L]\setminus\{0\}}\right).$$

3. *For each* $w \in [n']$, *compute* $\widehat{U}_w = \prod_{i\in[0,L]} U_{w,i}$ *and* $\widehat{U}_{n'+1} = \prod_{i\in[0,L]} T_i$.

4. *Compute the cross-terms as follows. For each slot index* $i \in [L]$*:*
   (a) *for each* $w \in [n']$*, compute* $\widehat{W}_{w,i} = \prod_{j \in [0,L] \setminus \{i\}} W_{i,j,w}$.
   (b) *compute* $\widehat{W}_{n'+1,i} = \left(\prod_{j \in [0,L] \setminus \{i\}} \widetilde{W}_{i,j} \right)^{-1}$.
5. *Output the master public key and the slot-specific helper secret keys as* $\mathsf{mpk} = \left(\mathcal{G}, h, Z, \Gamma, \left\{ \widehat{U}_w \right\}_{w \in [n'+1]} \right)$, *and*

$$\mathsf{hsk}_i = \left(\mathcal{G}, i, \mathbf{x}_i, A_i, B_i, \left\{ \widehat{W}_{w,i} \right\}_{w \in [n'+1]} \right), \forall i \in [L].$$

$\mathsf{Enc}(\mathsf{mpk}, \mathbf{y}, m)$*: On input the master public key* mpk*, a vector* $\mathbf{y} = (y_1, \ldots, y_n) \in \mathbb{Z}_q^{n^+}$ *(as an attribute) and a message* $m \in \mathbb{G}_\mathsf{T}$*, the ciphertext is computed as:*

1. *Parse* $\mathsf{mpk} = \left(\mathcal{G}, h, Z, \Gamma, \left\{ \widehat{U}_w \right\}_{w \in [n'+1]} \right)$.
2. *Set* $\widetilde{\mathbf{y}} = (\mathbf{y}, 0, 0) \in \mathbb{Z}_q^{n'+1}$ *and sample* $s, r, z \leftarrow_\$ \mathbb{Z}_q^+$. *Also, parse* $\widetilde{\mathbf{y}} = (\widetilde{y}_1, \ldots, \widetilde{y}_{n'+1})$.
3. *Message embedding: set* $C_1 = m \cdot Z^s$ *and* $C_2 = g_1^s$.
4. *Attribute and Slot embedding: for each* $w \in [n'+1]$*, set* $C_{3,w} = h^{\widetilde{y}_w \cdot r + s} \cdot \widehat{U}_w^{-z}$. *Set* $C_4 = \Gamma^z$.
5. *Output the ciphertext* $c = \left(C_1, C_2, \{C_{3,w}\}_{w \in [n'+1]}, C_4 \right)$.

$\mathsf{Dec}(\mathsf{sk}, \mathsf{hsk}, c)$*: Parse the input secret key* sk*, helper secret key* hsk *and ciphertext* c *as* $\mathsf{sk} = \widetilde{r}_i$*, and*

$$\mathsf{hsk} = \left(\mathcal{G}, i, \mathbf{x}_i, A_i, B_i, \left\{ \widehat{W}_{w,i} \right\}_{w \in [n'+1]} \right), c = \left(C_1, C_2, \{C_{3,w}\}_{w \in [n'+1]}, C_4 \right),$$

for some $i \in [L]$*. Let* $\widetilde{\mathbf{x}}_i = (\widetilde{x}_{1,i}, \ldots, \widetilde{x}_{n'+1,i}) = (\mathbf{x}_i, \widetilde{r}_i, 1) \in \mathbb{Z}_q^{n'+1}, X_i = \sum_{w=1}^{n'+1} \widetilde{x}_{w,i} \in \mathbb{Z}_q$*. Compute and output the following:*

$$\frac{C_1}{e(C_2, B_i)} \cdot \left[\prod_{w=1}^{n'+1} \left\{ e\left(C_{3,w}^{\widetilde{x}_{w,i}}, A_i \right) \cdot e\left(C_4, \widehat{W}_{w,i}^{\widetilde{x}_{w,i}} \right) \right\} \right]^{X_i^{-1}}.$$

Remark: In the setup algorithm in our scheme, we introduce a *dummy* slot "0" and pre-register an *honestly* generated dummy key pk_0. This slot does not impact the security definition in any way because the associated secret key sk_0 is thrown away once the one-time setup is executed. This modification is done only for a simpler analysis of the security proof in the GGM.

Theorem 3 (Completeness of Construction 1). *The slotted RIPE scheme* Π_{sRIPE} *with message space* $\mathcal{M} = \mathbb{G}_\mathsf{T}$ *and attribute space* $\mathcal{U} = \mathbb{Z}_q^{n^+}$ *from Construction 1 is complete.*

Theorem 4 (Compactness and Efficiency of Construction 1). *The slotted RIPE scheme* Π_{sRIPE} *with message space* $\mathcal{M} = \mathbb{G}_\mathsf{T}$ *and attribute space* $\mathcal{U} = \mathbb{Z}_q^{n^+}$ *from Construction 1 satisfies the following properties:*

- $|\mathsf{crs}| = n \cdot L^2 \cdot \mathsf{poly}(\lambda), |\mathsf{mpk}| = n \cdot \mathsf{poly}(\lambda), |\mathsf{hsk}| = (n \cdot \mathsf{poly}(\lambda) + O(\log L))$
- $\mathbf{Runtime}(\mathsf{KGen}) = O(L) \cdot \mathsf{poly}(\lambda)$, $\mathbf{Runtime}(\mathsf{IsValid}) = L \cdot \mathsf{poly}(\lambda)$
- $\mathbf{Runtime}(\mathsf{Aggr}) = n \cdot L^2 \cdot \mathsf{poly}(\lambda)$

We refer to the full version [25] for the proofs of Theorems 4 and 3.

Theorem 5 (Perfect Correctness of Construction 1). *The slotted RIPE scheme Π_{sRIPE} with message space $\mathcal{M} = \mathbb{G}_{\mathsf{T}}$ and attribute space $\mathcal{U} = \mathbb{Z}_q^{n^+}$ from Construction 1 is perfectly correct.*

Proof. Fix some λ, attribute size $n = n(\lambda)$, a slot count $L = L(\lambda)$ and an index $i \in [L]$. Let $\mathsf{crs} \leftarrow\!\!\$\ \mathsf{Setup}(1^\lambda, 1^n, 1^L)$ and $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow\!\!\$\ \mathsf{KGen}(\mathsf{crs}, i)$ be defined as in the scheme from Construction 1. Take any set of public keys $\{\mathsf{pk}_j\}_{j \in [L] \setminus \{i\}}$, where $\mathsf{IsValid}(\mathsf{crs}, j, \mathsf{pk}_j) = 1$. Therefore, we have

$$\mathsf{pk}_j = \left(T_j, \left\{\widetilde{W}_{\ell,j}\right\}_{\ell \in [L] \setminus \{j\}}\right), \forall j \in [L] \setminus \{i\} \quad , \quad \mathsf{sk}_j = \widetilde{r}_j \text{ for some } \widetilde{r}_j \in \mathbb{Z}_q^+.$$

For each $j \in [L]$, let $\mathbf{x}_j \in \mathbb{Z}_q^{n^+}$ be the predicate vector associated to pk_j and let $\widetilde{\mathbf{x}}_j = (\mathbf{x}_j, \widetilde{r}_j, 1)$. Further, let mpk and hsk_i be as computed by $\mathsf{Aggr}(\mathsf{crs}, ((\mathsf{pk}_j, \mathbf{x}_j))_{j \in [L]})$. Now, note that in the Dec algorithm, the computation associated to the message components yield

$$\frac{C_1}{e(C_2, B_i)} = \frac{m \cdot Z^s}{e\left(g_1^s, g_2^\alpha \cdot A_i^\beta\right)} = \frac{m \cdot e\left(g_1, g_2\right)^{\alpha \cdot s}}{e\left(g_1, g_2\right)^{\alpha \cdot s} \cdot e\left(g_1, g_2\right)^{s\beta t_i}} = \frac{m}{e\left(g_1, g_2\right)^{s\beta t_i}} \tag{2}$$

Now observe that for any vector $\mathbf{x}_i \in \mathbb{Z}_q^{n^+}$ for some $i \in [L]$ and an attribute $\mathbf{y} \in \mathbb{Z}_q^{n^+}$ with $\langle \mathbf{x}_i, \mathbf{y} \rangle = 0$, it also holds that $\langle \widetilde{\mathbf{x}}_i, \widetilde{\mathbf{y}} \rangle = \langle \mathbf{x}_i, \mathbf{y} \rangle + \langle \widetilde{r}_i, 0 \rangle + 1 \cdot 0 = 0$. For brevity, we set up the notations $g_{\mathsf{T}} = e\left(g_1, g_2\right)$ and the discrete logarithm as $\mathsf{DL}(K) = k$, where $K = g_{\mathsf{t}}^k$ for any $k \in \mathbb{Z}_q$ (i.e., irrespective of any group type $\mathsf{t} \in \{1, 2, \mathsf{T}\}$) for the rest of the proof. To ensure correctness with the rest of decryption above, it is thus enough to show that

$$\prod_{w=1}^{n'+1} \left\{ e\left(C_{3,w}^{\widetilde{x}_{w,i}}, A_i\right) \cdot e\left(C_4, \widehat{W}_{w,i}^{\widetilde{x}_{w,i}}\right) \right\} = g_{\mathsf{T}}^{s\beta t_i X_i} \tag{3}$$

so that Dec yields the message $m \in \mathbb{G}_{\mathsf{T}}$. We will analyze individual pairing products in the above form separately. Before that we note a few things about the public keys *after they are fused with the predicate vectors* during Aggr. For any $i \in [L], j \in [0, L]$, we have

$$T_j = \left(\prod_{w\in[n]} U_{w,j}^{-x_{w,j}}\right)\cdot U_{n',j}^{-\widetilde{r}_j} = \prod_{w\in[n']} g_1^{-u_{w,j}\widetilde{x}_{w,j}} = g_1^{-\sum_{w\in[n']} u_{w,j}\widetilde{x}_{w,j}}$$
$$\implies \mathsf{DL}(T_j) = -\sum_{w\in[n']} u_{w,j}\widetilde{x}_{w,j},$$
$$\widetilde{W}_{i,j} = \left(\prod_{w\in[n]} W_{i,j,w}^{x_{w,j}}\right)\cdot W_{i,j,n'}^{\widetilde{r}_j} = \prod_{w\in[n']}\left(A_i^{u_{w,j}/\gamma}\right)^{\widetilde{x}_{w,j}}$$
$$= A_i^{\frac{1}{\gamma}\cdot\sum_{w\in[n']} u_{w,j}\widetilde{x}_{w,j}} = A_i^{-\mathsf{DL}(T_j)/\gamma},$$

where we redefined $\widetilde{x}_{n',0} = \widetilde{r}_0$. Further, for any $w \in [n']$ and $i \in [L]$, we have:

$$\widehat{W}_{w,i}^{\widetilde{x}_{w,i}} = \prod_{j\in[0,L]\setminus\{i\}} W_{i,j,w}^{\widetilde{x}_{w,i}} = \prod_{j\in[0,L]\setminus\{i\}}\left(A_i^{u_{w,j/\gamma}}\right)^{\widetilde{x}_{w,i}} = A_i^{\left(\widetilde{x}_{w,i}\cdot\sum_{j\in[0,L]\setminus\{i\}} u_{w,j}\right)/\gamma} \tag{4}$$

Defining the first pairing product as $\theta_1 = \prod_{w=1}^{n'+1} e\left(C_{3,w}^{\widetilde{x}_{w,i}}, A_i\right)$, we have:

$$\begin{aligned}
\theta_1 &= \prod_{w=1}^{n'+1} e\left(\left(h^{\widetilde{y}_w\cdot r+s}\cdot \widehat{U}_w^{-z}\right)^{\widetilde{x}_{w,i}}, A_i\right)\\
&= \prod_{w=1}^{n'+1}\left\{e\left(h^{r\cdot\widetilde{x}_{w,i}\widetilde{y}_w}, A_i\right)\cdot e\left(h^{s\cdot\widetilde{x}_{w,i}}, A_i\right)\cdot e\left(\widehat{U}_w^{-z\widetilde{x}_{w,i}}, A_i\right)\right\}\\
&= e\left(h^{r\cdot\sum_{w=1}^{n'+1}\widetilde{x}_{w,i}\widetilde{y}_w}, A_i\right)\cdot e\left(g_1^{s\beta\sum_{w=1}^{n'+1}\widetilde{x}_{w,i}}, A_i\right)\cdot\prod_{w=1}^{n'+1} e\left(\widehat{U}_w^{-z\widetilde{x}_{w,i}}, A_i\right)\\
&= e\left(h^0, A_i\right)\cdot e\left(g_1^{s\beta X_i}, g_2^{t_i}\right)\cdot\prod_{w=1}^{n'+1} e\left(\widehat{U}_w^{-z\widetilde{x}_{w,i}}, A_i\right)\\
&= g_{\mathsf{T}}^{s\beta t_i X_i}\ \cdot\ \theta_{11}\ \cdot\ \theta_{12},
\end{aligned}$$

where $\theta_{11} = \prod_{w=1}^{n'} e\left(\widehat{U}_w^{-z\widetilde{x}_{w,i}}, A_i\right)$ and $\theta_{12} = e\left(\widehat{U}_{n'+1}^{-z}, A_i\right)$ (recall $\widetilde{x}_{n'+1,i} = 1$)

$$\theta_{11} = \prod_{w\in[n']} e\left(\prod_{j=0}^{L} U_{w,j}^{-z\widetilde{x}_{w,i}}, A_i\right) = \prod_{w\in[n']} e\left(\left(g_1^{\sum_{j=0}^{L} u_{w,j}}\right)^{-z\widetilde{x}_{w,i}}, g_2^{t_i}\right)$$
$$= \prod_{w\in[n']} g_{\mathsf{T}}^{-zt_i\widetilde{x}_{w,i}\sum_{j=0}^{L} u_{w,j}} = \prod_{w\in[n']} g_{\mathsf{T}}^{zt_i\left(-\widetilde{x}_{w,i}u_{w,i}\right)}\cdot\prod_{w\in[n']} g_{\mathsf{T}}^{-zt_i\widetilde{x}_{w,i}\sum_{j\in[0,L]\setminus\{i\}} u_{w,j}}$$

$$\Rightarrow \theta_{11} = g_{\mathsf{T}}^{zt_i \mathsf{DL}(T_i)} \cdot \zeta_1, \text{ where } \zeta_1 = \prod_{w\in[n']} g_{\mathsf{T}}^{-zt_i \widetilde{x}_{w,i} \sum_{j\in[0,L]\setminus\{i\}} u_{w,j}} \text{ and}$$

$$\theta_{12} = e\left(\widehat{U}_{n'+1}^{-z}, A_i\right) = e\left(\prod_{j=0}^{L} T_j^{-1}, A_i^z\right) = \prod_{j=0}^{L} e\left(T_j^{-1}, A_i^z\right) = \prod_{j=0}^{L} e\left(\prod_{w=1}^{n'} U_{w,j}^{\widetilde{x}_{w,j}}, A_i^z\right)$$

$$= \prod_{j=0}^{L} e\left(g_1^{\sum_{w\in[n']} u_{w,j}\widetilde{x}_{w,j}}, A_i^z\right) = \prod_{j=0}^{L} e\left(g_1^{-\mathsf{DL}(T_j)}, g_2^{zt_i}\right) = \prod_{j=0}^{L} g_{\mathsf{T}}^{-zt_i \mathsf{DL}(T_j)}$$

$$= g_{\mathsf{T}}^{-zt_i \mathsf{DL}(T_i)} \cdot \zeta_2, \text{ where } \zeta_2 = g_{\mathsf{T}}^{-zt_i \sum_{j\in[0,L]\setminus\{i\}} \mathsf{DL}(T_j)}.$$

We have $\theta_1 = g_{\mathsf{T}}^{s\beta t_i X_i} \cdot \left(\cancel{g_{\mathsf{T}}^{zt_i \mathsf{DL}(T_i)}} \cdot \zeta_1\right) \cdot \left(\cancel{g_{\mathsf{T}}^{-zt_i \mathsf{DL}(T_i)}} \cdot \zeta_2\right) \Rightarrow \theta_1 = g_{\mathsf{T}}^{s\beta t_i X_i} \cdot \zeta_1 \cdot \zeta_2$

Defining the second pairing product as $\theta_2 = \prod_{w=1}^{n'+1} e\left(C_4, \widehat{W}_{w,i}^{\widetilde{x}_{w,i}}\right)$, we have:

$$\theta_2 = \left\{\prod_{w\in[n']} e\left(g_1^{z\gamma}, \widehat{W}_{w,i}^{\widetilde{x}_{w,i}}\right)\right\} \cdot e\left(g_1^{z\gamma}, \widehat{W}_{n'+1,i}\right) \text{ (recall } \widetilde{x}_{n'+1,i} = 1 \text{ and } C_4 = \Gamma^z = g^{z\gamma})$$

$$= \left\{\prod_{w\in[n']} e\left(g_1^{z\gamma}, A_i^{\left(\widetilde{x}_{w,i}\cdot\sum_{j\in[0,L]\setminus\{i\}} u_{w,j}\right)/\gamma}\right)\right\} \cdot e\left(g_1^{z\gamma}, \left(\prod_{j\in[0,L]\setminus\{i\}} \widetilde{W}_{i,j}\right)^{-1}\right)$$

$$= \prod_{w\in[n']} e\left(g_1^{z\gamma}, g_2^{\left(t_i\widetilde{x}_{w,i}\cdot\sum_{j\in[0,L]\setminus\{i\}} u_{w,j}\right)/\gamma}\right)\Bigg\} \cdot \prod_{j\in[0,L]\setminus\{i\}} e\left(g_1^{z\gamma}, \left(A_i^{-\mathsf{DL}(T_j)/\gamma}\right)^{-1}\right)$$

$$= \prod_{w\in[n']} g_{\mathsf{T}}^{zt_i\widetilde{x}_{w,i}\cdot\sum_{j\in[0,L]\setminus\{i\}} u_{w,j}} \cdot \prod_{j\in[0,L]\setminus\{i\}} e\left(g_1^{z\gamma}, g_2^{t_i\mathsf{DL}(T_j)/\gamma}\right)$$

$$= \zeta_1^{-1} \cdot \prod_{j\in[0,L]\setminus\{i\}} g_{\mathsf{T}}^{zt_i\mathsf{DL}(T_j)} = \zeta_1^{-1} \cdot g_{\mathsf{T}}^{zt_i\sum_{j\in[0,L]\setminus\{i\}}\mathsf{DL}(T_j)} = \zeta_1^{-1} \cdot \zeta_2^{-1}$$

This completes the proof since

$$\prod_{w=1}^{n'+1}\left\{e\left(C_{3,w}^{\widetilde{x}_{w,i}}, A_i\right)\cdot e\left(C_4, \widehat{W}_{w,i}^{\widetilde{x}_{w,i}}\right)\right\} = \theta_1\cdot\theta_2 = g_{\mathsf{T}}^{s\beta t_i X_i}\cdot\zeta_1\cdot\zeta_2\cdot\zeta_1^{-1}\cdot\zeta_2^{-1} = g_{\mathsf{T}}^{s\beta t_i X_i}.$$

Theorem 6 (Security of Construction 1). *The slotted RIPE scheme* Π_{sRIPE} *with message space* $\mathcal{M} = \mathbb{G}_{\mathsf{T}}$ *and attribute space* $\mathcal{U} = \mathbb{Z}_q^{n^+}$ *from Construction 1 is secure in the* GGM.

Below, we show that our slotted RIPE scheme Π_{sRIPE} (Construction 1) is secure in the generic group model. We start with some notations and definitions for generic and symbolic group models.

Generic Bilinear Group Model. Our definitions for generic bilinear group model is adapted from [4,12]. Let $\mathcal{G} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathsf{T}, q, g_1, g_2, e)$ be a bilinear group setting, $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_\mathsf{T}$ be lists of group elements in $\mathbb{G}_1, \mathbb{G}_2$, and $\mathbb{G}_\mathsf{T}$ respectively. Let $\mathcal{D}$ be a distribution over $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_\mathsf{T}$. The generic group model for a bilinear group setting $\mathcal{G}$ and a distribution $\mathcal{D}$ is described in Fig. 1. In this model, the challenger first initializes the lists $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_\mathsf{T}$ by sampling the group elements according to $\mathcal{D}$, and the adversary receives handles for the elements in the lists. For $\mathsf{t} \in \{1, 2, \mathsf{T}\}$, $\mathcal{L}_\mathsf{t}[h]$ denotes the h-th element in the list $\mathcal{L}_\mathsf{t}$. The handle to this element is simply the pair (t, h). An adversary A running in the generic bilinear group model can apply group operations and the bilinear map e to the elements in the lists. To do this, A has to call the appropriate oracle specifying handles for the input elements. A also gets access to the internal state variables of the challenger via handles, and we assume that the equality queries are "free", in the sense that they do not count when measuring the computational complexity A. For $\mathsf{t} \in \{1, 2, \mathsf{T}\}$, the challenger computes the result of a query, say $\delta \in \mathbb{G}_\mathsf{t}$, and stores it in the corresponding list as $\mathcal{L}_\mathsf{t}[\mathsf{pos}] = \delta$ where pos is its next *empty* position in $\mathcal{L}_\mathsf{t}$, and returns to A its (newly created) handle $(\mathsf{t}, \mathsf{pos})$. Handles are not unique (i.e., the same group element may appear more than once in a list under different handles). As in [4], the equality test oracle in [12] is replaced with the zero-test oracle $\mathsf{Zt}_\mathsf{T}(\cdot)$ that, on input a handle (t, h), returns 1 if $\mathcal{L}_\mathsf{t}[h] = 0$ and 0 otherwise only for the case $\mathsf{t} = \mathsf{T}$.

State: Lists $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_\mathsf{T}$ over $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathsf{T}$ respectively.

Initializations: Lists $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_\mathsf{T}$ sampled according to distribution $\mathcal{D}$.

Oracles: The oracles provide black-box access to the group operations, the bilinear map, and zero-tests.

- $\forall \mathsf{t} \in \{1, 2, \mathsf{T}\}$: $\mathsf{Add}_\mathsf{t}(h_1, h_2)$ appends $\mathcal{L}_\mathsf{t}[h_1] + \mathcal{L}_\mathsf{t}[h_2]$ to $\mathcal{L}_\mathsf{t}$ and returns its handle $(\mathsf{t}, |\mathcal{L}_\mathsf{t}|)$.
- $\forall \mathsf{t} \in \{1, 2, \mathsf{T}\}$: $\mathsf{Neg}_\mathsf{t}(h)$ appends $-\mathcal{L}_\mathsf{t}[h]$ and returns its handle $(\mathsf{t}, |\mathcal{L}_\mathsf{t}|)$.
- $\mathsf{Map}_e(h_1, h_2)$ appends $e(\mathcal{L}_1[h_1], \mathcal{L}_2[h_2])$ and returns its handle $(\mathsf{T}, |\mathcal{L}_\mathsf{T}|)$.
- $\mathsf{Zt}_\mathsf{T}(h)$ returns 1 if $\mathcal{L}_\mathsf{T}[h] = 0$ and 0 otherwise.

All oracles return $\perp$ when given invalid indices.

Fig. 1. GGM for bilinear group setting $\mathcal{G} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathsf{T}, q, g_1, g_2, e)$ and distribution $\mathcal{D}$.

Symbolic Group Model. The symbolic group model (SGM) for a bilinear group setting $\mathcal{G}$ and a distribution $\mathcal{D}$ gives to the adversary the same interface as the corresponding generic group model (GGM), except that internally the challenger stores lists of elements from the ring $\mathbb{Z}_q[\mathtt{x}_1, \ldots, \mathtt{x}_k]$ instead of lists of group elements, where $\{\mathtt{x}_k\}_{k \in \mathbb{N}}$ are indeterminates. The

$\mathsf{Add}_{\mathfrak{t}}(\cdot,\cdot), \mathsf{Neg}_{\mathfrak{t}}(\cdot), \mathsf{Map}_{e}(\cdot,\cdot), \mathsf{Zt}_{\mathsf{T}}(\cdot)$ oracles respectively compute addition, negation, multiplication, and zero tests in the ring. For our proof, we will work in the ring $\mathbb{Z}_q[\mathtt{x}_1, \ldots, \mathtt{x}_k, 1/\mathtt{x}_i]$ for some $i \in [k]$. Note that any element $f \in \mathbb{Z}_q[\mathtt{x}_1, \ldots, \mathtt{x}_k, 1/\mathtt{x}_i]$ can be represented as

$$f(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \sum_{\mathbf{c} \in \mathbb{Z}^k} \eta_{\mathbf{c}} \prod_{i=1}^{k} \mathtt{x}_i^{c_i} \text{ with } \mathbf{c} = (c_1, \ldots, c_k) \in \mathbb{Z}^k$$

using $\{\eta_{\mathbf{c}} \in \mathbb{Z}_q\}_{\mathbf{c} \in \mathbb{Z}^k}$, where $\eta_{\mathbf{c}} = 0$ for all but finite $\mathbf{c} \in \mathbb{Z}^k$. Note that this expression is unique. We now begin our proof for Theorem 6 below.

Proof. At a high level, we show a sequence of hybrids each of which is a game between a challenger and a PPT adversary A. In the first (resp., last) hybrid, the challenger encrypts a pair $(\mathbf{y}_b, m_b)$ corresponding to bit $b = 0$ (resp., $b = 1$). The intermediate hybrids ensure that the distributions in any pair of subsequent hybrids from the first to the last one are statistically indistinguishable.

Since the proof is in the GGM, w.l.o.g. the challenger simulates all the generic bilinear group oracle queries for A. In particular, the challenger stores the actual computed elements in the list $\mathcal{L}_{\mathfrak{t}}$ based on its group type $\mathfrak{t} \in \{1, 2, \mathsf{T}\}$. The handle to an actual element stored in any of these lists are just a tuple $(\mathfrak{t}, \mathsf{pos})$ specifying the group type $\mathfrak{t}$ and its position in the table $\mathcal{L}_{\mathfrak{t}}$. Since our scheme contains several variables, we will refrain from explicitly specifying the handles to the actual elements for convenience. Further, when we move to the SGM, we will denote any literal variable v as $\mathtt{v}$ and composite terms like $v_1 v_2$ (resp., $\frac{v_1}{v_2}$) as $\mathtt{v}_1\mathtt{v}_2$ (resp., $\frac{\mathtt{v}_1}{\mathtt{v}_2}$) to represent an individual monomial in a (possibly multivariate) polynomial. For variables denoted with Greek alphabets, say α, β, γ, we represent their corresponding formal variables as $\upalpha, \upbeta, \upgamma$. We also define $\mathbb{Z}_q$-$\mathsf{span}(\mathcal{S})$ as the set of $\mathbb{Z}_q$-linear combinations of all elements in any set $\mathcal{S}$. Assume A issues an arbitrary polynomial number $Q_{\mathsf{zt}}(\lambda)$ of queries to its Zt_{T} oracle in each hybrid.

$\mathbf{H}_1(\lambda)$: This is the real scheme corresponding to bit $b = 0$ in the GGM. In more detail, the hybrid goes as follows.

- **Setup phase:** A sends an attribute length $n = n(\lambda)$ and slot count $L = L(\lambda)$ to the challenger, upon which it first initializes $\mathsf{ctr} = 0$, a dictionary D, and the set $\mathcal{C}_L = \emptyset$ to account for corrupted slots. Next, it computes $\mathcal{G} = (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_{\mathsf{T}}, q, g_1, g_2, e) \leftarrow\!\!\$\ \mathsf{GroupGen}(1^\lambda)$ and initializes three tables as $\mathcal{L}_{\mathfrak{t}}[1] = g_{\mathfrak{t}}, \forall \mathfrak{t} \in \{1, 2, \mathsf{T}\}$. The challenger prepares a tuple $\big(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_{\mathsf{T}}, q, \{(\mathfrak{t}, 1)\}_{\mathfrak{t} \in \{1,2,\mathsf{T}\}}\big)$, where $(\mathfrak{t}, 1)$ represents the handle to $g_{\mathfrak{t}}, \forall \mathfrak{t} \in \{1, 2, \mathsf{T}\}$. To allow A to compute the group operations including the bilinear map e, the challenger also simulates all the oracles $\mathsf{Add}_{\mathfrak{t}}, \mathsf{Neg}_{\mathfrak{t}}, \mathsf{Map}_{e}, \mathsf{Zt}_{\mathsf{T}}$ with implicit access to the lists $\{\mathcal{L}_{\mathfrak{t}}\}_{\mathfrak{t} \in \{1,2,\mathsf{T}\}}$. It then computes the crs components as follows:

1. Set $n' = n + 1$. Compute $h = g_1^{\beta}, \Gamma = g_1^{\gamma} \in \mathbb{G}_1$ and $Z = e(g_1, g_2)^{\alpha} \in \mathbb{G}_{\mathsf{T}}$ as in the real Setup algorithm. Update $\mathcal{L}_1$ with the elements β, γ and $\mathcal{L}_{\mathsf{T}}$ with α respectively.

2. For each slot index $i \in [0, L]$, do the following:
   (a) $\forall w \in [n']$, compute $U_{w,i} = g_1^{u_{w,i}} \in \mathbb{G}_1$ as in the real scheme and update $\mathcal{L}_1$ with $u_{w,i}$.
   (b) $\forall i > 0$, compute $A_i = g_2^{t_i}, B_i = g_2^{\alpha+\beta \cdot t_i} \in \mathbb{G}_2$ as in the real scheme and update $\mathcal{L}_2$ with $t_i, (\alpha + \beta \cdot t_i)$ in order.
   (c) $\forall i > 0, w \in [n']$ and for each $j \in [0, L] \setminus \{i\}$, compute $W_{i,j,w} = g_2^{\frac{t_i \cdot u_{j,w}}{\gamma}} \in \mathbb{G}_2$ as in the real scheme and update $\mathcal{L}_2$ with $\frac{t_i \cdot u_{j,w}}{\gamma}$.
3. For $\widetilde{\mathbf{x}}_0 = (\widetilde{x}_{1,0}, \ldots, \widetilde{x}_{n',0}) \leftarrow\!\!\$\ \mathbb{Z}_q^{n'+}$, set $\mathsf{pk}_0 = \left(T_0, \left\{\widetilde{W}_{i,0}\right\}_{i\in[L]}\right)$ as in the real scheme. Define $u'_0 = \sum_{w=1}^{n'} u_{w,0} \cdot \widetilde{x}_{w,0} = -\mathsf{DL}(T_0)$ so that

$$T_0 = g_1^{u'_0} \in \mathbb{G}_1 \quad , \quad \widetilde{W}_{i,0} = g_2^{\frac{t_i \cdot u'_0}{\gamma}} \in \mathbb{G}_2, \forall i \in [L].$$

   Update $\mathcal{L}_1$ with u'_0 and $\mathcal{L}_2$ with $\left\{\frac{t_i \cdot u'_0}{\gamma}\right\}_{i\in[L]}$ in order.
4. Set

$$\mathsf{crs} = \Big(\mathcal{G}, Z, h, \Gamma, \{A_i, B_i\}_{i\in[L]}, \left\{\{U_{w,i}\}_{i\in[0,L]}, \{W_{i,j,w}\}_{i\in[L],j\in[0,L]\setminus\{i\}}\right\}_{w\in[n']}, \mathsf{pk}_0\Big).$$

5. Return to A a tuple crs' that includes $\left(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\mathsf{T}, q, \{(\mathfrak{t}, 1)\}_{\mathfrak{t}\in\{1,2,\mathsf{T}\}}\right)$ along with the handles to all elements in the same order as they are arranged in the crs above.

- **Pre-challenge query phase:** A can issue *key generation* queries or *corruption* queries in this phase.

1. Consider the key-generation queries first. Upon getting a slot index $i \in [L]$, the challenger updates $\mathsf{ctr} = \mathsf{ctr} + 1$, sets $\mathbf{x}_i^{\mathsf{ctr}} = \mathbf{x}_i$ and does the following:
   (a) Sample $\widetilde{r}_i^{\mathsf{ctr}} \leftarrow\!\!\$\ \mathbb{Z}_q^+$ and compute $\mathsf{pk}_i^{\mathsf{ctr}} = \left(T_i^{\mathsf{ctr}}, \left\{\widetilde{W}_{j,i}^{\mathsf{ctr}}\right\}_{j\in[L]\setminus\{i\}}\right)$ as in KGen.
   (b) Note that the element $T_i^{\mathsf{ctr}} \in \mathbb{G}_1$ from $\mathsf{pk}_i^{\mathsf{ctr}}$ has the following structure:

$$T_i^{\mathsf{ctr}} = g_1^{-\widetilde{r}_i^{\mathsf{ctr}} u_{n',i}}, \text{ where } \mathsf{sk}_i^{\mathsf{ctr}} = \widetilde{r}_i^{\mathsf{ctr}} \text{ is the secret key.}$$

   Even given the handle to $u_{n',i}$, A cannot compute a handle for $\mathsf{DL}(T_i^{\mathsf{ctr}}) = -\widetilde{r}_i^{\mathsf{ctr}} u_{n',i}$ on its own. Hence, the challenger updates $\mathcal{L}_1$ with $\mathsf{DL}(T_i^{\mathsf{ctr}})$.
   (c) Further, each term in $\left\{\widetilde{W}_{j,i}^{\mathsf{ctr}} \in \mathbb{G}_2\right\}_{j\in[L]\setminus\{i\}}$ has the following structure:

$$\widetilde{W}_{j,i}^{\mathsf{ctr}} = W_{j,i,n'}^{\widetilde{r}_i^{\mathsf{ctr}}} = g_2^{\frac{t_j u_{n',i}}{\gamma} \cdot \widetilde{r}_i^{\mathsf{ctr}}}$$

   For reasons similar to Item (b) above, the challenger updates $\mathcal{L}_2$ with each element individually from the set $\left\{\widetilde{r}_i^{\mathsf{ctr}} \cdot \frac{t_j u_{n',i}}{\gamma}\right\}_{j\in[L]\setminus\{i\}}$.

(d) Define $\mathsf{pk}_{\mathsf{ctr}} = \mathsf{pk}_i^{\mathsf{ctr}}, \mathsf{sk}_{\mathsf{ctr}} = \mathsf{sk}_i^{\mathsf{ctr}}$ and $\mathsf{pk}'_{\mathsf{ctr}}$ as a sequence of handles to all elements in the same order as they are arranged in $\mathsf{pk}_{\mathsf{ctr}}$.
(e) Return the tuple $(\mathsf{ctr}, \mathsf{pk}'_{\mathsf{ctr}})$ to A and update $\mathsf{D}[\mathsf{ctr}] = (i, \mathsf{pk}_{\mathsf{ctr}}, \mathsf{sk}_{\mathsf{ctr}})$.

2. When A sends $c \in [\mathsf{ctr}]$ issuing a corruption query, the challenger returns sk' to A where $\mathsf{D}[c] = (i', \mathsf{pk}', \mathsf{sk}')$.

- **Challenge phase:** In this phase, A specifies the following challenge information:

$$\{(c_i, \mathbf{x}_i, \mathsf{pk}_i^*)\}_{i\in[L]} \quad \text{and} \quad ((\mathbf{y}_0, m_0), (\mathbf{y}_1, m_1)) \in (\mathbb{Z}_q^{n+} \times \mathbb{G}_{\mathsf{T}})^2.$$

Preprocessing the challenge information. For each $i \in [L]$, the challenger checks that $\mathbf{x}_i \neq \mathbf{0}^n$ and does the following:

1. If $c_i \in [\mathsf{ctr}]$, it checks $\mathsf{D}[c_i] = (i', \mathsf{pk}', \mathsf{sk}')$. If $i \neq i'$, it halts. Else, it sets $\mathsf{pk}_i^* = \mathsf{pk}'$. Further, if A issued a corruption query for c_i before, it updates $\mathcal{C}_L = \mathcal{C}_L \cup \{i\}$.
2. If $c_i = \perp$, pk_i^* represents a corrupt secret key generated by A itself. Hence, it parses pk_i^* and halts if $\mathsf{IsValid}(\mathsf{crs}, i, \mathsf{pk}_i^*) = 0$.[7] Else, it updates $\mathcal{C}_L = \mathcal{C}_L \cup \{i\}$.

Computing key aggregation. The challenger then computes

$$\left(\mathsf{mpk}, (\mathsf{hsk}_i)_{i\in[L]}\right) = \mathsf{Aggr}\left(\mathsf{crs}, ((\mathsf{pk}_i^*, \mathbf{x}_i))_{i\in[L]}\right), \text{ where}$$

$$\mathsf{mpk} = (\mathcal{G}, g, h, Z, \Gamma, \{\widehat{U}_w\}_{w'\in[n'+1]}), \text{ and } \{\mathsf{hsk}_i = (\mathcal{G}, i, A_i, B_i, \{\widehat{W}_{w,i}\}_{w\in[n'+1]})\}_{i\in[L]}.$$

Computing the challenge ciphertext. The challenger now uses mpk and the pair $(\mathbf{y}_0, m_0)$, and generates $c^* \leftarrow_{\$} \mathsf{Enc}(\mathsf{mpk}, \mathbf{y}_0, m_0)$ where $c^* = \left(C_1, C_2, \{C_{3,w}\}_{w\in[n'+1]}, C_4\right)$.

1. Note that $C_1 = m_0 \cdot e(g_1, g_2)^{\alpha s} \in \mathbb{G}_{\mathsf{T}}$ and $C_2 = g_1^s \in \mathbb{G}_1$. Accordingly, the challenger updates $\mathcal{L}_{\mathsf{T}}$ with αs and $\mathcal{L}_1$ with s respectively.
2. With $\widetilde{\mathbf{y}}_0 = (\mathbf{y}_0, 0, 0) = (y_1^0, \ldots, y_n^0, 0, 0)$, note that the elements $\{C_{3,w} \in \mathbb{G}_1\}_{w\in[n'+1]}$ have the following structure:

$$\text{for all } w \in [n],\ C_{3,w} = h^{y_w^0 \cdot r + s} \cdot \widehat{U}_w^{-z} = g_1^{r\beta y_w^0 + s\beta - z\cdot u_w}$$
$$\text{for } w = n',\ C_{3,n'} = g_1^{r\beta\cdot 0 + s\beta - z\cdot u'_n} = g_1^{s\beta - z\cdot u_{n'}}$$
$$\text{for } w = n'+1, C_{3,n'+1} = g_1^{r\beta\cdot 0 + s\beta} \cdot \widehat{U}_{n'+1}^{-z} = g_1^{s\beta} \cdot \prod_{i=0}^{L} T_i^{-z}$$
$$= g_1^{s\beta} \cdot \prod_{i=0}^{L} g_1^{z\sum_{w=1}^{n'} \widetilde{x}_{w,i}\cdot u_{w,i}}$$

[7] By Definition 1, A is supposed to send well-formed keys that passes the $\mathsf{IsValid}(\mathsf{crs}, \cdot, \cdot)$ test. Hence, from now on we do not mention it any more, but assume the challenger checks it implicitly.

$$= g_1^{s\beta} \cdot \prod_{i=0}^{L} g_1^{z \cdot u'_i} = g_1^{s\beta + z \cdot u'_0 - z \sum_{i=1}^{L} \mathsf{DL}(T_i)}$$

$$where u_w = \sum_{i=0}^{L} u_{w,i} \text{ ,and } u_{n'} = \sum_{i=0}^{L} u_{n',i}.$$

Accordingly, the challenger updates $\mathcal{L}_1$ with the elements $\{r\beta y_w^0 + s\beta - z \cdot u_w\}_{w \in [n]}$, $(s\beta - z \cdot u_{n'})$, and $\left[s\beta + z \cdot u'_0 - z \cdot \sum_{i=1}^{L} \mathsf{DL}(T_i)\right]$ in order.

3. Since $C_4 = g_1^{\gamma z} \in \mathbb{G}_1$, it updates $\mathcal{L}_1$ with $z\gamma$.

Group oracle queries. Since Aggr is deterministic, A is able to compute $\left(\mathsf{mpk}, (\mathsf{hsk}_i)_{i \in [L]}\right)$ on its own. In the GGM, A is able to compute handles for the elements in mpk and $\{\mathsf{hsk}_i\}_{i \in [L]}$. To this end, it queries the appropriate group oracles to generate such handles as follows:

1. A only needs to compute the handles for $\{\widehat{U}_w\}_{w \in [n'+1]}$ to complete its information about mpk. Note that $\forall w \in [n']$, $\widehat{U}_w = \prod_{i=0}^{L} U_{w,i} = g_1^{u_w}$, where $u_w = \sum_{i=0}^{L} u_{w,i}$. Hence, $\forall w \in [n']$, A invokes the Add_1 oracle L times *iteratively* on the handles in $\mathcal{L}_1$ for $\{u_{w,i}\}_{i \in [0,L]}$ and gets a handle for u_w. Further, to get a handle for $\widehat{U}_{n'+1} = \prod_{i=0}^{L} T_i$, it has to first a get a handle for each T_i that is fused with the predicate $\mathbf{x}_i$. Note the structure of each T_i after Step (2) in Aggr:

$$T_i = g_1^{\sum_{w=1}^{n'} -\widetilde{x}_{w,i} \cdot u_{w,i}} = g_1^{\sum_{w=1}^{n} (-x_{w,i} \cdot u_{w,i})} \times g_1^{\left(-\widetilde{r}_i \cdot u_{n',i}\right)} \in \mathbb{G}_1.$$

   Given a handle for the second multiplicand, it is easy to note that the first one is publicly computable using Neg_1 and Add_1 oracles. Once A obtains the handles for $\{T_i\}_{i \in [L]}$, it can call Add_1 oracle on these handles to get the same for $\widehat{U}_{n'+1}$.
2. A only needs to compute the handles for $\{\widehat{W}_{w,i}\}_{w \in [n'+1]}$ to get complete information about hsk_i for each $i \in [L]$. Note that $\forall w \in [n']$, $\widehat{W}_{w,i} = \prod_{j \in [0,L] \setminus \{i\}} W_{i,j,w} = g_2^{t_i \cdot (u_w - u_{w,i})/\gamma}$, since $(u_w - u_{w,i}) = \sum_{j \in [0,L] \setminus \{i\}} u_{w,j}$. It is again easy to see that a handle for such an element can be computed by calling the Add_2 oracle $L - 1$ times *iteratively* on the handles in $\mathcal{L}_2$ for $\left\{\frac{t_i \cdot u_{w,j}}{\gamma}\right\}_{j \in [0,L] \setminus \{i\}}$. Further, note that to get a handle for $\widehat{W}_{n'+1,i} = \prod_{j \in [0,L] \setminus \{i\}} \widetilde{W}_{i,j}^{-1}$, it has to first a get a handle for each $\widetilde{W}_{j,i}$ that is fused with the predicate $\mathbf{x}_i$. Note the structure of each $\widetilde{W}_{j,i}$ after Step (2) in Aggr:

$$\widetilde{W}_{j,i} = \left(\prod_{w=1}^{n} W_{i,j,w}^{\widetilde{x}_{w,i}}\right) \cdot W_{i,j,n'}^{\widetilde{r}_i} = g_2^{\sum_{w=1}^{n} \frac{t_j u_{w,i}}{\gamma} \cdot x_{w,i}} \times g_2^{\left(\frac{t_j u_{n',i}}{\gamma} \cdot \widetilde{r}_i\right)} \in \mathbb{G}_2.$$

   Again, given a handle for the second multiplicand, the same can be computed publicly for the entire product using handles for $\{W_{i,j,w}\}$. Once A obtains the handles to each element in $\{\widetilde{W}_{j,i}\}_{j \in [L] \setminus \{i\}}$, it can call Add_2 oracle on these handles to get the same for $\widehat{W}_{n+1,i}$.

3. Finally, it defines mpk' and each hsk'_i as sequences of handles to all elements (except $i, \mathbf{x}_i$) in the same order as arranged in mpk and each $\mathsf{hsk}_i, \forall i \in [L]$.

- **Output phase:** A outputs a bit $b' \in \{0, 1\}$.

For ease of presentation, in Table 2 we show all unit and composite terms generated in the scheme itself, and stored in the respective lists.

Table 2. The above table shows all terms from the scheme for which handles are stored in the respective lists $\mathcal{L}_\mathsf{t}, \forall \mathsf{t} \in \{1, 2, \mathsf{T}\}$. Assume $\mathcal{A}$ issues some arbitrary polynomial number, Q_k, of key queries in the pre-challenge query phase (some of which may be corrupted). The table lists all the terms for each of these keys $\{\mathsf{pk}_c\}_{c\in[Q_\mathsf{k}]}$ received by $\mathcal{A}$ in the second row. Hence, these terms are also indexed with superscripts for the key query count $c \in [Q_\mathsf{k}]$ (along with the slot index, say $i \in [L]$, for which $\mathcal{A}$ asked the key). The terms corresponding to mpk and hsk_i are not shown in the table, since the handles for these are publicly computable by $\mathcal{A}$ using the group oracles. Note that such terms correspond to keys for all the registered L slots (possibly all of which may be corrupted or even adversarially generated). Hence, the individual variables in each of those terms in mpk and hsk_i are independent of the counter variable $c \in [Q_\mathsf{k}]$ respectively. In c, observe that we also have $(\mathsf{DL}(m) + \alpha s)$ in $\mathcal{L}_\mathsf{T}$, where $\mathsf{DL}(m) \in \mathbb{Z}_q$ is w.r.t. g_T.

	$\mathcal{L}_1$	$\mathcal{L}_2$	$\mathcal{L}_\mathsf{T}$
crs	$\boxed{g_1}$, $\boxed{\beta}$, $\boxed{\gamma}$ $\boxed{u'_0} = \sum_{w=1}^{n'} u_{w,0}\widetilde{x}_{w,0}$, $\left\{\boxed{u_{w,i}}\right\}_{i\in[0,L],w\in[n']}$	$\boxed{g_2}$, $\left\{\boxed{t_i}, \boxed{\alpha+\beta t_i}\right\}_{i\in[L]}$ $\boxed{\frac{t_i u'_0}{\gamma}}$, $\left\{\boxed{\frac{t_i u_{w,j}}{\gamma}}\right\}_{\substack{i\in[L]\\ j\in[0,L]\setminus\{i\}\\ w\in[n']}}$	$\boxed{g_\mathsf{T}}$ $\boxed{\alpha}$
$\{\mathsf{pk}_c\}_{c\in[Q_\mathsf{k}]}$	$\left\{\boxed{-\widetilde{r}_i^c u_{n',i}}\right\}_{c\in[Q_\mathsf{k}]}$ $\left(\text{for } \{T_i^c\}_{c\in[Q_\mathsf{k}]}\right)$	$\left\{\boxed{\widetilde{r}_i^c \cdot \frac{t_j u_{n',i}}{\gamma}}\right\}_{\substack{j\in[L]\setminus\{i\}\\ c\in[Q_\mathsf{k}]}}$ $\left(\text{for } \left\{\widetilde{W}_{j,i}^c\right\}_{j\in[L]\setminus\{i\},c\in[Q_\mathsf{k}]}\right)$	–
c	$\boxed{s}$ (for C_2) , $\boxed{z\gamma}$ (for C_4) , $\boxed{r\beta y_w^0 + s\beta - z u_w}$ (for $C_{3,w}, \forall w \in [n]$), $\boxed{s\beta - z u_{n'}}$ (for $C_{3,n'}$), where $u_{n'} = \sum_{i=0}^{L} u_{n',i}$ $\boxed{s\beta - z\mathsf{DL}(T)}$ (for $C_{3,n'+1}$), where $\mathsf{DL}(T) = \sum_{i=0}^{L} \mathsf{DL}(T_i)$	–	$\boxed{\mathsf{DL}(m) + \alpha s}$

$\mathbf{H}_2(\lambda)$: In this hybrid, we switch to the SGM *partially*. Namely, the interaction between challenger and A remains exactly as it was in $\mathbf{H}_1(\lambda)$, but now the challenger stores formal variables for all the terms from Table 2 in the respective lists $\mathcal{L}_{\mathtt{t}}, \forall \mathtt{t} \in \{1, 2, \mathsf{T}\}$. Thus, all the handles A receives refer to multivariate polynomials from the following ring:

$$\zeta = \mathbb{Z}_q \Big[\alpha, \beta, \gamma, \mathtt{u}'_0, \{\mathtt{u}_{\mathtt{w},\mathtt{i}}\}_{\mathtt{i}\in[\mathtt{L}],\mathtt{w}\in[\mathtt{n}']}, \{\widetilde{\mathtt{r}}^{\mathtt{c}}_{\mathtt{i}}\}_{\mathtt{i}\in[\mathtt{L}],\mathtt{c}\in[\mathtt{Q_k}]}, \{\mathtt{t}_{\mathtt{i}}\}_{\mathtt{i}\in[\mathtt{L}]}, \frac{1}{\gamma}, \mathtt{s}, \mathtt{r}, \mathtt{z}, \{\mathtt{y}_{\mathtt{w}}\}_{\mathtt{w}\in[\mathtt{n}'+1]}\Big].$$

Concretely, A gets handles to formal polynomials from $\mathcal{L}_{\mathtt{t}}$ for each $\mathtt{t} \in \{1, 2, \mathsf{T}\}$, where:

1. $\mathcal{L}_{\mathsf{T}} = \{1, \alpha, \mathsf{DL}(m) + \alpha\mathtt{s}\}$.
2. $\mathcal{L}_1 = \mathcal{L}_1^{\mathsf{crs}} \cup \mathcal{L}_1^{\mathsf{key}} \cup \mathcal{L}_1^{c}$, where
   (a) $\mathcal{L}_1^{\mathsf{crs}} = \big(1, \beta, \gamma, \mathtt{u}'_0, \{\mathtt{u}_{\mathtt{w},\mathtt{i}}\}_{\mathtt{i}\in[0,\mathtt{L}],\mathtt{w}\in[\mathtt{n}']}\big)$,
   (b) $\mathcal{L}_1^{\mathsf{key}} = \Big(\{-\widetilde{\mathtt{r}}^{\mathtt{c}}_{\mathtt{i}}\mathtt{u}_{\mathtt{n}',\mathtt{i}}\}_{c\in[Q_k]}\Big)$ for some $i \in [L]$, and
   (c) $\mathcal{L}_1^{c} = \Big(\mathtt{s}, \mathtt{zg}, \Big\{\mathtt{rby}_{\mathtt{w}} + \mathtt{sb} - \mathtt{z}\sum_{\mathtt{i}=0}^{\mathtt{L}} \mathtt{u}_{\mathtt{w},\mathtt{i}}\Big\}_{w\in[n]}, \mathtt{sb} - \mathtt{zu}_{\mathtt{n}'}, \mathtt{sb} - \mathtt{zDL}(\mathtt{T})\Big)$.
3. $\mathcal{L}_2 = \mathcal{L}_2^{\mathsf{crs}} \cup \mathcal{L}_2^{\mathsf{key}}$, where
   (a) $\mathcal{L}_2^{\mathsf{crs}} = \Big(1, \{\mathtt{t}_{\mathtt{i}}, \mathtt{a} + \mathtt{bt}_{\mathtt{i}}\}_{\mathtt{i}\in[\mathtt{L}]}, \frac{\mathtt{t}_{\mathtt{i}}\mathtt{u}'_0}{\mathtt{g}}, \Big\{\frac{\mathtt{t}_{\mathtt{i}}\mathtt{u}_{\mathtt{w},\mathtt{j}}}{\mathtt{g}}\Big\}_{\mathtt{i}\in[\mathtt{L}],\mathtt{j}\in[\mathtt{0},\mathtt{L}]\setminus\{\mathtt{i}\},\mathtt{w}\in[\mathtt{n}']}\Big)$, and
   (b) $\mathcal{L}_2^{\mathsf{key}} = \Big(\Big\{\frac{\widetilde{\mathtt{r}}^{\mathtt{c}}_{\mathtt{i}}\mathtt{t}_{\mathtt{j}}\mathtt{u}_{\mathtt{n}',\mathtt{i}}}{\mathtt{g}}\Big\}_{j\in[L]\setminus\{i\},c\in[Q_k]}\Big)$ for some $i \in [L]$.

However, when A issues any zero-test query via Zt_{T} oracle, the challenger replaces the formal variables with their corresponding elements from $\mathbb{Z}_q$. In this case, if the variable is not assigned a value in $\mathbb{Z}_q$, it samples the corresponding value from the same distribution as it did in $\mathbf{H}_1(\lambda)$. However, once a value is assigned to a variable, it is fixed throughout the rest of $\mathbf{H}_2(\lambda)$. We show in [25] that $\mathbf{H}_1(\lambda) \equiv \mathbf{H}_2(\lambda)$.

Given the tuple $\mathsf{P} = (\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_{\mathsf{T}})$, we define $\mathcal{C}(\mathcal{L}_{\mathsf{T}}) = \mathcal{L}_{\mathsf{T}} \cup \{V_1 \cdot V_2 \mid \forall V_1 \in \mathcal{L}_1, V_2 \in \mathcal{L}_2\}$. Basically, it is the set of all monomials from ζ with variables in $\mathbb{G}_{\mathsf{T}}$ that $\mathcal{A}$ can compute querying Map_e on the handles it received for elements in $\mathcal{L}_1, \mathcal{L}_2$. We estimate the size of $\mathcal{C}(\mathcal{L}_{\mathsf{T}})$ as follows. Note that we have $|\mathcal{C}(\mathcal{L}_{\mathsf{T}})| = |\mathcal{L}_{\mathsf{T}}| + |\mathcal{L}_1| \cdot |\mathcal{L}_2|$ where $|\mathcal{L}_{\mathsf{T}}| = 3$,

$$\begin{aligned}
|\mathcal{L}_1| &= |\mathcal{L}_1^{\mathsf{crs}}| + \left|\mathcal{L}_1^{\mathsf{key}}\right| + |\mathcal{L}_1^{c}| \\
&\leq \{(L+1)n' + 4\} + LQ_{\mathsf{k}} + (n+4) = L(n + Q_{\mathsf{k}} + 1) + 2n + 9, \text{ and} \\
|\mathcal{L}_2| &= |\mathcal{L}_2^{\mathsf{crs}}| + \left|\mathcal{L}_2^{\mathsf{key}}\right| \\
&\leq \{2 + 2L + n'L^2\} + \{L(L-1)Q_{\mathsf{k}}\} = L^2(n + Q_{\mathsf{k}} + 1) - L(Q_{\mathsf{k}} - 2) + 2.
\end{aligned}$$

There are several variables in ζ and several terms in $\mathcal{L}_1, \mathcal{L}_2$. Hence, for brevity, we do not state all the elements of $\mathcal{C}(\mathcal{L}_\mathsf{T})$ explicitly with all possible cross combinations of the monomials from $\mathcal{L}_1, \mathcal{L}_2$. However, it is easy to see by inspection that the maximal total degree of each term in $\mathcal{C}(\mathcal{L}_\mathsf{T})$ is $d = 7$ corresponding to the term $\left[\mathtt{rby}_\mathtt{w} \cdot \frac{\widetilde{\mathtt{r}}_\mathtt{i}^\mathtt{c} \mathtt{t}_\mathtt{j} \mathtt{u}_{\mathtt{n}',\mathtt{i}}}{\mathtt{g}}\right]$ for any $w \in [n'], i \in [L], j \in [0, L] \setminus \{i\}, c \in [Q_\mathsf{k}]$. We also note that any handle submitted by $\mathcal{A}$ to the Zt_T oracle during its interaction refers to a polynomial $f \in \zeta$ as

$$f\left(\alpha, \beta, \gamma, \mathtt{u}'_0, \{\mathtt{u}_{\mathtt{w},\mathtt{i}}\}_{\mathtt{i}\in[\mathtt{L}],\mathtt{w}\in[\mathtt{n}']}, \{\widetilde{\mathtt{r}}_\mathtt{i}^\mathtt{c}\}_{\mathtt{i}\in[\mathtt{L}],\mathtt{c}\in[\mathtt{Q}_\mathtt{k}]}, \{\mathtt{t}_\mathtt{i}\}_{\mathtt{i}\in[\mathtt{L}]}, \frac{1}{\gamma}, \mathtt{s}, \mathtt{r}, \mathtt{z}, \{\mathtt{y}_\mathtt{w}\}_{\mathtt{w}\in[\mathtt{n}'+1]}\right) = \sum_{\theta \in \mathcal{C}(\mathcal{L}_\mathsf{T})} \eta_\Theta \Theta,$$

where the coefficients $\{\eta_\Theta \in \mathbb{Z}_q\}_{\Theta \in \mathcal{C}(\mathcal{L}_\mathsf{T})}$ can be computed efficiently. Further, since all the monomials in $\mathcal{C}(\mathcal{L}_\mathsf{T})$ are distinct, the coefficients η_Θ are unique.

$\mathbf{H}_3(\lambda)$: In this hybrid, *all* queries to Zt_T oracle are answered using formal variables. Namely, for any Zt_T query on a handle to a polynomial $f \in \zeta$, the challenger returns 1 if:

$$f\left(\alpha, \beta, \gamma, \mathtt{u}'_0, \{\mathtt{u}_{\mathtt{w},\mathtt{i}}\}_{\mathtt{i}\in[\mathtt{L}],\mathtt{w}\in[\mathtt{n}']}, \{\widetilde{\mathtt{r}}_\mathtt{i}^\mathtt{c}\}_{\mathtt{i}\in[\mathtt{L}],\mathtt{c}\in[\mathtt{Q}_\mathtt{k}]}, \{\mathtt{t}_\mathtt{i}\}_{\mathtt{i}\in[\mathtt{L}]}, \frac{1}{\gamma}, \mathtt{s}, \mathtt{r}, \mathtt{z}, \{\mathtt{y}_\mathtt{w}\}_{\mathtt{w}\in[\mathtt{n}'+1]}\right) = 0.$$

We show in [25] that $\mathbf{H}_2(\lambda) \approx \mathbf{H}_3(\lambda)$.

$\mathbf{H}_4(\lambda)$: In this hybrid, the challenge ciphertext computes an encryption of m_0 with respect to $\mathbf{y}_1$. That is, everything remains as it is in $\mathbf{H}_3(\lambda)$ except that the challenger generates

$$c^* = (C_1, C_2, \{C_{3,w}\}_{w\in[n'+1]}, C_4) \leftarrow\!\!\$\ \mathsf{Enc}(\mathsf{mpk}, \mathbf{y}_1, m_0).$$

Arguing indistinguishability between $\mathbf{H}_3(\lambda)$ and $\mathbf{H}_4(\lambda)$ is the crux of this proof. We provide this analysis in our full version [25]. From here on, the challenger moves to $\mathbf{H}_6(\lambda)$ directly if $m_0 = m_1$. Else if $m_0 \neq m_1$, it still moves to $\mathbf{H}_6(\lambda)$, but via the next hybrids.

$\mathbf{H}_{5,1}(\lambda)$: In this hybrid, $Z^s \in \mathbb{G}_\mathsf{T}$ is replaced with $\mathfrak{U} \leftarrow\!\!\$\ \mathbb{G}_\mathsf{T}$.
$\mathbf{H}_{5,2}(\lambda)$: In this hybrid, the challenge ciphertext encrypts m_1 instead of m_0.
$\mathbf{H}_{5,3}(\lambda)$: In this hybrid, $\mathfrak{U}$ is changed to the honestly computed Z^s again.
$\mathbf{H}_6(\lambda)$: In this hybrid, the challenger moves to GGM from the symbolic setting of SGM. This is the real scheme corresponding to bit $b = 1$ in the GGM.

Hybrid Indistinguishability. Due to space constraints, we defer all the formal proofs for the indistinguishability of hybrids in our full version [25].

Final pairing-based RIPE scheme. By combining the slotted RIPE scheme of Construction 1 and the ("powers-of-two") transformation provided in our full version [25] , we obtain a secure registered IPE with an extra $O(\log L)$ factor in all its compactness and efficiency measures.

7 Implementation and Benchmarks

We developed a Python prototype[8] of our sRIPE scheme from Sect. 6 with the BLS12-381 elliptic curve for pairings, which we implemented via the petrelic Python wrapper [47] around RELIC [10]. This configuration allows each element in $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ to be represented using 49, 97 and 384 bytes respectively. We obtained the benchmarks below on a personal computer with an Intel Core i7-10700 3.8GHz CPU and 128GB of RAM running Ubuntu 22.04.1 LTS with kernel 5.15.0-58-generic. Exponentiations in $\mathbb{G}_1$ (resp., $\mathbb{G}_2$) and each pairing took 0.13 (resp., 0.18) milliseconds and 0.68 milliseconds on average on our machine.

Benchmarks. We provide benchmarks in Fig. 2, showing the sizes of mpk and the |crs| as well as the execution times of setup, aggregate, encrypt and decrypt in relation to parameters L and n. For encryption and decryption, we executed

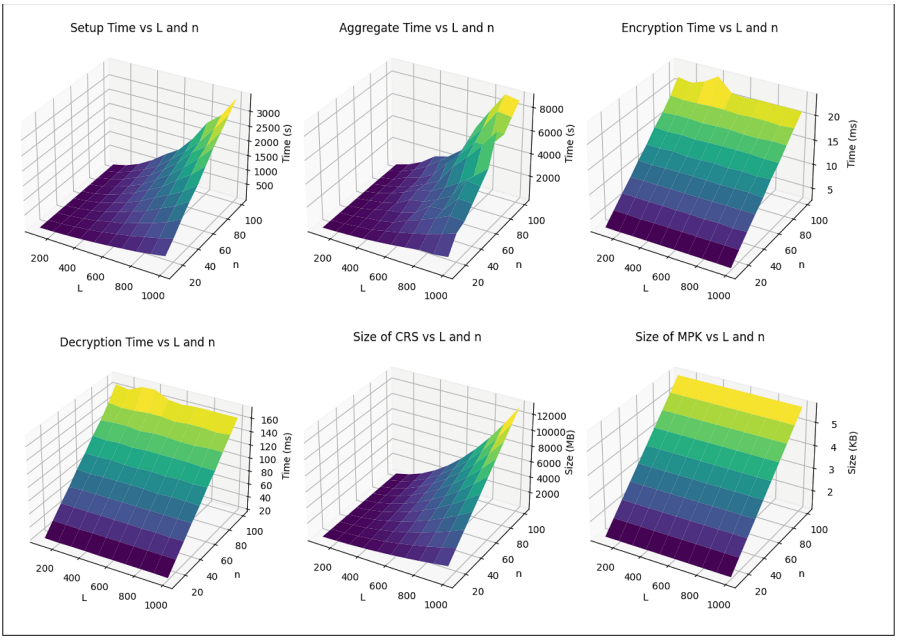

Fig. 2. Benchmarks for $L \in \{100, 200, \cdots, 1000\}$ and $n \in \{10, 20, \cdots, 100\}$

[8] https://anonymous.4open.science/r/slotted-ripe-DD12/.

the algorithms 100 times for each pair (L, n), and then computed the average runtime. The setup and aggregate were run once for each unique pair of (L, n). We did not plot the sizes of the ciphertexts, but these can be determined deterministically based on n as $|c| = 580 + 49n$ bytes. The size of the helper secret key for each user is $|\mathsf{hsk}| = 340 + 97n$ bytes. Note that the setup and aggregation time grows acutely with L and n. Improving the efficiency of our sRIPE scheme is left as a future work.

Acknowledgments. The authors thank the anonymous reviewers for their helpful comments. The first author was supported by the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM). The second and the sixth author were partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU and by Sapienza University under the project SPECTRA. The third author was partially supported by the European Union (ERC AdG REWORC - 101054911), and by True Data 8 (Distributed Ledger & Multiparty Computation) under the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE. The fourth author was partially funded by the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038 and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

1. Agrawal, S.: Indistinguishability obfuscation without multilinear maps: new methods for bootstrapping and instantiation. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11476, pp. 191–225. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17653-2_7
2. Agrawal, S., Kitagawa, F., Modi, A., Nishimaki, R., Yamada, S., Yamakawa, T.: Bounded functional encryption for turing machines: adaptive security from general assumptions. In: Kiltz, E., Vaikuntanathan, V. (eds.) Theory of Cryptography: 20th International Conference, TCC 2022, Chicago, IL, USA, 7–10 November 2022, Proceedings, Part I, pp. 618–647. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-22318-1_22
3. Agrawal, S., Maitra, M., Vempati, N.S., Yamada, S.: Functional encryption for turing machines with dynamic bounded collusion from LWE. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021. LNCS, vol. 12828, pp. 239–269. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84259-8_9
4. Agrawal, S., Yamada, S.: Optimal broadcast encryption from pairings and LWE. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12105, pp. 13–43. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45721-1_2
5. Ananth, P., Brakerski, Z., Segev, G., Vaikuntanathan, V.: From selective to adaptive security in functional encryption. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 657–677. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48000-7_32
6. Ananth, P., Jain, A., Lin, H., Matt, C., Sahai, A.: Indistinguishability obfuscation without multilinear maps: new paradigms via low degree weak pseudorandomness

and security amplification. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 284–332. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_10
7. Ananth, P., Jain, A.: Indistinguishability obfuscation from compact functional encryption. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9215, pp. 308–326. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47989-6_15
8. Ananth, P., Sahai, A.: Projective arithmetic functional encryption and indistinguishability obfuscation from degree-5 multilinear maps. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10210, pp. 152–181. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56620-7_6
9. Ananth, P., Vaikuntanathan, V.: Optimal bounded-collusion secure functional encryption. In: Hofheinz, D., Rosen, A. (eds.) TCC 2019. LNCS, vol. 11891, pp. 174–198. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-36030-6_8
10. Aranha, D.F., Gouvêa, C.P.L., Markmann, T., Wahby, R.S., Liao, K.: RELIC is an Efficient LIbrary for Cryptography (2020). https://github.com/relic-toolkit/relic
11. Attrapadung, N., Hanaoka, G., Yamada, S.: Conversions among several classes of predicate encryption and applications to ABE with various compactness tradeoffs. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 575–601. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48797-6_24
12. Baltico, C.E.Z., Catalano, D., Fiore, D., Gay, R.: Practical functional encryption for quadratic functions with applications to predicate encryption. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 67–98. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_3
13. Barak, B., et al.: On the (im) possibility of obfuscating programs. J. ACM (JACM) **59**(2), 1–48 (2012)
14. Barthe, G., Fagerholm, E., Fiore, D., Mitchell, J.C., Scedrov, A., Schmidt, B.: Automated analysis of cryptographic assumptions in generic group models. J. Cryptol. **32**(2), 324–360 (2019)
15. Bitansky, N.: Verifiable random functions from non-interactive witness-indistinguishable proofs. J. Cryptol. **33**(2), 459–493 (2020)
16. Bitansky, N., Vaikuntanathan, V.: Indistinguishability obfuscation from functional encryption. In: Guruswami, V. (ed.) 56th FOCS, pp. 171–190. IEEE Computer Society Press (2015)
17. Boneh, D., et al.: Fully key-homomorphic encryption, arithmetic circuit ABE and compact garbled circuits. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 533–556. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_30
18. Boneh, D., Sahai, A., Waters, B.: Functional encryption: definitions and challenges. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 253–273. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19571-6_16
19. Brakerski, Z., Döttling, N., Garg, S., Malavolta, G.: Candidate iO from homomorphic encryption schemes. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part I. LNCS, vol. 12105, pp. 79–109. Springer, Heidelberg (May (2020). https://doi.org/10.1007/s00145-023-09471-5
20. Brakerski, Z., Döttling, N., Garg, S., Malavolta, G.: Factoring and pairings are not necessary for io: circular-secure lwe suffices. In: 49th International Colloquium on Automata, Languages, and Programming (ICALP 2022). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2022)

21. Brakerski, Z., Segev, G.: Function-private functional encryption in the private-key setting. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015. LNCS, vol. 9015, pp. 306–324. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46497-7_12
22. Chotard, J., Dufour-Sans, E., Gay, R., Phan, D.H., Pointcheval, D.: Dynamic decentralized functional encryption. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12170, pp. 747–775. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56784-2_25
23. Cong, K., Eldefrawy, K., Smart, N.P.: Optimizing registration based encryption. In: Paterson, M.B. (ed.) IMACC 2021. LNCS, vol. 13129, pp. 129–157. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92641-0_7
24. Döttling, N., Kolonelos, D., Lai, R.W.F., Lin, C., Malavolta, G., Rahimi, A.: Efficient laconic cryptography from learning with errors. In: Hazay, C., Stam, M. (eds.) EUROCRYPT 2023. LNCS, vol. 14006, pp. 417–446. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30620-4_14
25. Francati, D., Friolo, D., Maitra, M., Malavolta, G., Rahimi, A., Venturi, D.: Registered (inner-product) functional encryption. Cryptology ePrint Archive (2023)
26. Francati, D., Friolo, D., Malavolta, G., Venturi, D.: Multi-key and multi-input predicate encryption from learning with errors. In: Advances in Cryptology-EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, 23–27 April 2023, Proceedings, Part III, pp. 573–604. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-30620-4_19
27. Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. In: 54th FOCS, pp. 40–49. IEEE Computer Society Press (2013)
28. Garg, S., Gentry, C., Halevi, S., Zhandry, M.: Functional encryption without obfuscation. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9563, pp. 480–511. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49099-0_18
29. Garg, S., Hajiabadi, M., Mahmoody, M., Rahimi, A.: Registration-based encryption: removing private-key generator from IBE. In: Beimel, A., Dziembowski, S. (eds.) TCC 2018. LNCS, vol. 11239, pp. 689–718. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03807-6_25
30. Garg, S., Hajiabadi, M., Mahmoody, M., Rahimi, A., Sekar, S.: Registration-based encryption from standard assumptions. In: Lin, D., Sako, K. (eds.) PKC 2019. LNCS, vol. 11443, pp. 63–93. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17259-6_3
31. Garg, S., Pandey, O., Srinivasan, A., Zhandry, M.: Breaking the sub-exponential barrier in obfustopia. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10212, pp. 156–181. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56617-7_6
32. Gay, R., Jain, A., Lin, H., Sahai, A.: Indistinguishability obfuscation from simple-to-state hard problems: new assumptions, new techniques, and simplification. In: Canteaut, A., Standaert, F.-X. (eds.) EUROCRYPT 2021. LNCS, vol. 12698, pp. 97–126. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77883-5_4
33. Gay, R., Pass, R.: Indistinguishability obfuscation from circular security. In: Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing, pp. 736–749 (2021)
34. Gentry, C., Lewko, A.B., Sahai, A., Waters, B.: Indistinguishability obfuscation from the multilinear subgroup elimination assumption. In: Guruswami, V. (ed.) 56th FOCS, pp. 151–170. IEEE Computer Society Press (2015)

35. Glaeser, N., Kolonelos, D., Malavolta, G., Rahimi, A.: Efficient registration-based encryption. Cryptology ePrint Archive, Paper 2022/1505 (2022). https://eprint.iacr.org/2022/1505
36. Goldwasser, S., Kalai, Y.T., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: Reusable garbled circuits and succinct functional encryption. In: Boneh, D., Roughgarden, T., Feigenbaum, J. (eds.) 45th ACM STOC, pp. 555–564. ACM Press (2013)
37. Gorbunov, S., Vaikuntanathan, V., Wee, H.: Functional encryption with bounded collusions via multi-party computation. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 162–179. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_11
38. Goyal, R., Hohenberger, S., Koppula, V., Waters, B.: A generic approach to constructing and proving verifiable random functions. In: Kalai, Y., Reyzin, L. (eds.) TCC 2017. LNCS, vol. 10678, pp. 537–566. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70503-3_18
39. Goyal, R., Vusirikala, S.: Verifiable registration-based encryption. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12170, pp. 621–651. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56784-2_21
40. Hemenway, B., Jafargholi, Z., Ostrovsky, R., Scafuro, A., Wichs, D.: Adaptively secure garbled circuits from one-way functions. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9816, pp. 149–178. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53015-3_6
41. Hohenberger, S., Lu, G., Waters, B., Wu, D.J.: Registered attribute-based encryption. Cryptology ePrint Archive (2022)
42. Hubacek, P., Wichs, D.: On the communication complexity of secure function evaluation with long output. In: Roughgarden, T. (ed.) ITCS 2015, pp. 163–172. ACM (2015)
43. Jain, A., Lin, H., Matt, C., Sahai, A.: How to leverage hardness of constant-degree expanding polynomials overa $\mathbb{R}$ to build $i\mathcal{O}$. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11476, pp. 251–281. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17653-2_9
44. Jain, A., Lin, H., Sahai, A.: Indistinguishability obfuscation from well-founded assumptions. In: Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing, pp. 60–73 (2021)
45. Jain, A., Lin, H., Sahai, A.: Indistinguishability obfuscation from LPN over $\mathbb{F}_p$, DLIN, and PRGs in NC^0. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part I. LNCS, vol. 13275, pp. 670–699. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-06944-4_23
46. Katz, J., Sahai, A., Waters, B.: Predicate encryption supporting disjunctions, polynomial equations, and inner products. In: Smart, N. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 146–162. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78967-3_9
47. Laurent Girod, W.L.: Petrelic is a python wrapper around relic (2022). https://github.com/spring-epfl/petrelic
48. Lin, H.: Indistinguishability obfuscation from constant-degree graded encoding schemes. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9665, pp. 28–57. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49890-3_2
49. Mahmoody, M., Qi, W., Rahimi, A.: Lower bounds for the number of decryption updates in registration-based encryption. Cryptology ePrint Archive, Report 2022/1285 (2022). https://eprint.iacr.org/2022/1285

50. Okamoto, T., Pietrzak, K., Waters, B., Wichs, D.: New realizations of somewhere statistically binding hashing and positional accumulators. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 121–145. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48797-6_6
51. O'Neill, A.: Definitional issues in functional encryption. Cryptology ePrint Archive, Report 2010/556 (2010). https://eprint.iacr.org/2010/556
52. Sahai, A., Waters, B.: How to use indistinguishability obfuscation: deniable encryption, and more. In: Shmoys, D.B. (ed.) 46th ACM STOC, pp. 475–484. ACM Press (2014)
53. Sahai, A., Waters, B.: Fuzzy identity-based encryption. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 457–473. Springer, Heidelberg (2005). https://doi.org/10.1007/11426639_27
54. Shamir, A.: Identity-based cryptosystems and signature schemes. In: Blakley, G.R., Chaum, D. (eds.) CRYPTO 1984. LNCS, vol. 196, pp. 47–53. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-39568-7_5
55. Shen, E., Shi, E., Waters, B.: Predicate privacy in encryption systems. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 457–473. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00457-5_27
56. Wee, H., Wichs, D.: Candidate obfuscation via oblivious LWE sampling. In: Canteaut, A., Standaert, F.-X. (eds.) EUROCRYPT 2021. LNCS, vol. 12698, pp. 127–156. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77883-5_5