Traitor Tracing without Trusted Authority from Registered Functional Encryption

Pedro Branco[1], Russell W. F. Lai[2], Monosij Maitra[3],

Giulio Malavolta[1], Ahmadreza Rahimi[4], Ivy K. Y. Woo[2]*

[1] Bocconi University
[2] Aalto University
[3] IIT Kharagpur
[4] Independent

*Slides made partly by Ivy K. Y. Woo

Scenario: Group Messaging

- L users wish to broadcast messages to each other privately, such that:
  - Small ciphertext, e.g. sublinear in L [Efficiency]
  - No information about any message exchanged is revealed [CPA-Security]
  - Trace a user that leaked its own secret key (e.g. device compromised) [Trace]
    → Allows to exclude traitor from the group

Scenario: Group Messaging

- L users wish to broadcast messages to each other privately, such that:
  - Small ciphertext, e.g. sublinear in L [Efficiency]
  - No information about any message exchanged is revealed [CPA-Security]
  - Trace a user that leaked its own secret key (e.g. device compromised) [Trace]
    → Allows to exclude traitor from the group
- Desired primitive: Traitor Tracing [CFN94]

Traitor Tracing

- Setting:
  - Authority: Generates public parameters (or master public key) + all users' secret keys
  - Encryptor: Encrypts w.r.t. master public key to all users
- Correctness: Any user with secret key can decrypt.

Traitor Tracing

- Setting:
  - Authority: Generates public parameters (or master public key) + all users' secret keys
  - Encryptor: Encrypts w.r.t. master public key to all users
- Correctness: Any user with secret key can decrypt.
- Security:
  - Encrypted message remains hidden without secret key
  - Trace Algorithm: Given a device that can decrypt, determines (at least one) corrupt user

Traitor Tracing

- Setting:
    - Authority: Generates public parameters (or master public key) + all users' secret keys
    - Encryptor: Encrypts w.r.t. master public key to all users
- Correctness: Any user with secret key can decrypt.
- Security:
    - Encrypted message remains hidden without secret key
    - Trace Algorithm: Given a device that can decrypt, determines (at least one) corrupt user
- Traitor Tracing [CFN94]: Long line of works on improving efficiency (e.g. [BSW06, BW06, . . . , GLW23, AKYY23])

Traitor Tracing

- Setting:
  - Authority: Generates public parameters (or master public key) + all users' secret keys
  - Encryptor: Encrypts w.r.t. master public key to all users
- Correctness: Any user with secret key can decrypt.
- Security:
  - Encrypted message remains hidden without secret key
  - Trace Algorithm: Given a device that can decrypt, determines (at least one) corrupt user
- Traitor Tracing [CFN94]: Long line of works on improving efficiency (e.g. [BSW06, BW06, . . . , GLW23, AKYY23])
- Key escrow problem: No security if authority is corrupt

Motivation

- This work:

Efficient traitor-tracing *without* a trusted authority

- Goals:
  - Remove trusted authority
  - Non-trivial, concrete efficiency (Ciphertext grows sublinear in number of users)
  - Security from simple and well-understood objects (e.g. not iO)

Our Contributions

- New model *without* Trusted Authority: Registered Traitor Tracing (RTT)

Our Contributions

- New model *without* Trusted Authority: Registered Traitor Tracing (RTT)
- Transformation: Registered Functional Encryption (RFE) → RTT

Our Contributions

- New model *without* Trusted Authority: Registered Traitor Tracing (RTT)
- Transformation: Registered Functional Encryption (RFE) → RTT
- Registered Quadratic Functional Encryption (RQFE)
  - Weak RQFE with *transparent* Setup in GGM
  - → RTT with unbounded collusion

Our Contributions

- New model *without* Trusted Authority: Registered Traitor Tracing (RTT)
- Transformation: Registered Functional Encryption (RFE) → RTT
- Registered Quadratic Functional Encryption (RQFE)
    - Weak RQFE with *transparent* Setup in GGM
    - → RTT with unbounded collusion
- Registered Linear Functional Encryption (RLFE)
    - RLFE in standard model (assumption proven in GGM)
    - → RTT for bounded collusion

Our Contributions

- New model *without* Trusted Authority: Registered Traitor Tracing (RTT)
- Transformation: Registered Functional Encryption (RFE) → RTT
- Registered Quadratic Functional Encryption (RQFE)
    - Weak RQFE with *transparent* Setup in GGM
    - → RTT with unbounded collusion
- Registered Linear Functional Encryption (RLFE)
    - RLFE in standard model (assumption proven in GGM)
    - → RTT for bounded collusion
- Prototype implementation for our RTT (RPLBE)

Our Contributions

- New model *without* Trusted Authority: Registered Traitor Tracing (RTT)
- Transformation: Registered Functional Encryption (RFE) → RTT
- Registered Quadratic Functional Encryption (RQFE)
  - Weak RQFE with *transparent* Setup in GGM
  - → RTT with unbounded collusion
- Registered Linear Functional Encryption (RLFE)
  - RLFE in standard model (assumption proven in GGM)
  - → RTT for bounded collusion
- Prototype implementation for our RTT (RPLBE)
- More applications of our RFEs

Related Works: Prior + Concurrent

- All prior schemes require trusted authority, except [Luo22]:
    - [Luo22]: No Setup + Relies on iO + Non-compact master public key + Non-deterministic decryption

Related Works: Prior + Concurrent

- All prior schemes require trusted authority, except [Luo22]:
  - [Luo22]: No Setup + Relies on iO + Non-compact master public key + Non-deterministic decryption
- Our framework is inspired from:
  - Registration-based Encryption [GHMR18, GHM+19]
  - Removes trusted authority in IBE

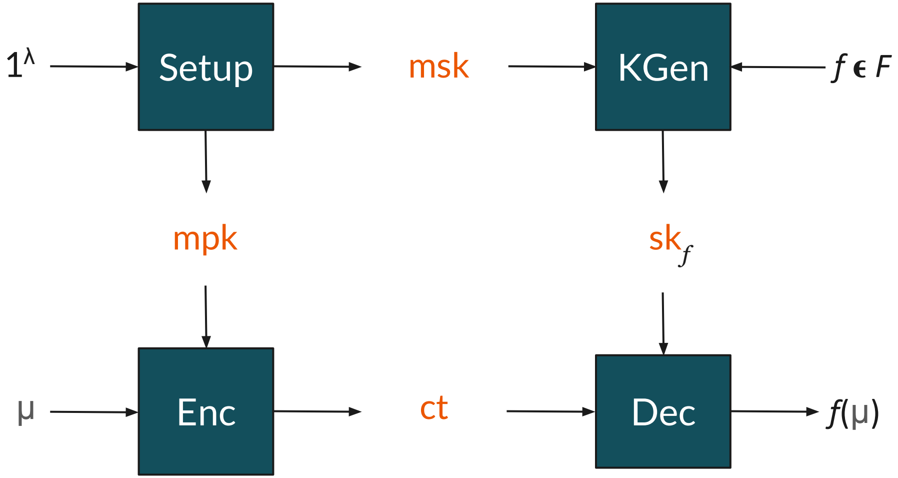
Related Works: Prior + Concurrent

- All prior schemes require trusted authority, except [Luo22]:
  - [Luo22]: No Setup + Relies on iO + Non-compact master public key + Non-deterministic decryption
- Our framework is inspired from:
  - Registration-based Encryption [GHMR18, GHM+19]
  - Removes trusted authority in IBE
- Concurrent works on RFE:
  - [DPY23] gets RLFE (with *non-transparent* setup) in GGM
  - [ZLZ+24] gets (very selective) RQFE and RLFE from variants of k-Lin, with *non-transparent* setup

Functional Encryption

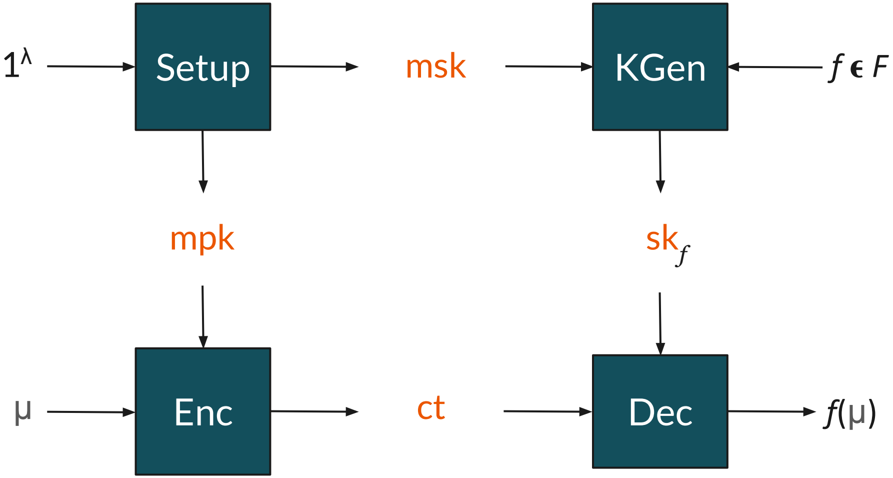
Functional Encryption

Traditionally



Functional Encryption

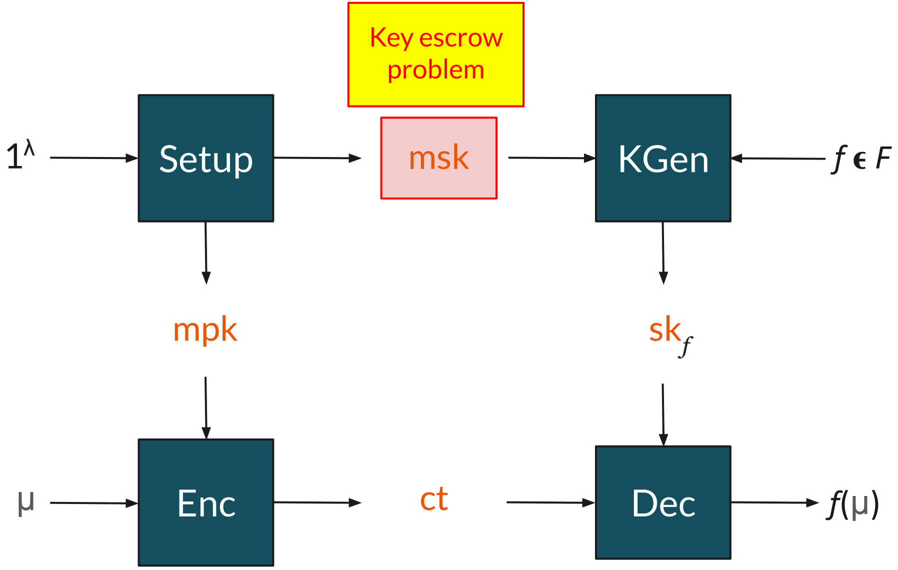
Traditionally



Security (Informally) : $\left[\, \mathrm{mpk}\,, \{\, \mathrm{sk}_f \,\}, \mathrm{ct}(\mu_0)\,, \{\, \mathrm{sk}_f \,\} \,\right] \approx \left[\, \mathrm{mpk}\,, \{\, \mathrm{sk}_f \,\}, \mathrm{ct}(\mu_1)\,, \{\, \mathrm{sk}_f \,\} \,\right]$ provided $f(\mu_0) = f(\mu_1)$

Functional Encryption

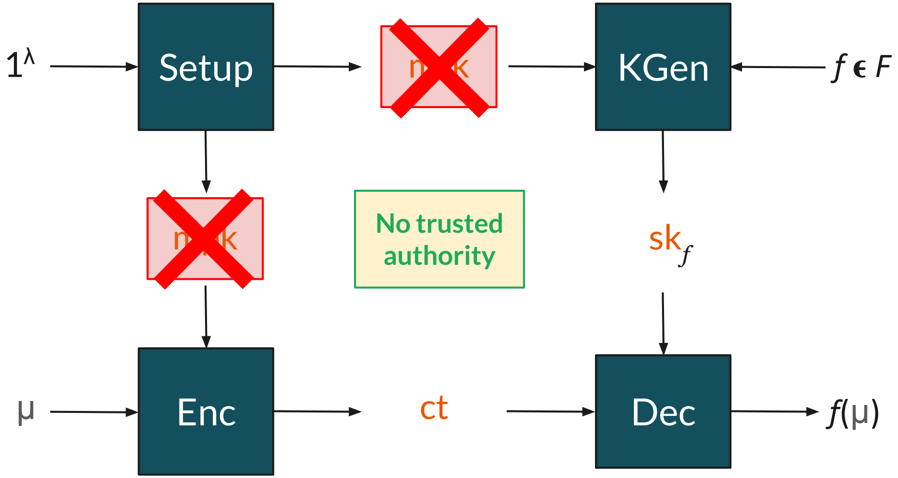
Traditionally



No security if msk gets leaked

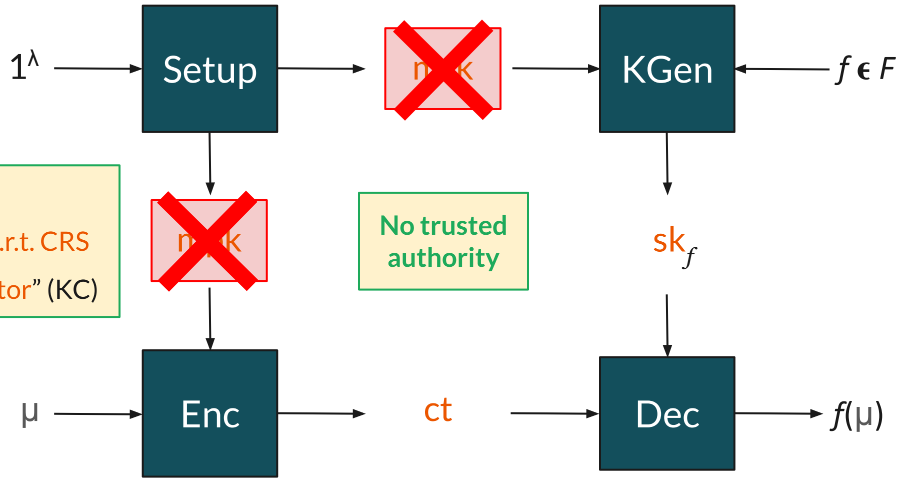
Registered Functional Encryption

Formalized in [FFM⁺23, DP23]



Registered Functional Encryption

Formalized in [FFM$^+$23, DP23]



No trusted authority

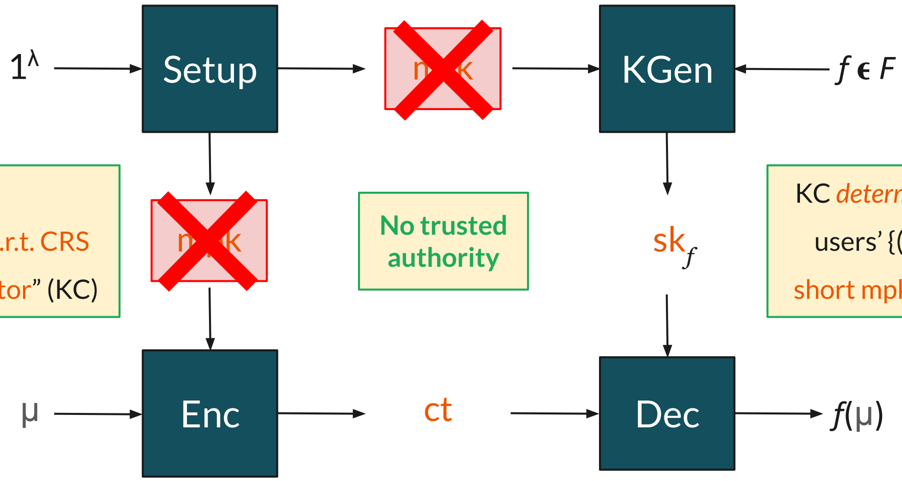
Setup computes a CRS

User samples its own key-pair (pk, sk) w.r.t. CRS

Users "registers" (pk, *f*) with a "Key Curator" (KC)

Registered Functional Encryption

Formalized in [FFM$^+$23, DP23]



Setup computes a CRS

User samples its own key-pair (pk, sk) w.r.t. CRS

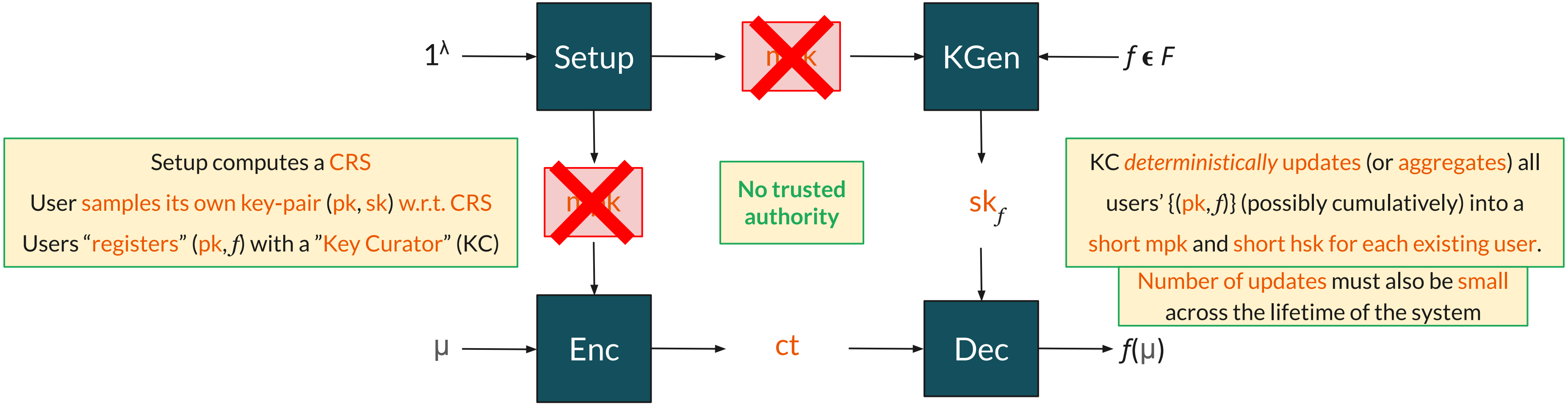
Users “registers” (pk, *f*) with a ”Key Curator” (KC)

No trusted authority

KC *deterministically* updates (or aggregates) all users’ {(pk, *f*)} (possibly cumulatively) into a short mpk and short hsk for each existing user.

Registered Functional Encryption

Formalized in [FFM$^+$23, DP23]



Setup computes a CRS

User samples its own key-pair (pk, sk) w.r.t. CRS

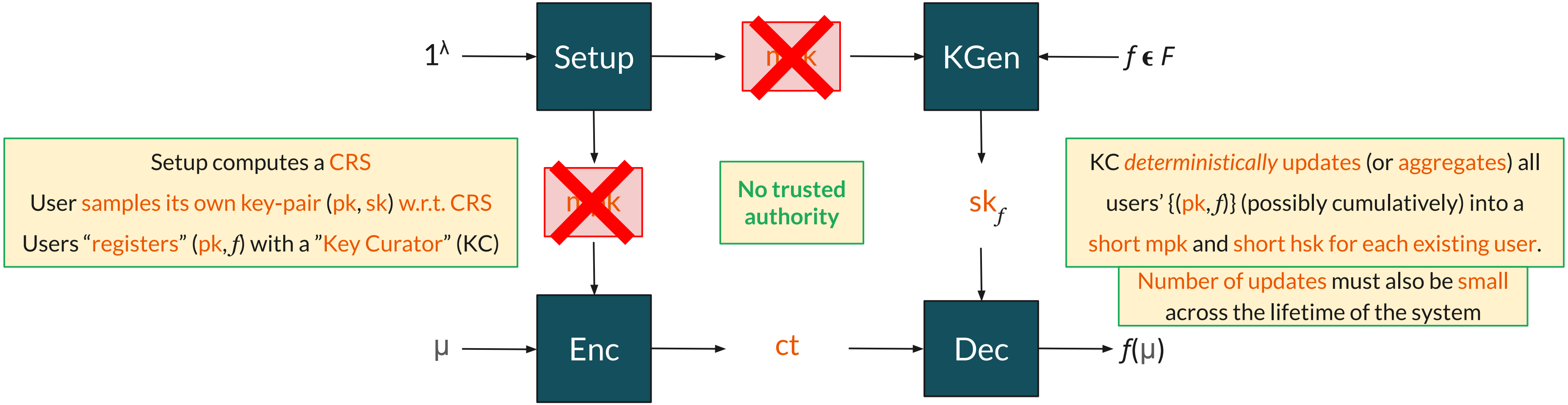
Users “registers” (pk, *f*) with a ”Key Curator” (KC)

KC *deterministically* updates (or aggregates) all users’ {(pk, *f*)} (possibly cumulatively) into a short mpk and short hsk for each existing user.

Number of updates must also be small across the lifetime of the system

Registered Functional Encryption

Formalized in [FFM+23, DP23]



Setup computes a CRS

User samples its own key-pair (pk, sk) w.r.t. CRS

Users "registers" (pk, *f*) with a "Key Curator" (KC)

KC *deterministically* updates (or aggregates) all users' {(pk, *f*)} (possibly cumulatively) into a short mpk and short hsk for each existing user.

Number of updates must also be small across the lifetime of the system

Security (Informally) : Similar to FE , except now registered keys can be generated maliciously

(Slotted) Registered Functional Encryption

(Slotted) Registered Functional Encryption

- crs ← Setup(1^λ, L): Generate common reference string

(Slotted) Registered Functional Encryption

- $\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda, L)$: Generate common reference string
- $(\mathsf{pk}_\ell, \mathsf{sk}_\ell) \leftarrow \mathsf{KGen}(\mathsf{crs}, \ell \in [L])$: User ℓ self-generates its public-secret key-pair

(Slotted) Registered Functional Encryption

- crs ← Setup(1^λ, L): Generate common reference string
- (pk_ℓ, sk_ℓ) ← KGen(crs, $\ell \in [L]$): User ℓ self-generates its public-secret key-pair
- (mpk, $(\mathsf{hsk}_\ell)_{\ell \in [L]}$) ← Aggr(crs, $(\mathsf{pk}_\ell, f_\ell)_{\ell \in [L]}$): Given pk_ℓ and function f_ℓ for ℓ-th user,
  - Aggregate users into system + generate helper secret key
  - Public, deterministic; mpk and each hsk_ℓ should be short, i.e., poly(λ , log L)

(Slotted) Registered Functional Encryption

- crs ← Setup(1^λ, L): Generate common reference string
- (pk_ℓ, sk_ℓ) ← KGen(crs, $\ell \in [L]$): User ℓ self-generates its public-secret key-pair
- (mpk, $(\mathrm{hsk}_\ell)_{\ell \in [L]}$) ← Aggr(crs, $(\mathrm{pk}_\ell, f_\ell)_{\ell \in [L]}$): Given pk_ℓ and function f_ℓ for ℓ-th user,
    - Aggregate users into system + generate helper secret key
    - Public, deterministic; mpk and each hsk_ℓ should be short, i.e., poly(λ , log L)
- ct ← Enc(mpk, μ): Encrypt message μ w.r.t. mpk

(Slotted) Registered Functional Encryption

- crs ← Setup(1^λ, L): Generate common reference string
- (pk_ℓ, sk_ℓ) ← KGen(crs, $\ell \in [L]$): User ℓ self-generates its public-secret key-pair
- (mpk, $(\mathsf{hsk}_\ell)_{\ell \in [L]}$) ← Aggr(crs, $(\mathsf{pk}_\ell, f_\ell)_{\ell \in [L]}$): Given pk_ℓ and function f_ℓ for ℓ-th user,
  - Aggregate users into system + generate helper secret key
  - Public, deterministic; mpk and each hsk_ℓ should be short, i.e., poly(λ , log L)
- ct ← Enc(mpk, μ): Encrypt message μ w.r.t. mpk
- μ′ ← Dec(sk_ℓ, hsk_ℓ, ct): User ℓ decrypts using its own secret key + helper secret key

(Slotted) Registered Functional Encryption

- crs ← Setup(1^λ, L): Generate common reference string
- (pk_ℓ, sk_ℓ) ← KGen(crs, $\ell \in [L]$): User ℓ self-generates its public-secret key-pair
- (mpk, $(hsk_\ell)_{\ell \in [L]}$) ← Aggr(crs, $(pk_\ell, f_\ell)_{\ell \in [L]}$): Given pk_ℓ and function f_ℓ for ℓ-th user,
  - Aggregate users into system + generate helper secret key
  - Public, deterministic; mpk and each hsk_ℓ should be short, i.e., poly(λ , log L)
- ct ← Enc(mpk, μ): Encrypt message μ w.r.t. mpk
- μ′ ← Dec(sk_ℓ, hsk_ℓ, ct): User ℓ decrypts using its own secret key + helper secret key

Weak RFE: All functions $(f_\ell)_{\ell \in [L]}$ are known/fixed at (and input to) Setup

(Slotted) Registered Functional Encryption

- crs ← Setup(1^λ, L): Generate common reference string
- (pk_ℓ, sk_ℓ) ← KGen(crs, $\ell \in [L]$): User ℓ self-generates its public-secret key-pair
- (mpk, $(\mathsf{hsk}_\ell)_{\ell \in [L]}$) ← Aggr(crs, $(\mathsf{pk}_\ell, f_\ell)_{\ell \in [L]}$): Given pk_ℓ and function f_ℓ for ℓ-th user,
  - Aggregate users into system + generate helper secret key
  - Public, deterministic; mpk and each hsk_ℓ should be short, i.e., poly(λ , log L)
- ct ← Enc(mpk, μ): Encrypt message μ w.r.t. mpk

  |ct| = poly(λ , log L)
- μ′ ← Dec(sk_ℓ, hsk_ℓ, ct): User ℓ decrypts using its own secret key + helper secret key

Weak RFE: All functions $(f_\ell)_{\ell \in [L]}$ are known/fixed at (and input to) Setup

Our RQFE: Construction Idea

- Compile: traditional QFE to RQFE, via “master secret key homomorphism”

Our RQFE: Construction Idea

- Compile: traditional QFE to RQFE, via “master secret key homomorphism”

- Linearly-homomorphic Encode function acting on QFE’s msk

$$\underbrace{\mathsf{Encode}(\mathsf{msk}_0)}_{\mathsf{mpk}_0} * \underbrace{\mathsf{Encode}(\mathsf{msk}_1)}_{\mathsf{mpk}_1} = \mathsf{Encode}(\mathsf{msk}_0 + \mathsf{msk}_1)$$

Independent QFE instances can be publicly combined into a global QFE instance.

$*$ = Group operation

Our RQFE: Construction Idea

- Compile: traditional QFE to RQFE, via "master secret key homomorphism"

- Linearly-homomorphic Encode function acting on QFE's msk

$$\underbrace{\mathsf{Encode}(\mathsf{msk}_0)}_{\mathsf{mpk}_0} * \underbrace{\mathsf{Encode}(\mathsf{msk}_1)}_{\mathsf{mpk}_1} = \mathsf{Encode}(\mathsf{msk}_0 + \mathsf{msk}_1)$$

Independent QFE instances can be publicly combined into a global QFE instance.

- Linearly-homomorphic KGen for function *f*

$$\underbrace{\mathsf{KGen}(\mathsf{msk}_0, f)}_{\mathsf{sk}_f^{(0)}} * \underbrace{\mathsf{KGen}(\mathsf{msk}_1, f)}_{\mathsf{sk}_f^{(1)}} = \mathsf{KGen}(\mathsf{msk}_0 + \mathsf{msk}_1, f)$$

$*$ = Group operation

Secret keys from independent instances can be publicly combined into a secret key for the global instance.

Our RQFE: Construction Idea

- Given “master secret key homomorphic” QFE, define RQFE “global” master public key:

$$\widetilde{\mathsf{mpk}} = \mathsf{mpk}_1 * \ldots * \mathsf{mpk}_L = \mathsf{Encode}(\mathsf{msk}_1 + \ldots + \mathsf{msk}_L)$$

Our RQFE: Construction Idea

- Given “master secret key homomorphic” QFE, define RQFE “global” master public key:

$$\widetilde{\mathsf{mpk}} = \mathsf{mpk}_1 * \ldots * \mathsf{mpk}_L = \mathsf{Encode}(\mathsf{msk}_1 + \ldots + \mathsf{msk}_L)$$

- Publicly computable using mpk_ℓ from each user

Our RQFE: Construction Idea

- Given “master secret key homomorphic” QFE, define RQFE “global” master public key:

$$\widetilde{\mathsf{mpk}} = \mathsf{mpk}_1 * \ldots * \mathsf{mpk}_L = \mathsf{Encode}(\mathsf{msk}_1 + \ldots + \mathsf{msk}_L)$$

- Publicly computable using mpk_ℓ from each user

- Interpretation: L users additive-secret-sharing msk of global mpk

Our RQFE: Construction Idea

- Given “master secret key homomorphic” QFE, define RQFE “global” master public key:

$$\widetilde{\mathsf{mpk}} = \mathsf{mpk}_1 * \ldots * \mathsf{mpk}_L = \mathsf{Encode}(\mathsf{msk}_1 + \ldots + \mathsf{msk}_L)$$

- Publicly computable using mpk_ℓ from each user

- Interpretation: L users additive-secret-sharing msk of global mpk

- Each user also publishes helper information to help others decrypt their own share

Our RQFE: Construction Idea

- Each user j provides helper secret keys for each user $i \neq j$ (rely on “weak” setting):

$$\left\{\mathsf{sk}_{f_i}^{(j)} = \mathsf{KGen}(\mathsf{msk}_j, f_i)\right\}_{i \neq j}$$

Each independent QFE instance provide secret keys for functions under all other instances.

Our RQFE: Construction Idea

- Each user j provides helper secret keys for each user $i \neq j$ (rely on “weak” setting):

$$\left\{ \mathsf{sk}_{f_i}^{(j)} = \mathsf{KGen}(\mathsf{msk}_j, f_i) \right\}_{i \neq j}$$

Each independent QFE instance provide secret keys for functions under all other instances.

- Apply homomorphic property to user keys:

$$\begin{pmatrix} \perp & \mathsf{sk}_{f_2}^{(1)} & \cdots & \mathsf{sk}_{f_L}^{(1)} \\ \mathsf{sk}_{f_1}^{(2)} & \perp & \cdots & \mathsf{sk}_{f_L}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{sk}_{f_1}^{(L)} & \mathsf{sk}_{f_2}^{(L)} & \cdots & \perp \end{pmatrix} \xrightarrow{*} \begin{pmatrix} \mathsf{KGen}(\sum_{j \neq 1} \mathsf{msk}_j, f_1) \\ \mathsf{KGen}(\sum_{j \neq 2} \mathsf{msk}_j, f_2) \\ \vdots \\ \mathsf{KGen}(\sum_{j \neq L} \mathsf{msk}_j, f_L) \end{pmatrix}$$

Such secret keys can be publicly combined under all other instances.

Our RQFE: Construction Idea

- Each user j provides helper secret keys for each user $i \neq j$ (rely on “weak” setting):

$$\left\{ \mathsf{sk}^{(j)}_{f_i} = \mathsf{KGen}(\mathsf{msk}_j, f_i) \right\}_{i \neq j}$$

Each independent QFE instance provide secret keys for functions under all other instances.

- Apply homomorphic property to user keys:

$$\begin{pmatrix} \perp & \mathsf{sk}^{(1)}_{f_2} & \cdots & \mathsf{sk}^{(1)}_{f_L} \\ \mathsf{sk}^{(2)}_{f_1} & \perp & \cdots & \mathsf{sk}^{(2)}_{f_L} \\ \vdots & \vdots & \ddots & \vdots \\ \mathsf{sk}^{(L)}_{f_1} & \mathsf{sk}^{(L)}_{f_2} & \cdots & \perp \end{pmatrix} \xrightarrow{*} \begin{pmatrix} \mathsf{KGen}(\sum_{j \neq 1} \mathsf{msk}_j, f_1) \\ \mathsf{KGen}(\sum_{j \neq 2} \mathsf{msk}_j, f_2) \\ \vdots \\ \mathsf{KGen}(\sum_{j \neq L} \mathsf{msk}_j, f_L) \end{pmatrix}$$

Such secret keys can be publicly combined under all other instances.

- Each user i misses msk for exactly the i-th function f_i, which is known to itself

$$\widetilde{\mathsf{sk}}_{f_i} = \mathsf{KGen}(\textstyle\sum_{j \neq i} \mathsf{msk}_j, f_i) * \mathsf{KGen}(\mathsf{msk}_i, f_i) = \mathsf{KGen}(\widetilde{\mathsf{msk}}, f_i)$$

Our RQFE: Construction Idea

- Instantiate prior template with “master secret key homomorphic” QFE
- Adopt (adaptively secure) QFE of Baltico et al. [BCFG17] (originally proven secure in GGM)

Our RQFE: Construction Idea

- Instantiate prior template with “master secret key homomorphic” QFE
- Adopt (adaptively secure) QFE of Baltico et al. [BCFG17] (originally proven secure in GGM)
- (Weak) RQFE security proven in simplified setting:

  Adversary provides randomness of maliciously generated keys

Our RQFE: Construction Idea

- Instantiate prior template with "master secret key homomorphic" QFE

- Adopt (adaptively secure) QFE of Baltico et al. [BCFG17] (originally proven secure in GGM)

- (Weak) RQFE security proven in simplified setting:

  Adversary provides randomness of maliciously generated keys

- We provide multiple transformations for security against maliciously registered keys
  - NIZK: prove well-formedness of keys
  - Leverage random oracle on our RQFE: Setup remains transparent
  - Modify RQFE scheme (without random oracle, loses transparent Setup)

Transformation: RFE → RTT

- Analogous to QFE to TT transformation in a prior work.
- Build Private Linear Broadcast Encryption: PLBE = Broadcast Encryption + Trace-Encrypt

Transformation: RFE → RTT

- Analogous to QFE to TT transformation in a prior work.
- Build Private Linear Broadcast Encryption: PLBE = Broadcast Encryption + Trace-Encrypt
- [BSW06] PLBE → Traitor Tracing

  [Gay16] PLBE = QFE with function

$$F_\ell(i, m) := \begin{cases} m & \text{if } i \leq \ell \\ 0 & \text{otherwise} \end{cases}$$

Transformation: RFE → RTT

- Analogous to QFE to TT transformation in a prior work.

- Build Private Linear Broadcast Encryption: PLBE = Broadcast Encryption + Trace-Encrypt

- [BSW06] PLBE → Traitor Tracing

  [Gay16] PLBE = QFE with function

$$F_\ell(i,m) := \begin{cases} m & \text{if } i \leq \ell \\ 0 & \text{otherwise} \end{cases}$$

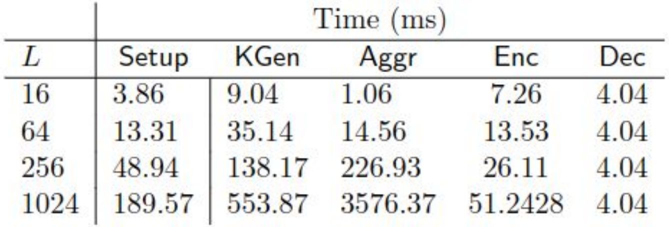
- We formalize the full chain of transformations in the registered setting

  Main observation: Weak RQFE suffices

Implementation: Registered PLBE

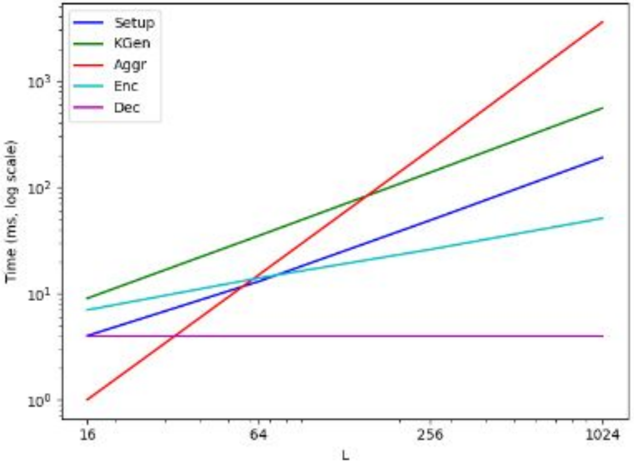
- Sizes for L = 1024:
  crs: 135KB, mpk: 6.6KB, ciphertext: 6.7KB
  pk: 102.5KB, sk: 97B, hsk: 194B

- Runtimes on PC:
  (AMD Ryzen 5 5600X, 3.7GHz CPU, 32GB of RAM)



	Time (ms)				
L	Setup	KGen	Aggr	Enc	Dec
16	3.86	9.04	1.06	7.26	4.04
64	13.31	35.14	14.56	13.53	4.04
256	48.94	138.17	226.93	26.11	4.04
1024	189.57	553.87	3576.37	51.2428	4.04

Table 4: Runtimes of our RPLBE algorithms for different L.



Other Application: Registered Threshold Encryption

- Registered Threshold Encryption (RTE):
  - Users sample their own (pk , sk) pairs & {pk} is aggregated into a short mpk.
  - Ciphertext grows with (*dynamically chosen*) threshold t.
  - System preserves threshold decryption (RTE generalizes distributed BE)

Other Application: Registered Threshold Encryption

- Registered Threshold Encryption (RTE):
  - Users sample their own (pk , sk) pairs & {pk} is aggregated into a short mpk.
  - Ciphertext grows with (*dynamically chosen*) threshold t.
  - System preserves threshold decryption (RTE generalizes distributed BE)

- RLFE → RTE: idea – Shamir's secret-sharing + linear function evaluation (in group exponent)

- t-out-of-L threshold:
  - User i runs RFE.KGen for a linear function $(1, i, \ldots, i^{t-1})$
  - Encrypt message μ: random degree t-1 polynomial P with P(0) = μ
  - RFE decryption : ensures user i learns P(i) (and nothing more)
  - Recover P(0) with t evaluation points $\{P(i) : i \in [t]\}$

Other Application: Registered Threshold Encryption

- Registered Threshold Encryption (RTE):
  - Users sample their own (pk , sk) pairs & {pk} is aggregated into a short mpk.
  - Ciphertext grows with (*dynamically chosen*) threshold t.
  - System preserves threshold decryption (RTE generalizes distributed BE)

- RLFE → RTE: idea – Shamir's secret-sharing + linear function evaluation (in group exponent)

- t-out-of-L threshold:
  - User i runs RFE.KGen for a linear function $(1, i, \ldots, i^{t-1})$
  - Encrypt message μ: random degree t-1 polynomial P with P(0) = μ
  - RFE decryption : ensures user i learns P(i) (and nothing more)
  - Recover P(0) with t evaluation points $\{P(i) : i \in [t]\}$

RLFE → RTE

RQFE → RTE
(*transparent setup*)

Summary

- New Model: Registered Traitor Tracing (RTT)
- Concretely efficient (weak) RQFE + Transformation to RTT with *transparent* setup
- Prototype implementation
- (More) Applications from our work:
    - RLFE → RTT with bounded collusion
    - RLFE → Single-key RFE for circuits
    - RTE from RLFE and RQFE (with *transparent* setup)

Summary

- New Model: Registered Traitor Tracing (RTT)
- Concretely efficient (weak) RQFE + Transformation to RTT with *transparent* setup
- Prototype implementation
- (More) Applications from our work:
    - RLFE → RTT with bounded collusion
    - RLFE → Single-key RFE for circuits
    - RTE from RLFE and RQFE (with *transparent* setup)

EPRINT: ia.cr/2024/179

Summary

- New Model: Registered Traitor Tracing (RTT)
- Concretely efficient (weak) RQFE + Transformation to RTT with *transparent* setup
- Prototype implementation
- (More) Applications from our work:
    - RLFE → RTT with bounded collusion
    - RLFE → Single-key RFE for circuits
    - RTE from RLFE and RQFE (with *transparent* setup)

EPRINT: ia.cr/2024/179

Thanks!