iUC-Secure Distributed File Transfer From Standard Attribute-based Encryption

Pascal Lafourcade[1][0000−0002−4459−511X], Gael Marcadet[1][0000−0003−1194−1343], and Léo Robert[2][0000−0002−9638−3143]

[1] Université Clermont-Auvergne, CNRS, Clermont-Auvergne-INP, LIMOS Clermont-Ferrand, France
[2] MIS, Université de Picardie Jules Verne, Amiens, France

Abstract. Attribute-Based Encryption (ABE) stands as a cryptographic cornerstone, enabling access control to messages based on user attributes. The security definition of standard ABE is shown to be impossible in Universal Composability (UC) against an *active* adversary. To overcome this issue, existing formal UC security definitions of ABE rely on additional properties for ABE, necessary to prove security against an active adversary, excluding standard ABE by definition. In light of the composability feature offered by UC and the absence of ideal functionality tailored for standard ABE, we propose the two following contributions: (1) We construct the first ideal functionality $\mathcal{F}_{\mathsf{ABE}}$ for ABE which, under reasonable hypothesis against static corruption, can be realized using an IND-CCA2-secure ABE scheme; and (2) our $\mathcal{F}_{\mathsf{ABE}}$ leads us to propose a protocol solving a simple yet highly practical, world-scaled company-focused problem: efficient file transfer. The proposed construction provides data integrity, sender authentication, attribute-based file access, featured with *constant* data size transferred between users. This is achieved by relying on two efficient building blocks: ABE and signature, which are layered atop of the hash-based distributed storage system IPFS. Our protocol, strengthened by a formal security definition and analysis under the Universally Composable (UC) framework called iUC, is proved to realize our problem-oriented authenticated attribute-based file transfer ideal functionality. Finally, we implement our proposal with a proof-of-concept written in Rust, and show it is practical and efficient.

Keywords: Universal Composability, Attribute-Based Encryption, Authenticated Attribute-based File Transfer

1 Introduction

Attribute-based encryption (ABE) is a fine-grained access encryption scheme in which a user securely shares a message to a group of users once, every user of this group being able to recover the encrypted message while every other users out of this group does not. Briefly, the formal definition of the aforementioned "group" is realized by associating to each user an attribute x and by adding an access

policy y to the ciphertext, the decryption procedure failing if the attribute x does not satisfy the access policy y. ABE has received lot of attention over the years to construct interesting primitive [?,?]. From a security standpoint, similarly to standard encryption schemes where indistinguishability holds only if the decryption key has not been corrupted, in ABE the indistinguishability for a ciphertext ψ holds only if the adversary does not have access to an attribute that can decrypt ψ. In game-based security, this is prevented by adding a winning condition preventing the adversary to decrypt the challenge ciphertext. Sadly, this mitigation cannot be transposed directly to the Universally Composable (UC) paradigm. Indeed in UC, traditional ideal functionalities for encryption aim to replace *all* plaintexts whose indistinguishability can be ensured by leakages. This is not suitable against *active* attribute corruption in which the adversary asks for the key-material associated to an attribute of its choice at *any time*. This issue has already been noticed by [?] in a closely related field of Role-Based Access Control (RBAC), where a user grants an access to some resources based on attributes. Security against *active* attribute corruption has already been achieved, for example by Camenisch *et al.* [?] using ABE equipped of an interactive decryption procedure between the user owning a ciphertext and a trusted third-party, owning decryption key for users. On one hand security against active adversary is achieved, but on the other hand, protocols using standard ABE have to integrate a more complex ABE primitive to fit in UC. This replacement is not always desirable, in particular for protocols whose efficiency is critical and may prefer standard ABE, even at the cost of a restricted security setting.

Contribution. We consider *standard* ABE to propose the first ideal functionality $\mathcal{F}_{\mathsf{ABE}}$ secure in the *static* attribute corruption setting. The protocol execution is divided into two distinct phases. The first phase, corresponding to a setup phase, consists for the adversary to instantiate any parties of its choice, with the possibility to corrupt them. During the second phase, the adversary is still allowed to instantiate parties but corruption of parties asking for decryption keys is no longer accepted. Then, assuming this constraint and an IND-CCA2-secure ABE scheme [3], we prove that our real protocol $\mathcal{P}_{\mathsf{ABE}}$ securely realize $\mathcal{F}_{\mathsf{ABE}}$. To increase usability, we have written $\mathcal{F}_{\mathsf{ABE}}$ and $\mathcal{P}_{\mathsf{ABE}}$ using the iUC framework, having the particularity to rely on the same formalism to express ideal, real and hybrid protocols. Based on the IITM model [?], this framework has been designed to be user-friendly, a welcomed feature to limit the complexity of reading and writing UC protocols.

To motivate the usability of our ideal functionality $\mathcal{F}_{\mathsf{ABE}}$, already strengthened by the easy-to-use iUC framework, we put it in practice to solve the file transfer problem in a large-scaled company. In particular, we construct a protocol allowing a user to share a document, let say, to all users working in a department. Attribute-based encryption is interesting in this setting, but confidentiality is not the only desired property. To increase confidence in our file transfer system, we add sender authentication and also integrity guarantee of

[3] An IND-CCA2-secure scheme can be efficiently derived from any IND-CPA-secure scheme via the Fujisaki-Okamoto transform [?].

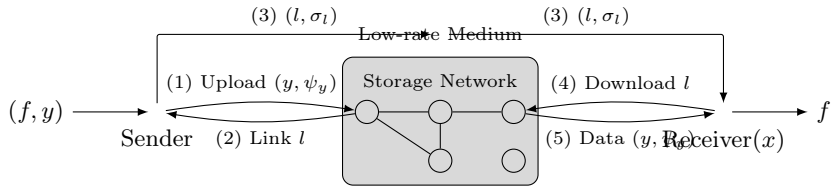

Fig. 1: Representation of our system where a sender shares a file f to a receiver.

the shared files. In addition to all these security properties, we dedicate our system to be particularly efficient in the case of large transferred files. Our study leads to the real protocol $\mathcal{P}_{\mathsf{AAFT}}$ a file transfer system ensuring the following three properties: (i) integrity of the transferred files, (ii) attribute-based file access, and (iii) explicit authentication of the sender. Our construction has the particularity to be constructed atop of a *distributed storage network*, a system composed of many servers whose general behavior is similar to a graph. A neat feature, compared to the single-server setting, is that it faces communication delay and workload issues. Inter-Planetary File System (IPFS) is an hash-based distributed storage system in which a server maintains a list of link-file pair (l, f) where l is the link of the file f, computed with a cryptographic hash function h as $l \leftarrow h(f)$. Later on, given the link, the server easily recovers and returns the file. We give an overview of our protocol in Fig. **??**, acting between a sender and a receiver. The sender obtains as an input a (potentially large) file f as well as some access policy y, and sends the file to every receiver having an attribute x satisfying y. As depicted in Fig. **??**, during step (1), the sender computes an encryption of f denoted ψ_y using attribute-based encryption where y is the access policy, and sends the couple (y, ψ_y) on a storage server of its choice. At step (2), the storage server responds with a link l computed as the hash of (y, ψ_y). At step (3), the sender computes σ_l the signature of l and sends the tuple (l, σ_l) through a limited communication medium, restricted to transmit data having length independent of the message. When the receiver obtains the link l and the associated signature σ_l, it obtains a proof of integrity and authenticates the sender simultaneously. At step (4) and (5), the receiver downloads the couple (y, ψ_y) using the link l, decrypts the result and checks authenticity of l using σ_l. This protocol is proved to securely realize our Authenticated Attribute-based File Transfer ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$, but also to be highly-practical, confirmed by our proof-of-concept fully-written in Rust and available at [**?**], sending up to 450 megabytes of data in 474 milliseconds.

Related Work. As explained above, the state-of-the-art for ABE in UC already proposes ideal functionality, but always equipped of an additional property or having a different design to guarantee security against active adversary. Abe and Ambrona [**?**] introduced an ideal functionality for ABE where the key generation is replaced by a blind key generation procedure including a non-interactive zero-knowledge proof. To obtain active security, Camenisch *et al.* [**?**] proposes to

rely on a trusted third party owning the decryption key of users. To decrypt a ciphertext, the ABE protocol is equipped of an interactive decryption procedure with the third party, which is not able to identify which ciphertext is being decrypted. These two works need an ABE scheme having either a blind key generation or interactive decryption procedure, excluding every standard efficient attribute-based encryption schemes such that [?]. To the best of our knowledge, there is no ideal functionality tailored for standard ABE.

We stress that our hybrid protocol $\mathcal{P}_{\mathsf{AAFT}}$, putting into application our ideal functionality $\mathcal{F}_{\mathsf{ABE}}$, constitutes a novel improvement in distributed file-transfer literature. Distributed file-transfer system has been introduced many years ago by Garay *et al.* [?], proposing a system based on a verifiable secret sharing, ensuring both confidentiality by distributing shares among the storage servers, as done in the more recent system called SAFE [?] with critical security and performance improvements. Due to the nature of secret sharing, confidentiality is only ensured with several storage servers and need to reconstruct the file. In contrast, by the confidentiality ensured by ABE, our system is still secure even with a (possibly corrupted) single storage server. Role-Based Access Control (RBAC) systems, a closely related topic, allows (or restricts) users to access some resources based on owned attribute. When a user accesses some content, it has first to be authenticated by an access-granting server. This is the case for the SESAME protocol [?], a RBAC based itself on Kerberos. The work of Freudenthal *et al.* [?], proposes to check the access permission of users with multiple trust authorities (*e.g.,* a public key infrastructure). The role-based access for distributed storage system is presented in [?]. They proposed a fix for the Object Store Devices specification [?] where unrestricted delegation is possible, in which confidentiality cannot be ensured. The proposed solution, elegantly modifies the original protocol by adding secure channels and signature, to enforce role-based access to the files, without modifying the specification. All of these papers differ from our contribution by the introduction of authorities in charge of granting an access to some content. Our work requires trusted authorities to handle public keys and to provide decryption keys, but are involved only during the initialisation, no authorization is required hereinafter. Introduced by Rizwan Ashgar *et al.* [?], ESPOON is a protocol working as RBAC but in outsourced environment with untrusted entities. Integrity is not ensured, whereas our work ensures the integrity and confidentiality of data in addition to sender authentication. The work of [?] proposes a solution between RBAC and a storage system. They formally defined a new security definition of RBAC, in the spirit of encryption indistinguishability. The adversary is asked to guess an encrypted message, and is assumed to have a full-control on a file system where the encrypted message is stored. It can also corrupt any user of its choice. To achieve the proposed definition, a new protocol is introduced relying on attribute-based encryption, as done hereinafter. Our work adds more features: we ensure data integrity and data authentication (thanks to hash-based distributed storage and signatures, respectively) in addition to data confidentiality brought by the attribute-based encryption. Universal Composability (UC) has already been

applied on the RBAC, initiated by Halevi *et al.* [**?**] proposing a UC model which requires at every communication a secure channel between the two parties, even if the entity is corrupted (but in this case, the secret key might be leaked), a standard assumption in UC. In our work, we have chosen to not consider any particular property on communication channels for two reasons. First, secrecy is not always possible for example with anonymous protocols, or even desirable for example when transferred data can be read in clear by the adversary. Second, authenticity is traditionally achieved using digital signature, that can also be used to sign messages in other protocols. Introducing an ideal functionality for digital signature is hence more appropriate. Even if it does not constitute an issue, their UC model is built on the original UC model of Canetti [**?**] in which session identifier prevents communication between sessions. In comparison, our protocol is proven under the iUC framework, where every entity is allowed to communicate with the others without restriction, a useful property for example with signature whose signing key are used in practice across multiple protocols.

Outline. In Section **??**, we briefly introduce all the necessary notions and terminology to understand our modelisation. In Section **??**, we present our ideal functionality $\mathcal{F}_{\mathsf{ABE}}$ and real protocol $\mathcal{P}_{\mathsf{ABE}}$ along the proof of realization. In Section **??**, we present the application of $\mathcal{F}_{\mathsf{ABE}}$ on the authenticated attribute-based file transfer with our ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$ along our hybrid protocol $\mathcal{P}_{\mathsf{AAFT}}$ and our proof-of-concept.

2 On iUC Framework

We provide an overview of the iUC framework. We refer interested readers looking for more details at the original paper [**?**]. A party pid involved in a protocol is traditionally equipped with session identifier sid, and acts in the protocol following a code specification called a role, and denoted role. The combination of the party identifier, the session identifier and the role constitutes the triplet $(\mathsf{pid}, \mathsf{sid}, \mathsf{role})$ and is called an *entity*. The existing role is specific to the designed protocol; for example a signature protocol consists of a role `signer` to sign messages and a role `verifier` to verify signed messages. The notion of entity is at the heart of the iUC framework, sharing similarities with object-oriented programming. In iUC, a *machine* denoted M_{role} implementing a role role corresponds to a class, both equipped with internal state used to store data. In a real protocol, a machine manages a single entity *i.e.,* represents a single party running in a single protocol execution. Yet, notice that a machine can be naturally extended to manage arbitrary number of entities, having different roles as well, the internal state being now used to share data across entities. For example, a signature ideal functionality benefits from this feature by adding authenticated messages in the internal state. In iUC, a machine, just like a class, can be instantiated several times, an instantiation being called an *instance*. Two important observations are to be made: First, the notion of entities and machines is sufficient to handle both real and ideal protocols. Second, a machine is not required to only handle

entities sharing the same sid but any entities, a particularly useful property to handle cross-protocols party such as certificate authority.

The iUC framework provides algorithms to describe behavior of instances *e.g.,* the number of accepted entities, the corruption model, the instance and entity initialization, and more. When an entity $(\mathsf{pid}, \mathsf{sid}, \mathsf{role})$ is contacted for the first time, the identity is submitted to every instances implementing the role role, until one instance accepts the entity, decided by the CheckID algorithm. If the instance does not have any accepted entity yet, it executes the Initialization algorithm to initialize its internal state. Once initialized, an entity executes the Main block containing the code to be executed by *honest* entities. Each role is associated with either a *public* or a *private* visibility. A public role is accessible to the environment, whereas a private role is limited to entities inside the protocol. An entity accepts requests coming from the environment via the input-output interface `I/O`, but also requests coming from the adversary via the network interface `NET`, possibly modelling interactions of an ideal functionality with the simulator. When needed, an entity may also accepts requests from more specific entities. The current running entity $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ is denoted $\mathsf{entity}_{\mathsf{cur}}$. An higher-protocol calling entity $(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$ is denoted $\mathsf{entity}_{\mathsf{call}}$. An instance has access to the set of managed entities that has been corrupted, denoted by CorruptionSet. An entity has to be considered if a subroutine has been corrupted, the corruption of an entity being verified by the corr algorithm returning `true` if the entity provided as an input has been corrupted, `false` otherwise. By $\mathsf{alg}^{(p)}$ we denote the execution of an algorithm alg whose the execution time is bounded by the polynomial p.

3 Standard Attribute-based Encryption Realization

Attribute-based Encryption, ABE for short, allows to broadcast a message to all users, whose only users having the read access with respect to an *access policy* associated to the message are able to read the message. We say that a user has a read access when it is associated to some *attribute*, say x, respecting the policy of the message, say y. In the paper, this statement is represented by $x \in y$. Briefly, an ABE scheme is defined by the tuple $(\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec})$. The Setup algorithm takes as an input the unary representation of the security parameter λ and it outputs a master key pair (msk, mpk). The encryption algorithm Enc expects as an input the master public key mpk, the access policy y and a message m, and it outputs the ciphertext ψ_y. To decrypt a message, one may previously asks to the authority owning the master key pair a decryption key denoted sk_x associated to some attribute x. This decryption key generation is handled by the KeyGen algorithm taking as input the master secret key msk as well as the attribute x and outputs the decryption key sk_x. This decryption key sk_x along a ciphertext ψ_x are provided to the decryption algorithm Dec, returning either the underlying plaintext m if and only if $x \in y$, or $\perp$ otherwise. An ABE scheme is said *correct* if for every master key pair $(msk, mpk) \leftarrow \mathsf{Setup}(1^\lambda)$, every ciphertext $\psi_y \leftarrow \mathsf{Enc}(mpk, y, m)$ for any message m and access policy y,

every decryption key $\mathsf{sk}_x \leftarrow \mathsf{KeyGen}(msk, x)$ for any attribute x with $x \in y$, we have $\Pr[\mathsf{Dec}(\mathsf{sk}_x, \psi_y) = m] = 1 - \epsilon$ for some negligible probability ϵ. In this work, we require an IND-CCA2-secure ABE which informally states that it must be infeasible to tell if a ciphertext ψ_y encrypts either the message m_0 or m_1 as long as no corrupted user has a secret key sk_x allowing to decrypt ψ_y. The security experiment is presented in Fig. **??**.

$\mathsf{Exp}^{\text{IND-CCA2}}_{\mathcal{A}}(\lambda)$

$\mathcal{X} \leftarrow \emptyset$
$(msk, mpk) \leftarrow \mathsf{Setup}(1^\lambda)$
$\mathcal{O} \leftarrow \{\mathsf{OKeyGen}(msk, \mathcal{X}, \bot; \cdot), \mathsf{ODec}(msk, \bot; \cdot, \cdot)\}$
$(y^*, m_0, m_1), \mathsf{state} \leftarrow \mathcal{A}_1^{\mathcal{O}}(mpk)$
$b \leftarrow\!\!\$\ \{0, 1\}$
$\psi_{y^*} \leftarrow \mathsf{Enc}(mpk, y^*, m_b)$
$\mathcal{O}' \leftarrow \{\mathsf{OKeyGen}(msk, \mathcal{X}, y^*; \cdot), \mathsf{ODec}(msk, \psi_{y^*}; \cdot, \cdot)\}$
$b' \leftarrow \mathcal{A}_2^{\mathcal{O}'}(\psi_{y^*}, \mathsf{state})$
return $b = b' \wedge (\nexists x \in \mathcal{X} : x \in y^*)$

Oracle $\mathsf{OKeyGen}(msk, \mathcal{X}, y^*; x)$

if $x \in y^*$: **return** $\bot$
$\mathcal{X} \leftarrow \mathcal{X} \cup \{x\}$
$\mathsf{sk}_x \leftarrow \mathsf{KeyGen}(msk, x)$
return sk_x

Oracle $\mathsf{ODec}(msk, \psi_{y^*}; \psi_y, x)$

if $\psi_{y^*} = \psi_y$: **return** $\bot$
$\mathsf{sk}_x \leftarrow \mathsf{KeyGen}(msk, x)$
$m \leftarrow \mathsf{Dec}(\mathsf{sk}_x, \psi_y)$
return m

Fig. 2: Experiment of the IND-CCA2 security for a ABE scheme.

Description of $\mathcal{F}_{\mathsf{ABE}}$ and $\mathcal{P}_{\mathsf{ABE}}$. The ideal functionality $\mathcal{F}_{\mathsf{ABE}}$, presented in Fig. **??**, proposes an instance managing several encryptors and decryptors as well as a *single* setup entity designated by the `setup` role. This setup entity, as its name suggests, deals with the setup algorithm and hence owns the master key pair, which has to remain private. Observe that using hierarchical session identifier property of iUC, an entity in $\mathcal{F}_{\mathsf{ABE}}$ and $\mathcal{P}_{\mathsf{ABE}}$ is supposed to have a session identifier sid the form $(\mathsf{pid}', \mathsf{sid}')$ with pid' the party identifier of the setup entity.

Recall that when encrypting a message, the set of corrupted attributes is required to be static *i.e.,* the environment is not allowed to dynamically obtain decryption key, otherwise is able to trivial distinguish by looking for ciphertext encrypting a leakage, so being impossible to prove secure as shown in [**?**]. Many scenarios are possible to obtain static corruption of attributes. We have chosen to divide the time in two distinct phases separated by a time $T \in \mathbb{N}$. During the first phase, when $t \leq T$, the environment is allowed to instantiate entities, corrupt them, but also to instantiate decryptors with attributes (thus to obtain decryption keys by corrupting decryptors as well). During the second phase, when $t > T$, the environment is no more allowed to obtain a decryption key of its choice. This modelisation is not unique, one may change hypothesis but is still required to prevent dynamic corruption of decryption keys and attributes.

The encryptor handles `Encrypt` requests used to encrypt a message. The security of the ABE ideal functionality states that for a given a master public

Ideal functionality $\mathcal{F}_{\mathsf{ABE}} = (\texttt{setup}, \texttt{encryptor}, \texttt{decryptor})$:

Participating roles: $\texttt{setup}, \texttt{encryptor}, \texttt{decryptor}$
Corruption model: static corruption
Protocol parameters:
- A polynomial $p \in \mathbb{Z}[x]$ used to bound the runtime execution of provided algorithms.
- A deterministic length-preserving leakage function L used to compute leakages.
- A time $T \in \mathbb{N}$ delimiting phase in which decryption keys are provided, from the phase where encryption and decryption are operated. We denote by $t \in \mathbb{N}$ the current time.

$M_{\texttt{setup},\texttt{encryptor},\texttt{decryptor}}$:

Implemented role(s): $\texttt{setup}, \texttt{encryptor}, \texttt{decryptor}$
Internal state:
- $\mathsf{msgList} \subset \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^*$ {*Set of encrypted messages*
- $\mathsf{keys} \subset (\{0,1\}^*)^3 \mapsto \{0,1\}^* \times \{0,1\}^*$
- $(mpk, \mathsf{Enc}, \mathsf{Dec}) \in (\{0,1\}^* \cup \bot)^3 = (\bot, \bot, \bot)$
- $\mathsf{pidsetup} \in \{0,1\}^* \cup \{\bot\} = \bot$
- $\mathsf{corrAttr} \subseteq \{0,1\}^* = \emptyset$

CheckID(pid, sid, role): Check that $\mathsf{sid} = (\mathsf{pid}', \mathsf{sid}')$, then accept every entity with the same SID, otherwise reject.

Corruption behavior:
- **AllowCorruption**(pid, sid, role): Returns `false` if role = `setup` or role = `decryptor` and $T < t$, otherwise returns `true`.

Initialization:
send responsively `InitABE` **to** `NET`
wait for $(\texttt{Init}, (mpk', Enc, Dec))$
$(mpk, \mathsf{Enc}, \mathsf{Dec}) \leftarrow mpk', Enc, Dec$
parse $\mathsf{sid}_{\mathsf{cur}}$ **as** (pid, sid)
$\mathsf{pidsetup} \leftarrow \mathsf{pid}$

Main:
recv $(\texttt{InitAttr}, x)$ **from** `I/O` **to** $(\_, \_, \texttt{decryptor})$ **s.t.** $\mathsf{keys}[\mathsf{entity}_{\mathsf{call}}] \neq \bot$:
  send responsively $(\texttt{InitReceiver}, x)$ **to** `NET`
  wait for $(\texttt{Init}, \mathsf{sk}_x)$
  for $(m, y, \psi_y) \in \mathsf{msgList}$:
    $m' \leftarrow \mathsf{Dec}^{(p)}(\mathsf{sk}_x, \psi_y)$
    if $(x \in y \wedge m' \neq L(\lambda, m)) \vee (x \notin y \wedge m' \neq \bot)$:
      send $(\texttt{Registered}, x, 0)$ **to** `NET` {*Decryption correctness failure*
  $\mathsf{keys}[\mathsf{entity}_{\mathsf{call}}] \leftarrow (x, \mathsf{sk}_x)$
  send $(\texttt{Registered}, x, 1)$ **to** `NET`

recv $(\texttt{CorrAttr}, x)$ **from** `NET` **s.t.** $t \leq T$:
  add x **to** corrAttr

recv `PubKey?` **from** `_` **to** $(\mathsf{pidsetup}, \_, \texttt{setup})$:
  reply $(\texttt{PubKey}, mpk)$

recv $(\texttt{Encrypt}, y, m, mpk')$ **from** `I/O` **to** $(\_, \_, \texttt{encryptor})$ **s.t.** $T < t$:
  if $mpk \neq mpk' \vee \exists x \in \mathsf{corrAttr}$ **s.t.** $x \in y$:
    $\psi_y \leftarrow \mathsf{Enc}^{(p)}(mpk', y, m)$
    reply $(\texttt{Ciphertext}, \psi_y)$
  $m' \leftarrow L(\lambda, m)$
  $\psi_y \leftarrow \mathsf{Enc}^{(p)}(mpk, y, m')$
  for $(\_, (x, \mathsf{sk}_x)) \in \mathsf{keys}$:
    if $(x \in y \wedge \mathsf{Dec}^{(p)}(\mathsf{sk}_x, \psi_y) \neq m') \vee (x \notin y \wedge \mathsf{Dec}^{(p)}(\mathsf{sk}_x, \psi_y) \neq \bot)$:
      reply $(\texttt{Ciphertext}, \bot)$ {*Encrytion correctness failure*
  add (m, y, ψ_y) **to** msgList
  reply $(\texttt{Ciphertext}, \psi_y)$

recv $(\texttt{Decrypt}, \psi_y)$ **from** `I/O` **to** $(\_, \_, \texttt{decryptor})$ **s.t.** $T < t \wedge \mathsf{keys}[\mathsf{entity}_{\mathsf{cur}}] \neq \bot$:
  $(x, \mathsf{sk}_x) \leftarrow \mathsf{keys}[\mathsf{entity}_{\mathsf{cur}}]$
  if $\nexists(\_, \_, \psi_y) \in \mathsf{msgList}$: **reply** $(\texttt{Plaintext}, \mathsf{Dec}^{(p)}(\mathsf{sk}_x, \psi_y))$
  if $\exists m, m'$ **s.t.** $(m, \_, \psi_y), (m', \_, \psi_y) \in \mathsf{msgList} \wedge m \neq m'$: **reply** $(\texttt{Plaintext}, \bot)$
  get(m, y, ψ_y) **from** msgList
  if $x \notin y$: **reply** $(\texttt{Plaintext}, \bot)$ {*Incompatible policy-access*
  reply $(\texttt{Plaintext}, m)$

Fig. 3: Description of our ideal functionality $\mathcal{F}_{\mathsf{ABE}}$.

Protocol $\mathcal{P}_{\mathsf{ABE}} = (\texttt{setup}, \texttt{encryptor}, \texttt{decryptor})$:

Participating roles: `setup`, `encryptor`, `decryptor`
Corruption model: static corruption
Protocol parameters:
- An IND-CCA2 attribute-based encryption scheme $\Pi = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$
- A time $T \in \mathbb{N}$ from which corruption and decryption keys are not allowed. We denote by $t \in \mathbb{N}$ the current time.

$M_{\texttt{setup}}$:

Implemented role(s): `setup`
Internal state:
- $mpk \in \{0,1\}^* \cup \{\bot\} = \bot$
- $msk \in \{0,1\}^* \cup \{\bot\} = \bot$
- $\mathsf{pidsetup} \in \{0,1\}^* \cup \{\bot\} = \bot$
- $\mathsf{keys} : (\{0,1\}^*)^3 \to \{0,1\}^*$

CheckID(pid, sid, role): Check that $\mathsf{sid} = (\mathsf{pid}', \mathsf{sid}')$. Accept a single entity.
Corruption behavior:
- **AllowCorruption**(pid, sid, role): return `false`

Initialization:
$(msk, mpk) \leftarrow \mathsf{Setup}(1^\lambda)$
parse $\mathsf{sid}_{\mathsf{cur}}$ **as** $(\mathsf{pid}, \mathsf{sid})$
$\mathsf{pidsetup} \leftarrow \mathsf{pid}$

Main:
recv $(\texttt{PubKey?})$ **from** $\_$ **to** $(\mathsf{pidsetup}, \_, \texttt{setup})$:
reply $(\texttt{PubKey}, mpk)$

recv $(\texttt{Register}, x)$ **from** $(\_, \_, \texttt{decryptor})$ **to** $(\mathsf{pidsetup}, \_, \texttt{setup})$:
if $\mathsf{keys}[\mathsf{entity}_{\mathsf{call}}] \neq \bot$: **reply** $(\texttt{Registered}, \bot)$
$\mathsf{sk}_x \leftarrow \mathsf{KeyGen}(msk, x)$
$\mathsf{keys}[\mathsf{entity}_{\mathsf{call}}] \leftarrow \mathsf{sk}_x$
reply $(\texttt{Registered}, \mathsf{sk}_x)$

$M_{\texttt{encryptor}}$:

Implemented role(s): `encryptor`
CheckID(pid, sid, role): Check that $\mathsf{sid} = (\mathsf{pid}', \mathsf{sid}')$. Accept a single entity.
Corruption behavior:
- **AllowAdvMessage**(pid, sid, role, $\mathsf{pid}_{\mathsf{recv}}$, $\mathsf{sid}_{\mathsf{recv}}$, $\mathsf{role}_{\mathsf{recv}}$, m): Check that $(\mathsf{pid} = \mathsf{pid}_{\mathsf{recv}})$. Otherwise, returns $\mathsf{role}_{\mathsf{recv}} \neq \texttt{setup}$ or m does not start with `Register`.

Main:
recv $(\texttt{Encrypt}, y, m, mpk)$ **from** `I/O` **s.t.** $T < t$:
$\psi_y \leftarrow \mathsf{Enc}(mpk, y, m)$
reply $(\texttt{Ciphertext}, \psi_y)$

$M_{\texttt{decryptor}}$:

Implemented role(s): `decryptor`
Internal state: $(x, \mathsf{sk}_x) \in \{0,1\}^* \times \{0,1\}^* = (\bot, \bot)$ {*Decryption key*
CheckID(pid, sid, role): Check that $\mathsf{sid} = (\mathsf{pid}', \mathsf{sid}')$. Accept a single entity.
Corruption behavior:
- **AllowAdvMessage**(pid, sid, role, $\mathsf{pid}_{\mathsf{recv}}$, $\mathsf{sid}_{\mathsf{recv}}$, $\mathsf{role}_{\mathsf{recv}}$, m): If $\mathsf{role}_{\mathsf{recv}} = \texttt{setup}$ and m starts with `Register` and $T < t$, outputs `false`. Otherwise, outputs $\mathsf{pid} = \mathsf{pid}_{\mathsf{recv}}$.

Main:
recv $(\texttt{InitAttr}, x)$ **from** `I/O` **s.t.** $\mathsf{sk}_x = \bot$:
parse $\mathsf{sid}_{\mathsf{cur}}$ **as** $(\mathsf{pid}, \mathsf{sid})$
send $(\texttt{Register}, x)$ **to** $(\mathsf{pid}, \mathsf{sid}_{\mathsf{cur}}, \texttt{setup})$
wait for $(\texttt{Registered}, \mathsf{sk}'_x)$
if $\mathsf{sk}'_x \neq \bot$: $\mathsf{sk}_x \leftarrow \mathsf{sk}'_x$

recv $(\texttt{Decrypt}, \psi_y)$ **from** `I/O` **s.t.** $T < t \wedge \mathsf{sk}_x \neq \bot$:
$m \leftarrow \mathsf{Dec}(\mathsf{sk}_x, \psi_y)$
reply $(\texttt{Plaintext}, m)$

Fig. 4: Description of the protocol $\mathcal{P}_{\mathsf{ABE}}$.

key mpk, a message m and an access policy y, *if mpk is the valid master public key and if there is no corrupted attribute x such that $x \in y$*, then it must be infeasible to distinguish the real message m encrypted in the real protocol and the encryption of the leakage $L(\lambda, m)$ where L is the length-preserving deterministic leakage function. Note that this leakage function is given as a protocol parameter, and can be instantiated by an higher-protocol. A winning adversary against the indistinguishability property of the ABE scheme can be used to construct an distinguisher against the $\mathcal{P}_{\mathsf{ABE}}$ and $\mathcal{F}_{\mathsf{ABE}}$. In case where the ciphertext encrypts a leakage, the ciphertext and the associated message are stored in the internal state of the instance, later used for the decryption. A ciphertext decryption request handled by entities having the `decryptor` role expects as an input a ciphertext ψ_y. The procedure is only executed when the decryptor entity has been registered to have a decryption key sk_x. In case where the received ciphertext is stored in the internal state, along the associated plaintext then the plaintext is directly returned as a response. If the ciphertext is not stored in the internal state, therefore the ciphertext has been computed outside of the ideal functionality and hence no security can be proven. So we decrypt ψ_y using the provided decryption algorithm Dec and returns the output as the plaintext response. Our real attribute-based encryption protocol $\mathcal{P}_{\mathsf{ABE}}$, presented in Fig. **??**, follows the specification of $\mathcal{F}_{\mathsf{ABE}}$, hence we omit the full description.

Lemma 1. *Assuming the existence of a perfectly-correct and* IND-CCA*2-secure attribute-based encryption $\Pi = (\mathsf{Setup}, \mathsf{Enc}, \mathsf{KeyGen}, \mathsf{Dec})$, then $\mathcal{P}_{\mathsf{ABE}} \leq \mathcal{F}_{\mathsf{ABE}}$.*

Proof. Suppose a perfectly-correct IND-CCA2-secure attribute-based encryption Π. We start by giving the description of our simulator $\mathcal{S}$, used with our ideal functionality $\mathcal{F}_{\mathsf{ABE}}$ in order to show that $\mathcal{P}_{\mathsf{ABE}} \leq \mathcal{F}_{\mathsf{ABE}}$. The simulator $\mathcal{S}$ starts the simulation by generating a new master key pair $(msk, mpk) \leftarrow \Pi.\mathsf{Setup}(1^\lambda)$, and it sends the initialization request $(mpk, \Pi.\mathsf{Enc}, \Pi.\mathsf{Dec})$ to $\mathcal{F}_{\mathsf{ABE}}$. When a request of the form $(\texttt{InitReceiver}, x)$ is sent from $\mathcal{F}_{\mathsf{ABE}}$ to $\mathcal{S}$, an honest decryptor is initialized and hence the simulator generates the decryption key $\mathsf{sk}_x \leftarrow \Pi.\mathsf{KeyGen}(msk, x)$ and responds with sk_x. A notification of the form $(\texttt{Registered}, x, b)$ is then received from the ideal functionality. If the bit b equals 1, then $\mathcal{S}$ registers than this decryptor has claimed the attribute x, otherwise it ignores the notification. Recall that we place ourself under the static corruption in which the adversary is allowed to corrupt only an entity directly after its initialization and only at this point of the entity lifetime. In details, the static corruption is initatiated by the entity, asking to the environment $\mathcal{E}$ its initial corruption status. We now specify the behavior of our simulator acting differently depending on the current time $t \in \mathbb{N}$ with respect to the time $T \in \mathbb{N}$:

- Case $t \leq T$: In case where the adversary corrupts (directly after the initialization) a decryptor and asks in the decryptor's name to obtain a decryption key to the `setup` entity using a request of the form $(\texttt{Register}, x)$ in the simulated real protocol, then the simulator executes honestly the decryption key generation code, but also notifies the ideal functionality of the corruption of x with the `CorrAttr` request.

– Case $T < t$: By construction of our real protocol, every `Register` requests are blocked and hence no more decryption key is provided to the environment.

Since we assume static corruption, every corruption request sent by the environment to initialized entities is blocked by the simulator, preventing dynamic corruption (under which no security can be proven). To be more clear, we suppose without loss of security that $\mathcal{E}$ always use the valid master public encryption key to an `encryptor`, since it does not provide any advantage for $\mathcal{E}$ to break the security of Π.

Hybrid 0. This hybrid corresponds to the execution of the ideal protocol $\mathcal{S}|\mathcal{F}_{\mathsf{ABE}}$ with the environment $\mathcal{E}$.

Hybrid 1. In this hybrid, we replace $\mathcal{S}|\mathcal{F}_{\mathsf{ABE}}$ with $\mathcal{S}'|\mathcal{F}_{\mathsf{Fwd}}$ where $\mathcal{S}'$ simulating $\mathcal{S}|\mathcal{F}_{\mathsf{ABE}}$ and where $\mathcal{F}_{\mathsf{Fwd}}$ is a forwarding IITM, transferring every request from $\mathcal{S}'$ to $\mathcal{E}$ and $\mathcal{E}$ to $\mathcal{S}'$. Requests coming from the network interface of $\mathcal{S}'$ are directly transferred to $\mathcal{S}$. Since we do not have perform any modification, we have perfect indistinguishability: $\Pr\left[(\mathcal{E}|\mathcal{S}|\mathcal{F}_{\mathsf{ABE}})(1^\lambda) \to 1\right] = \Pr\left[(\mathcal{E}|\mathcal{S}'|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right]$. For more clarity, we index all the simulators by the hybrid's index, hence $\mathcal{S}'$ is referred as $\mathcal{S}'_1$.

Hybrid 2. Observe that by the perfect correctness of the attribute-based encryption, the correctness issues occurring during both the register procedure (ensuring valid generation of decryption keys) and during encryption cannot occurs. Hence in this hybrid, we modify $\mathcal{S}'_1$ to construct $\mathcal{S}'_2$ in which we remove these correctness validation procedures in this hybrid without impacting the view of $\mathcal{E}$. Hence, $\Pr\left[(\mathcal{E}|\mathcal{S}'_1|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right] = \Pr\left[(\mathcal{E}|\mathcal{S}'_2|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right]$.

Hybrid i for $i \in [2, n+2]$. In this hybrid, we focus on the i-th encryption request sent to the simulator $\mathcal{S}'_i$. In this hybrid, we replace the encryption of the leakage $L(\lambda, m_i)$ by the encryption of the message m_i. Suppose that $\mathcal{E}$ is able to distinguish with a non-negligible probability between $(\mathcal{E}|\mathcal{S}'_i|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda)$ and $(\mathcal{E}|\mathcal{S}'_{i-1}|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda)$. We construct an adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ against the security of Π, simulating $\mathcal{E}$ and whose the role is create a perfect simulation of $\mathcal{S}'_i|\mathcal{F}_{\mathsf{Fwd}}$ without having access to the master secret key msk owned by the challenger of the IND-CCA2 game. In details, $\mathcal{A}_1$ is running *before* obtaining the challenge ciphertext, and $\mathcal{A}_2$ continues the run of the simulation of $\mathcal{E}$ *after* that the challenge ciphertext was obtained. The adversary $\mathcal{A}_1$ obtains as the input the encryption key mpk, and has access to the key generation oracle $\mathsf{OKeyGen}$ and the decryption oracle ODec. The adversary $\mathcal{A}_1$ works as follows:

– When receiving a `PubKey?` request, it responds $(\texttt{PubKey}, mpk)$.
– When receiving a `InitAttr` request, it simulates a `Register` request with the same provided parameters, described below.
– When $\mathcal{E}$ sends a key generation request of the form $(\texttt{Register}, x)$ from a `decryptor` entity: If there is no record $(\mathsf{entity}, \_, \_)$ yet, then it calls the $\mathsf{OKeyGen}$ oracle to generate a secret decryption key sk_x associated to the provided attribute x, it registers $(\mathsf{entity}, \mathsf{sk}_x, x)$ and returns sk_x. Otherwise, it returns an error.
– When $\mathcal{E}$ sends a request of the form $(\texttt{Encrypt}, mpk, y_j, m_j)$ for some $j < i$, then $\mathcal{A}_1$ computes $\psi_j \leftarrow \mathsf{Enc}(mpk, y_j, m_j)$ and responds with $(\texttt{Ciphertext}, \psi_j)$.

When $j = i$, then $\mathcal{A}_1$ computes the leakage $\bar{m}_j \leftarrow L(\lambda, m_j)$ where L is a length-preserving leakage function, and encodes its all internal state in the state variable, and sends to the challenger the challenge response $((y_j, m_j, \bar{m}_j), \mathsf{state})$.

- When $\mathcal{E}$ sends a decryption request of the form $(\mathtt{Dec}, \psi_j)$ from entity for $j < i$, it checks that the entity has already asked for decryption key sk_x by checking if there is a record $(\mathsf{entity}, \mathsf{sk}_x, x)$. If there is no match, aborts with $(\mathtt{Plaintext}, \perp)$. Otherwise, computes $m_j \leftarrow \Pi.\mathsf{Dec}(\mathsf{sk}_x, \psi_j)$ and returns $(\mathtt{Plaintext}, m_j)$.

We now describe our second adversary $\mathcal{A}_2$ taking as an input the challenge ciphertext ψ_i and the state state constructed by $\mathcal{A}_1$ used to continue the simulation of $\mathcal{E}$. The adversary $\mathcal{A}_2$ works as follows:

- The adversary $\mathcal{A}_2$ begins its simulation by sending the ciphertext ψ_i to $\mathcal{E}$, that is supposed to encrypt either m_i or $\bar{m}_i$. Observe that when m_i is encrypted, then $\mathcal{A}$ is simulating $(\mathcal{E}|\mathcal{S}'_i)$ or $(\mathcal{E}|\mathcal{S}'_{i-1})$ otherwise.
- The $\mathtt{InitAttr}$ and $\mathtt{Register}$ requests are the same as defined for $\mathcal{A}_1$.
- When $\mathcal{E}$ sends a request of the form $(\mathtt{Encrypt}, mpk, y_j, m_j)$ for some $j > i$, then it computes $\psi_j \leftarrow \mathsf{Enc}(mpk, y_j, L(\lambda, m_j))$ and records (ψ_j, y, m) and finally responds with $(\mathtt{Ciphertext}, \psi_j)$.
- When $\mathcal{E}$ sends a decryption request of the form $(\mathtt{Dec}, \psi_j)$ from entity for $j > i$: If there is no record $(\mathsf{entity}, \mathsf{sk}_x, x)$ or no record (ψ_j, y, m_j) then it responds with a failure. The case where there is several records for the same ciphertext is not considered since prevented by the attribute-based encryption scheme Π. If $x \in y$, then it responds with $(\mathtt{Plaintext}, m)$, otherwise it responds with $(\mathtt{Plaintext}, \perp)$.
- When $\mathcal{E}$ stops the simulation, $\mathcal{A}_2$ outputs 1 if $\mathcal{E}$ outputs 1. Otherwise, $\mathcal{A}_2$ outputs 0.

It is clear that our adversary $\mathcal{A}$ is polynomial-time. Since $\mathcal{E}$ is universally bounded, hence $\mathcal{A}$ constitutes a valid adversary for our IND-CCA2 experiment. Hence, we have:

$$\begin{aligned}
\mathsf{Adv}^{\text{IND-CCA2}}_{\mathcal{A},\Pi} &= \left| \frac{1}{2} \cdot \Pr\left[b' = 0 | b = 0\right] + \frac{1}{2} \cdot \Pr\left[b' = 1 | b = 1\right] - \frac{1}{2} \right| \\
&= \left| \frac{1}{2} \cdot \left(\Pr\left[b' = 0 | b = 0\right] - \Pr\left[b' = 1 | b = 0\right]\right) \right| \\
&= \frac{1}{2} \cdot \left| \Pr\left[(\mathcal{E}|\mathcal{S}'_i|\mathcal{F}_{\mathsf{Fwd}})(\lambda) \to 1\right] - \Pr\left[(\mathcal{E}|\mathcal{S}'_{i-1}|\mathcal{F}_{\mathsf{Fwd}})(\lambda) \to 0\right] \right|
\end{aligned}$$

Therefore, we conclude on the indistinguishability between $\mathcal{E}|\mathcal{S}'_2|\mathcal{F}_{\mathsf{Fwd}}$ and $\mathcal{E}|\mathcal{S}'_{n+2}|\mathcal{F}_{\mathsf{Fwd}}$ by:

$$\begin{aligned}
&\left| \Pr\left[(\mathcal{E}|\mathcal{S}'_2|\mathcal{F}_{\mathsf{Fwd}})(\lambda) \to 1\right] - \Pr\left[(\mathcal{E}|\mathcal{S}'_{n+2}|\mathcal{F}_{\mathsf{Fwd}})(\lambda) \to 0\right] \right| \\
&\leq n \cdot \sum_{i=3}^{n+2} \left| \Pr\left[(\mathcal{E}|\mathcal{S}'_i|\mathcal{F}_{\mathsf{Fwd}})(\lambda) \to 1\right] - \Pr\left[(\mathcal{E}|\mathcal{S}'_{i-1}|\mathcal{F}_{\mathsf{Fwd}})(\lambda) \to 0\right] \right| \\
&\leq 2n \cdot \mathsf{Adv}^{\text{IND-CCA2}}_{\mathcal{A},\Pi}
\end{aligned}$$

Hybrid $n + 3$. Observe that at this point, every ciphertext is encrypting the real message. Hence, instead of performing a plaintext recovery from the internal

state of our simulator, our modified simulator $\mathcal{S}'_{n+3}$ ignores the ciphertext register and directly performs the decryption. As a consequence, we do not perform the attribute validation check $x \in y$, that we remove from the (simulated) ideal functionality $\mathcal{F}_{\mathsf{ABE}}$. By correctness of the attribute-based encryption, we have $\Pr\left[(\mathcal{E}|\mathcal{S}'_{n+2}|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right] = \Pr\left[(\mathcal{E}|\mathcal{S}'_{n+3}|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right]$.
Hybrid $n+4$. This hybrid works exactly as the previous hybrid except that we do not share the master public key in the simulated $\mathcal{F}_{\mathsf{ABE}}$. Instead, the simulated $\mathcal{F}_{\mathsf{ABE}}$ asks the simulator $\mathcal{S}'_{n+4}$ to obtain the master public key and is returned back to the environment. Since the master public key initially stored in the simulated $\mathcal{F}_{\mathsf{ABE}}$ is already the master public key generated by the simulator, then the view of $\mathcal{E}$ is not changed. Similarly, the encryption and the decryption procedures done in the simulated ideal functionality $\mathcal{F}_{\mathsf{ABE}}$ are delegated to the simulators, forwarding for instance the request $(\texttt{Encrypt}, y, m, mpk)$ to the same encryptor in the simulated $\mathcal{P}_{\mathsf{ABE}}$. We follow the same approach for decryption requests. The response produced by the encryptor or the decryptor in the simulated protocol $\mathcal{P}_{\mathsf{ABE}}$ is returned back to the simulated ideal functionality $\mathcal{F}_{\mathsf{ABE}}$, forwarding the response to the environment $\mathcal{E}$. Finally, since the simulated $\mathcal{F}_{\mathsf{ABE}}$ does note use its internal state anymore, we remove it. Observe that all these modifications does not affect the view of $\mathcal{E}$ since the simulated ideal functionality $\mathcal{F}_{\mathsf{ABE}}$ was not performing any check or internal state access. Hence, we have $\Pr\left[(\mathcal{E}|\mathcal{S}'_{n+3}|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right] = \Pr\left[(\mathcal{E}|\mathcal{S}'_{n+4}|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right]$.

At this point, our simulator $\mathcal{S}'_{n+4}$ constitutes the most interesting part of the protocol, encrypting, decrypting and generating keys for entities, without performing any attribute validation (as done in our original $\mathcal{F}_{\mathsf{ABE}}$ *i.e.,* $x \in y$), and does not consider any internal state between the simulated entities. In other words, $\mathcal{S}'_{n+4}$ is our real protocol $\mathcal{P}_{\mathsf{ABE}}$. Even more, the simulated ideal functionality $\mathcal{F}_{\mathsf{ABE}}$ is now limited to forward the machine. As result, our simulator $\mathcal{S}'_{n+4}$ is now connected to the environment via the intermediate of two forwards machines. By removing these two forward machines $\mathcal{F}_{\mathsf{Fwd}}$ from $\mathcal{S}'|\mathcal{F}_{\mathsf{Fwd}}|\mathcal{F}_{\mathsf{Fwd}}$ and connect every wires from the environment via the $\texttt{I/O}$ interface directly to $\mathcal{S}'_{n+4}$, all these modifications being structural, we are ensured to have a perfect indistinguishability. Since all parties are simulated by $\mathcal{F}_{\mathsf{ABE}}$ and each party follows the instruction of the real protocol $\mathcal{P}_{\mathsf{ABE}}$ without having access to any shared register between entities, we have $(\mathcal{E}|\mathcal{S}'_{n+4}) = (\mathcal{E}|\mathcal{P}_{\mathsf{ABE}})$. By our hybrid argument, we have shown that $(\mathsf{Enc}|\mathcal{S}|\mathcal{F}_{\mathsf{ABE}}) \equiv (\mathsf{Enc}|\mathcal{P}_{\mathsf{ABE}})$, thus $\mathcal{P}_{\mathsf{ABE}} \leq \mathcal{F}_{\mathsf{ABE}}$. Since the protocol is environmentally bounded and complete, then the Lemma **??** holds.

4 Authenticated Attribute-based File Transfer

4.1 Description of our Ideal Functionality $\mathcal{F}_{\mathsf{AAFT}}$

Our authenticated attribute-based file transfer ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$ depicted in Fig. **??**, has been designed to allow an higher-protocol to easily rely on authenticated attribute-based file transfer. Each entity managed by the instance of $\mathcal{F}_{\mathsf{AAFT}}$ is associated to one of two following roles: A role `sender` representing

Ideal functionality $\mathcal{F}_{\mathsf{AAFT}} = (\texttt{sender}, \texttt{receiver})$:

Participating roles: `sender`, `receiver`
Corruption model: static corruption
Protocol parameters:
- A time $T \in \mathbb{N}$ delimiting phase in which file sending is not accessible. We denote by $t \in \mathbb{N}$ the current time.

$M_{\texttt{sender},\texttt{receiver}}$:

CheckID(pid, sid, role): Accept all entities with the same SID.

Corruption behavior:
- **AllowCorruption**(pid, sid, role): Returns role $\neq$ `decryptor` or $t < T$.

Internal state:
- $\mathsf{attr} \subseteq (\{0,1\}^*)^3 \times \{0,1\}^* \times \{0,1\} = \emptyset$
- $\mathsf{sentFiles} \subseteq (\{0,1\}^*)^3 \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* = \emptyset$
- $\mathsf{receivedFiles} \subseteq (\{0,1\}^*)^3 \times \{0,1\}^* = \emptyset$

Main:

recv $(\texttt{InitAttr}, x)$ **from** `I/O` **to** $(\_, \_, \texttt{receiver})$ **s.t.** $\nexists(\mathsf{entity}_{\mathsf{cur}}, \cdot, \cdot) \in \mathsf{attr}$:
  add $(\mathsf{entity}_{\mathsf{cur}}, x, 0)$ **to** attr
  send $(\texttt{Registered}, x)$ **to** `NET`

recv $(\texttt{CorrAttr}, receiver, x)$ **from** `NET` **s.t.** $t \leq T$:
  get $(receiver, x, b)$ **from** attr
  $b \leftarrow 1$

recv $(\texttt{Send}, f, y)$ **from** `I/O` **to** $(\_, \_, \texttt{sender})$ **s.t.** $T < t$:
  $r \leftarrow\!\!\$\ \{0,1\}^\lambda$
  add $(\mathsf{entity}_{\mathsf{cur}}, y, r, f)$ **to** sentFiles
  if $\exists\ (\cdot, x, 1)$ **s.t.** $x \in y$:
    send $(\texttt{SendCorrupted}, y, r, f)$ **to** `NET`
  else:
    send $(\texttt{SendHonest}, y, r, |f|)$ **to** `NET`

recv $(\texttt{Receive}, y, r, sender)$ **from** `NET` **to** $(\_, \_, \texttt{receiver})$:
  if $\nexists\ (sender, \cdot, r, \cdot) \in \mathsf{sentFiles}$:
    send responsively $(\texttt{WaitFile}, y, r, sender)$ **to** `NET`
    wait for $(\texttt{ProvideFile}, b, f)$
    if $b = 1$:
      add $(\mathsf{entity}_{\mathsf{cur}}, f)$ **to** receivedFiles
  else:
    get $(sender, y, r, f)$ **from** sentFiles
    if $\mathsf{attr}[\mathsf{entity}_{\mathsf{cur}}] \in y$:
      add $(\mathsf{entity}_{\mathsf{cur}}, f)$

recv `Collect` **from** `I/O` **to** $(\_, \_, \texttt{receiver})$:
  reply $\mathcal{F} = \{f : (\mathsf{entity}_{\mathsf{cur}}, f) \in \mathsf{receivedFiles}\}$

Fig. 5: Description of our ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$.

an entity sending a file and a role `receiver` receiving a file. The ideal functionality maintains three distinct internal states attr, sentFiles and receivedFiles used respectively to remember inputted and corrupted attributes, to authenticate files sent by honest senders and finally to store valid received files, eventually shared with the environment via the `Collect` request.

A sender handles only `Send` requests coming from the `I/O` interface, expecting as a parameter the input data file f as well as the access policy y. A short tag r is uniformly sampled from $\{0,1\}^\lambda$ and stored along the file f and the access policy y into the sentFiles internal states, shared between all entities (sharing the same session identifier). This set consists of all authenticated files. To send the file, the ideal functionality shares the file to the simulator $\mathcal{S}$ using one of two manner

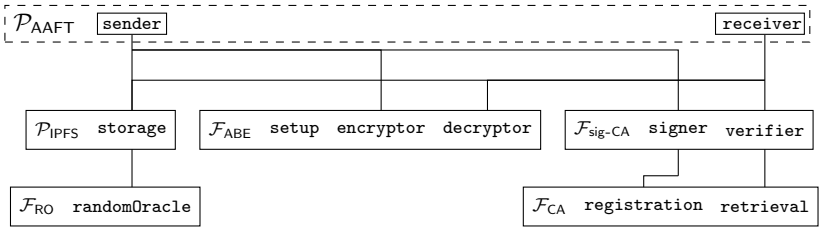

Fig. 6: Graphical representation of our protocol $\mathcal{P}_{\mathsf{AAFT}}$. The random oracle ideal functionality $\mathcal{F}_{\mathsf{RO}}$ comes from [**?**], whereas the ideal functionalities for digital signature $\mathcal{F}_{\mathsf{sig\text{-}CA}}$ and certificate authority $\mathcal{F}_{\mathsf{CA}}$ comes from [**?**].

depending on the corruption of attributes: If the environment has an attribute $x \in y$ then confidentiality of the file f cannot be ensured, hence shared with $\mathcal{S}$. On the other hand, the environment does not have an attribute $x \in y$, hence the file is not shared with the simulator $\mathcal{S}$. It models confidentiality in the sense that it remains safely in the ideal functionality, following the standard encryption in UC such that [**?**] producing ciphertext encrypting a leakage instead of the real plaintext. In contrast with the `Send` request handling requests from the higher-protocol, the file reception modelled by the `Receive` request is received from the `NET` interface *i.e.,* from the simulator. This is motivated by the real-life mail system in which the server receives files from the network, awaiting the user to connect in order to collect messages. It is up to the simulator to correctly simulates the protocol and notifies the ideal functionality if a receiver receives a file. Observe that the code for `Receive` ensures, in case of honest sender, authentication and file access depending on the attribute x owned by the current receiver by checking if $x \in y$. If the sender is corrupted, then authentication and file access is delegated to the simulator.

Observe that the `Send` and `CorrAttr` functions are accessible only if the current time $t \in \mathbb{N}$ is strictly greater than a constant time $T \in \mathbb{N}$ defined as a parameter of the protocol, a crucial restriction to include the ABE ideal functionality in our hybrid protocol. This restriction leads us to seperate the time in two distinct phases. During the first phase, we allow the environment to instantiate any entities but also to *statically corrupt* any entity of its choice, including receivers and hence to obtain attributes. During the second phase, we prevent the environment to corrupt a receiver, and allow the environment to send a file.

4.2 Our Hybrid Protocol $\mathcal{P}_{\mathsf{AAFT}}$

Depicted in Fig. **??**, our hybrid protocol $\mathcal{P}_{\mathsf{AAFT}}$ is proved to realize our ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$. It relies on several subroutines to model respectively a certificate authority, the random oracle, digital signatures, but also IPFS being part of our contribution. We first present all these subroutines before to introduce our hybrid protocol $\mathcal{P}_{\mathsf{AAFT}}$.

Ideal functionality $\mathcal{F}_{\mathsf{CA}} = (\texttt{registration}, \texttt{retrieval})$:

Participating roles: `registration`, `retrieval`
Corruption model: incorruptible

$M_{\texttt{registration,retrieval}}$:

Implemented role(s): {`registration`, `retrieval`}
Internal state:
- $\mathsf{keys} : (\{0,1\}^*)^2 \to \{0,1\}^* \cup \{\bot\}$

CheckID(pid, sid, role): Accept all entities.

Main:

```
recv (RegisterKey, key) from I/O to (_, _, registration) :
  if keys[pid_call, sid_call] ≠ ⊥:
    reply (RegisteredKey, false)
  else:
    keys[pid_call, sid_call] ← key
    reply (RegisteredKey, true)
recv (RetrieveKey, (pid, sid)) from _ to (_, _, retrieval) :
  reply (RetrievedKey, keys[pid, sid])
```

Ideal functionality $\mathcal{F}_{\mathsf{RO}} = (\texttt{randomOracle})$:

Participating roles: `randomOracle`
Corruption model: incorruptible

$M_{\texttt{randomOracle}}$:

Implemented role(s): `randomOracle`
Internal state:
- $\mathsf{H} \subseteq \{0,1\}^* \times \{0,1\}^\lambda = \emptyset$

CheckID(pid, sid, role): Accept all entities.

Main:

```
recv (Hash, m) from _ :
  if ∃(m, h) ∈ H:
    reply (Hashed, h)
  else:
    h ←$ {0,1}^λ
    add (m, h) to H
    reply (Hashed, h)
```

Fig. 7: Ideal functionalities $\mathcal{F}_{\mathsf{CA}}$ [**?**] and $\mathcal{F}_{\mathsf{RO}}$ [**?**].

Description of $\mathcal{F}_{\mathsf{CA}}$. The Certificate Authority (CA) allows to register public keys and certifying that a given public key corresponds to some user. The modelisation of $\mathcal{F}_{\mathsf{CA}}$ presented in Fig. **??** is taken from [**?**], consisting of two roles `registration` and `retrieval` , permitting respectively to register key and to retrieve a public key pk associated to the pair (pid, sid). Since the ideal functionality is self-explained, we omit its description, and only notice that the instance of $\mathcal{F}_{\mathsf{CA}}$ cannot be corrupted by the adversary and manages all entities, meaning that there is a single instance.

Description of $\mathcal{F}_{\mathsf{RO}}$. The ideal functionality $\mathcal{F}_{\mathsf{RO}}$ introduced in [**?**] and presented in Fig. **??** exposes a single role `randomOracle`. This straightforward ideal functionality handles `Hash` requests given an arbitrary-sized bitstring m, asso-

Protocol $\mathcal{P}_{\mathsf{IPFS}} = (\texttt{storage})$:

Participating roles: `storage`
Corruption model: dynamic corruption without erasure

$M_{\texttt{storage}}$:

Subroutines: $\mathcal{F}_{\mathsf{RO}} : \texttt{randomOracle}$
Implemented role(s): `storage`
Internal state:
– $\mathsf{files} : \{0,1\}^* \to \{0,1\}^* = \emptyset$ {*Stored files*

CheckID(pid, sid, role): Accept a single entity.

Main:

recv $(\texttt{Upload}, f)$ **from** _ :
  send $(\texttt{Hash}, f)$ **to** $(\overline{\mathsf{pid}}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathsf{RO}} : \texttt{randomOracle})$
  wait for $(\texttt{Hashed}, l)$
  $\mathsf{files}[l] \leftarrow f$
  reply $(\texttt{Uploaded}, l)$

recv `Links` **from** _ :
  reply $\{l : \forall l \overset{\_}{\mapsto} f \in \mathsf{files}\}$

recv $(\texttt{Download}, l)$ **from** _ :
  reply $(\texttt{Downloaded}, \overline{\mathsf{files}}[l])$

Fig. 8: Protocol $\mathcal{P}_{\mathsf{IPFS}}$ for a storage server in the IPFS network.

ciates a λ-sized random bitstring h. In particular, if m was never queried before, then $\mathcal{F}_{\mathsf{RO}}$ generates a random bitstring h, stores the couple (m, h) and returns h. Otherwise (m has already been queried), and thus $\mathcal{F}_{\mathsf{RO}}$ returns the h associated to m. Similarly to $\mathcal{F}_{\mathsf{CA}}$, we assume that the ideal functionality $\mathcal{F}_{\mathsf{RO}}$ manages all entities, meaning that there is a single instance of $\mathcal{F}_{\mathsf{RO}}$ in the protocol.

Description of $\mathcal{P}_{\mathsf{IPFS}}$. An IPFS network consists of connected servers, maintaining an internal state associating to a file f a link l where l is the hash of f computed using the cryptographic hash function. IPFS plays a central role in the efficiency of our construction by allowing a potentially large data to be transferred over a distributed storage network. We have modelled a single storage server in iUC as a real protocol denoted $\mathcal{P}_{\mathsf{IPFS}}$ and depicted in Fig. **??**. A storage server having the role `storage`, relies on the random oracle $\mathcal{F}_{\mathsf{RO}}$ used to hash files. It is equipped of the three following functions: `Upload` used to store files, `Links` returning all saved links, and `Download` which given a link l returns the file f associated with l. A storage server is not intended to provide more than the efficiency in our construction. Hence it can be dynamically corrupted without erasure *i.e.,* corruption occurs at any time, leaving the full control of the corrupted server to the adversary, all its internal state being leaked.

Description of $\mathcal{F}_{\mathsf{sig\text{-}CA}}$. The ideal functionality for digital signature, introduced in [**?**] and recalled in Fig. **??**, is composed of the two roles `signer` and `verifier`, allowing respectively to create a signature of a given message and to verify a signature. During the instance initialization, the party identifier pid' of the signer is obtained from the session identifier having the form $\mathsf{sid} = (\mathsf{pid}', \mathsf{sid}')$. Before providing any signature, the signer expects from the higher protocol an initialization

Ideal functionality $\mathcal{F}_{\mathsf{sig\text{-}CA}} = (\texttt{signer}, \texttt{verifier})$:

Participating roles: `signer`, `verifier`
Corruption model: static corruption
Protocol parameters:
- A polynomial $p \in \mathbb{Z}[x]$ used to bound the runtime execution of provided algorithms.

$M_{\texttt{signer},\texttt{verifier}}$:

Implemented role(s): `signer`, `verifier`
Subroutines: $\mathcal{F}_{\mathsf{CA}}$: `registration`
Internal state:
- $(\mathsf{Sign}, \mathsf{Verif}, \mathsf{pk}, \mathsf{sk}) \in (\{0,1\}^* \cup \{\bot\})^4 = (\bot, \bot, \bot, \bot)$
- $\mathsf{pidowner} \in \{0,1\}^* \cup \{\bot\} = \bot$
- $\mathsf{msgList} \subseteq \{0,1\}^* = \emptyset$
- $\mathsf{KeysGenerated} \in \{\mathsf{ready}, \bot\} = \bot$

CheckID(pid, sid, role): Check that sid has a $(\mathsf{pid}', \mathsf{sid}')$ format. If the check fails, return `false`, otherwise accept all entities with the same SID.

Corruption behavior:
- **LeakedData**(pid, sid, role): If called while $(\mathsf{pid}, \mathsf{sid}, \mathsf{role})$ determines its initial corruption status, use the default behavior of **LeakedData**. That is, output the initially received message and the sender of that message. Otherwise, if role = `signer` and pid = pidowner, return KeysGenerated. In all other cases, return $\bot$.
- **AllowAdvMessage**(pid, sid, role, $\mathsf{pid}_{\mathsf{recv}}$, $\mathsf{sid}_{\mathsf{recv}}$, $\mathsf{role}_{\mathsf{recv}}$, m): Check that $(\mathsf{pid} = \mathsf{pid}_{\mathsf{recv}})$. If $\mathsf{role}_{\mathsf{recv}} = \mathcal{F}_{\mathsf{CA}}$: `registration`, also check that role = `signer` and $\mathsf{sid} = (\mathsf{pid}, \mathsf{sid}')$. If all checks succeed, output `true`, otherwise output `false`.

Initialization:
send responsively InitMe **to** `NET`
wait for $(\texttt{Init}, (Sign, Verif, pk, sk))$
$(\mathsf{Sign}, \mathsf{Verif}, \mathsf{pk}, \mathsf{sk}) \leftarrow (Sign, Verif, pk, sk)$
parse $\mathsf{sid}_{\mathsf{cur}}$ **as** $(\mathsf{pid}, \mathsf{sid})$
$\mathsf{pidowner} \leftarrow \mathsf{pid}$

Main:
recv `InitSign` **from** `I/O` **to** $(\mathsf{pidowner}, \_, \texttt{signer})$:
  send $(\texttt{RegisterKey}, \mathsf{pk})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \epsilon, \mathcal{F}_{\mathsf{CA}}$: `registration`$)$
  wait for $(\texttt{RegisterKey}, \_)$
  $\mathsf{KeysGenerated} \leftarrow \texttt{true}$
  reply $(\texttt{InitSign}, 1)$

recv $(\texttt{Sign}, m)$ **from** `I/O` **to** $(\mathsf{pidowner}, \_, \texttt{signer})$ **s.t.** $\mathsf{KeysGenerated} = \texttt{true}$:
  $\sigma_m \leftarrow \mathsf{Sign}^{(p)}(m, \mathsf{sk})$
  $b \leftarrow \mathsf{Verif}^{(p)}(m, \sigma_m, \mathsf{pk})$
  if $\sigma_m = \bot \vee b \neq 1$:
    reply $(\texttt{Signature}, \bot)$
  else:
    add m **to** msgList
    reply $(\texttt{Signature}, \sigma_m)$

recv $(\texttt{Verify}, m, \sigma_m, pk)$ **from** `I/O` **to** $(\_, \_, \texttt{verifier})$:
  $b \leftarrow \mathsf{Verif}^{(p)}(m, \sigma_m, pk)$
  if $pk = \mathsf{pk} \wedge b = 1 \wedge m \notin \mathsf{msgList} \wedge (\mathsf{pidowner}, \mathsf{sid}_{\mathsf{cur}}, \texttt{signer}) \notin \mathsf{CorruptionSet}$:
    reply $(\texttt{VerResult}, \texttt{false})$ {*Prevents the signature forgeries*
  reply $(\texttt{VerResult}, b)$

Fig. 9: Description of the ideal functionality $\mathcal{F}_{\mathsf{sig\text{-}CA}}$ [?].

Protocol $\mathcal{P}_{\mathsf{AAFT}} = (\texttt{sender}, \texttt{receiver})$:

Participating roles: `sender`, `receiver`
Subroutines: $\mathcal{F}_{\mathsf{ABE}}, \mathcal{F}_{\mathsf{sig\text{-}CA}}, \mathcal{F}_{\mathsf{CA}} : \texttt{retrieval}, \mathcal{F}_{\mathsf{RO}}, \mathcal{P}_{\mathsf{IPFS}}$
Corruption model: static corruption
Protocol parameters:
- The party identifier pidsetup, identifying the entity of the ABE scheme handling the master keys.
- A time $T \in \mathbb{N}$ delimiting phase in which decryption keys are provided, from the phase where encryption and decryption are operated. We denote by $t \in \mathbb{N}$ the current time.

$M_{\texttt{sender}}$:

Implemented role(s): `sender`
CheckID(pid, sid, role): Accept a single entity.
Internal state:
- $mpk \in \{0,1\}^* \cup \{\bot\} = \bot$

Corruption behavior:
- **DetermineCorrStatus**(pid, sid, role): return $\mathsf{corr}(\mathsf{pid}, (\mathsf{pid}, \epsilon), \texttt{signer})$ or $\mathsf{corr}(\mathsf{pid}, (\mathsf{pidsetup}, \mathsf{sid}), \texttt{encryptor})$.
- **AllowAdvMessage**(pid, sid, role, $\mathsf{pid}_{\mathsf{recv}}$, $\mathsf{sid}_{\mathsf{recv}}$, $\mathsf{role}_{\mathsf{recv}}$, m): Check that $(\mathsf{pid} = \mathsf{pid}_{\mathsf{recv}})$.

Initialization:
  send PubKey? **to** $(\mathsf{pidsetup}, (\mathsf{pidsetup}, \mathsf{sid}_{\mathsf{cur}}), \mathcal{F}_{\mathsf{ABE}} : \texttt{setup})$
  wait for mpk'
  $mpk \leftarrow mpk'$
  send InitSign **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \epsilon)), \mathcal{F}_{\mathsf{sig\text{-}CA}} : \texttt{signer})$
  wait for _

Main:
  recv $(\texttt{Send}, y, f)$ **from** I/O **s.t.** $T < t$:
    send responsively Storage? **to** NET
    wait for $(\texttt{Storage}, storage)$
    send $(\texttt{Encrypt}, y, f, mpk)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathsf{ABE}} : \texttt{encryptor})$
    wait for $(\texttt{Ciphertext}, \psi_y)$
    send $(\texttt{Upload}, (y, \psi_y))$ **to** $storage$
    wait for _
    send $(\texttt{Hash}, (y, \psi_y))$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathsf{RO}} : \texttt{randomOracle})$
    wait for $(\texttt{Hashed}, l)$
    send $(\texttt{Sign}, l)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \epsilon), \mathcal{F}_{\mathsf{sig\text{-}CA}} : \texttt{signer})$
    wait for $(\texttt{Signature}, \sigma_l)$
    send $(\texttt{Sent}, l, \sigma_l)$ **to** NET

Fig. 10: Description of our protocol $\mathcal{P}_{\mathsf{AAFT}}$ (Part 1)

request allowing the signer to register the verification key pk to the certificate authority, modelled via the $\mathcal{F}_{\mathsf{CA}}$ ideal functionality. After this initialization step, the signer is allowed to sign any message. Note that before returning the signature, the signer stores the signed message m in a set of authenticated messages managed by the instance, used later for the signature verification. To verify a signature, an entity having the `verifier` role expects as an input a message m, a signature σ_m and a public verification key pk. The security of a digital signature is modelled by the ideal functionality by always rejecting every valid signature σ_m coming from an uncorrupted signer whose the message m does not belong to the set of authenticated messages, and whose the provided verification key pk is the valid one (*i.e.,* the public verification key provided by the adversary in the ideal functionality).

Description of our Hybrid Protocol $\mathcal{P}_{\mathsf{AAFT}}$. We are now ready to introduce $\mathcal{P}_{\mathsf{AAFT}}$ our hybrid file transfer protocol based on a hash-based distributed

$M_{\texttt{receiver}}$:

Implemented role(s): `receiver`
Internal state:
- initiated $\in \{0,1\} = 0$
- files $\subseteq \{0,1\}^* = \emptyset$

CheckID(pid, sid, role): Check that sid = (pid′, sid′). Accept a single entity.
Corruption behavior:
- **AllowCorruption**(pid, sid, role): Returns $t < T$.
- **DetermineCorrStatus**(pid, sid, role): return corr(pid, (pidsetup, sid), `decryptor`).
- **AllowAdvMessage**(pid, sid, role, pid_{recv}, sid_{recv}, $\text{role}_{\text{recv}}$, m): Check that ($\text{pid} = \text{pid}_{\text{recv}}$).

Main:

recv (`InitAttr`, x) **from** `I/O` **s.t.** initiated $= 0$:
  initiated $\leftarrow 1$
  send (`InitAttr`, x) **to** (pid_{cur}, sid_{cur}, $\mathcal{P}_{\text{ABE}}$: `decryptor`)

recv (`Receive`, $sender$, $storage$, l, σ_l) **from** `NET` **s.t.** $T < t \wedge$ initiated $= 1$:
  parse $storage$ **as** (_, _, $\mathcal{P}_{\text{IPFS}}$: `storage`), $sender$ **as** (pid, _, `sender`)
  send (`Download`, l) **to** $storage$
  wait for (`Downloaded`, (y, ψ_y))
  send (`RetreiveKey`, (pid, ϵ)) **to** (pid_{cur}, sid_{cur}, $\mathcal{F}_{\text{CA}}$: `retrieval`)
  wait for (`RetreivedKey`, pk)
  send (`Verify`, l, σ_l, pk) **to** (pid_{cur}, (pid, ϵ)), $\mathcal{F}_{\text{sig-CA}}$: `verifier`)
  wait for (`VerResult`, b)
  send (`Hash`, (y, ψ_y)) **to** (pid_{cur}, sid_{cur}, `randomOracle`)
  wait for (`Hashed`, l')
  send (`Decrypt`, ψ_y) **to** (pid_{cur}, (pidsetup, sid_{cur})), $\mathcal{F}_{\text{ABE}}$: `decryptor`)
  wait for (`Plaintext`, f)
  if $l = l' \wedge b = 1 \wedge f \neq \perp$: **add** f **to** files

recv `Collect` **from** `I/O` :
  reply files

Fig. 11: Description of our protocol $\mathcal{P}_{\mathsf{AAFT}}$ (Part 2)

storage system in Fig. **??** and Fig. **??**. Since our protocol is self-explained, we only highlight the overall behavior. During the reception of a file f along an access policy y, an honest sender encrypts f using the ideal functionality $\mathcal{F}_{\mathsf{ABE}}$ to obtain the ciphertext ψ_y. This ciphertext is hashed to obtain the link l and is sent to the storage (chosen by environment $\mathcal{E}$). After having obtained the signature σ_l for l, the sender shares the link l along the signature σ_l with the environment $\mathcal{E}$. When receiving a link l, a signature σ_l and two identities respectively for the storage and the sender, an honest receiver checks the validity of the signature with respect to the signer and the link and decrypts the ciphertext ψ_y, previously obtained from the designated storage server.

Theorem 1. *Assuming ideal functionalities for certificate authority $\mathcal{F}_{\mathsf{CA}}$, digital signature $\mathcal{F}_{\mathsf{sig\text{-}CA}}$, random oracle $\mathcal{F}_{\mathsf{RO}}$ and attribute-based encryption $\mathcal{F}_{\mathsf{ABE}}$ and the* IPFS *protocol $\mathcal{P}_{\mathsf{IPFS}}$, then* $(\texttt{sender}, \texttt{receiver} | \mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{ABE}}, \mathcal{F}_{\mathsf{sig\text{-}CA}}, \mathcal{F}_{\mathsf{CA}}, \mathcal{P}_{\mathsf{IPFS}})$ *realizes $\mathcal{F}_{\mathsf{AAFT}}$.*

Proof. We start this proof by giving a description of our simulator $\mathcal{S}$, used with the ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$ to show that $\mathcal{P}_{\mathsf{AAFT}} \leq \mathcal{S} | \mathcal{F}_{\mathsf{AAFT}}$. In a nutshell, $\mathcal{S}$ runs a simulation of the real protocol and handles request coming from both

the environment and the ideal functionality. During the simulation, without loss of security and functionality, we require the leakage function $L(\lambda, m)$ used by the ideal functionality $\mathcal{F}_{\mathsf{ABE}}$ to return a zero-string $0^{|m|}$. We now provide the description of $\mathcal{S}$:

- When receiving a notification request of the form $(\texttt{Registered}, x)$ from the ideal functionality (simulating a receiver), attesting the access to some attribute x. In such case, inputs the same receiver in the simulated real protocol with $(\texttt{InitAttr}, x)$.
- When receiving a corruption request from the environment to corrupt the entity entity defined by the triplet $(\mathsf{pid}, \mathsf{sid}, \mathsf{role})$. If the current time $t < T$, then it accepts the corruption request. Otherwise, ignore the corruption request. Each time a corruption request leads to the corruption either of a sender or a receiver, the simulator notifies the ideal functionality as well, leading to a synchronization of senders and receivers corruption between the simulated real protocol and the ideal functionality. In case where the entity in charge of handling the master secret keys receives a decryption key, assuming $t < T$, then the decryption key sk_x is computed and sent back to the corrupted decryptor. Observe that by construction, a corrupted decryptor equals a corrupted receiver. The simulator $\mathcal{S}$ notifies the ideal functionality that the environment is allowed to decrypt any ciphertext associated with a policy access y whose $x \in y$ by sending the request $(\texttt{CorrAttr}, x)$ to $\mathcal{F}_{\mathsf{AAFT}}$.
- When receiving a request of the form $(\texttt{SendHonest}, y, r, |f|)$ or the request of the form $(\texttt{SendCorrupted}, y, r, f)$ from the initial functionality, the simulator inputs the simulated sender of $\mathcal{P}_{\mathsf{AAFT}}$ with the access policy y and the real file f in the case of `SendCorrupted`, or the zero-string $0^{|f|}$ of the case of `SendHonest`. In addition, in case of `SendHonest`, the simulator records the pair (r, ψ_y) where ψ_y is the ciphertext obtained via the simulated `encryptor` from the ideal functionality $\mathcal{F}_{\mathsf{ABE}}$. This record is used later to provide the random r to the ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$.
- When receiving a request of the form $(\texttt{Receiver}, \textit{sender}, \textit{storage}, l, \sigma_l)$ from the environment $\mathcal{E}$ via the network interface, then it inputs the simulated real receiver, running all sanity checks including decryption, signature verification and link validation. Let ψ_y be the ciphertext obtained during the execution of the simulated receiver in $\mathcal{P}_{\mathsf{AAFT}}$. If all checks success, then if the simulator recovers the pair (r, ψ_y) and sends $(\texttt{Receive}, y, r, \textit{sender})$ to the ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$, and no response is expected later. Otherwise, there is no records (r, ψ_y) and hence it sends $(\texttt{Receive}, y, \perp, \textit{sender})$ to the ideal functionality. By construction, the ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$ responds with a responsive request $(\texttt{WaitFile}, y, r, \textit{sender})$ where r equals $\perp$. At this point, the ideal functionality expects a file f to register. Since there is no records by the simulator but the decryptor file f being authenticated, the simulator responds to this responsive request with $(\texttt{ProvideFile}, 1, f)$.

It is clear that our simulator $\mathcal{S}$ is polynomial-time. We are now ready to initiate our sequence of hybrids, where our first hybrid consists of the $(\mathcal{S}|\mathcal{F}_{\mathsf{AAFT}})$ ideal protocol, and our last hybrid is our hybrid protocol $\mathcal{P}_{\mathsf{AAFT}}$:

Hybrid 0. This hybrid is the execution of the ideal protocol $(\mathcal{S}|\mathcal{F}_{\mathsf{AAFT}})$ with the environment $\mathcal{E}$, connected respectively to the `NET` interface of $\mathcal{S}$ and to the `I/O` interface of $\mathcal{F}_{\mathsf{AAFT}}$, with $\mathcal{S}$ being connected to the `NET` interface of $\mathcal{F}_{\mathsf{AAFT}}$.

Hybrid 1. This hybrid works as the previous hybrid, except that we now execute the protocol $(\mathcal{S}'|\mathcal{F}_{\mathsf{Fwd}})$ where $\mathcal{S}'$ runs the simulated ideal protocol $(\mathcal{S}|\mathcal{F}_{\mathsf{AAFT}})$ and where $\mathcal{F}_{\mathsf{Fwd}}$ simply forwards every request from the environment $\mathcal{E}$ to $\mathcal{S}'$ and conversely. Since this modification is only structural without any modification on the ideal protocol behavior, then we have a perfect indistinguishability between these two hybrids. In the following, for a better clarity, we index each simulator $\mathcal{S}'$ with the current hybrid index, hence: $\Pr\left[(\mathcal{E}|\mathcal{S}'_1|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right] = \Pr\left[(\mathcal{E}|\mathcal{S}|\mathcal{F}_{\mathsf{AAFT}})(1^\lambda) \to 1\right]$.

Hybrid 2. This hybrid works as the previous hybrid, except that we introduce a new simulator $\mathcal{S}'_2$ consisting of $\mathcal{S}'_1$ where we delegate the all attribution verification both during the `Send` and `Receive` requests to the simulator $\mathcal{S}$. First, let focus on the `Send` part of the ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$, currently simulated by our simulator $\mathcal{S}'_2$. We rewrite the code of `Send` to remove the random r as well as the corrupted attribute condition, all of these lines being replaced only by a request of the form $(\texttt{Send}, y, f)$. The code still stores files being sent by an honest sender, but omits the random r *i.e.,* it records a tuple of the form (sender, y, f). The file reception request in the ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$, handling requests of the form $(\texttt{Receive}, y, r, \mathit{sender})$, now handles requests of the form $(\texttt{Receive}, y, f, \mathit{sender})$. The condition verifying of there is no tuple of the form $(\mathit{sender}, \cdot, r, \cdot)$ is still performed but with the tuple of the form $(\mathit{sender}, \cdot, f)$. The random r is replaced by $\perp$ in the code which is executed when the condition is checked. The executed code when the condition fails, including the attribute verification $(x \in y)$, is replaced by the insertion of the file f to the received file register. The simulator, on its side, does not register the pair (r, ψ_y) anymore.

Observe that all these modifications are hidden to the environment, and we claim that the view of the environment remains unchanged. This can be easily deduced since the at this point, the ideal functionality does not perform the attribute verification by itself but rather delegate this task to the ideal functionality $\mathcal{F}_{\mathsf{ABE}}$, which is secure and correct by design. However, compared to the previous hybrid, the received file is now always f instead of the zero-string. In case where the environment does not have a valid decryption key to decrypt ψ_y, it encrypts the leakage $L(\lambda, f)$ in this hybrid, instead of $L(\lambda, 0^{|f|})$ in the previous one. Thanks to our specification of L, always outputting a zero-string, both of these leakages are the same. Therefore, by construction of the ideal functionality $\mathcal{F}_{\mathsf{ABE}}$ and by the leakage function L, the environment cannot distinguish, otherwise breaking the security of $\mathcal{F}_{\mathsf{ABE}}$. Hence, $\Pr\left[(\mathcal{E}|\mathcal{S}'_2|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right] = \Pr\left[(\mathcal{E}|\mathcal{S}'_1|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right]$.

Hybrid 3. At this point, the (simulated) ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$ still maintain a set of sent files, essentially used to provide authentication of the files for honest senders. An honest receiver, on its side, verifies that a received file f belongs to the set of sent files (with respect to the provided sender) and responds to the simulator via the network interface if the file is not found. Observe that in this

case, the simulator already activates the (honest) receiver with a request `Receive` if and only if the provided signature σ_l authenticates the link l, which corresponds to the hash of the ABE ciphertext ψ_y. Hence, by construction, authentication of the file with respect to the provided sender is already ensured by the signature ideal functionality $\mathcal{F}_{\mathsf{sig\text{-}CA}}$. Hence, the condition in the ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$ is no more necessary and all the code handling `Receive` requests is now limited to add the received file f (added in the previous hybrid) to the set of received files. This constitutes our new simulator $\mathcal{S}'_3$. Since the authentication in the ideal functionality $\mathcal{F}_{\mathsf{sig\text{-}CA}}$ is correct and secure by definition, this modification does not impact the view of the environment $\mathcal{E}$ and hence $\Pr\left[(\mathcal{E}|\mathcal{S}'_3|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right] = \Pr\left[(\mathcal{E}|\mathcal{S}'_2|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right]$.

Hybrid 4. This hybrid works as the previous hybrid, except that we remove the attribute state attr that are not used anymore in this hybrid. Additionally, we replace the internal state $\mathsf{receivedFiles}$ by a local internal state specific to each receiver. This last modification clearly does not affect the view of the the environment $\mathcal{E}$. Hence, we have $\Pr\left[(\mathcal{E}|\mathcal{S}'_4|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right] = \Pr\left[(\mathcal{E}|\mathcal{S}'_3|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right]$.

Observe that in our last hybrid, the ideal functionality $\mathcal{F}_{\mathsf{AAFT}}$ simulated in our last simulator $\mathcal{S}'_4$ never rely on internal state and essentially constitutes a forward machine between the simulated hybrid protocol and the `I/O` interface where the environment is connected. As a result, we claim that $\Pr\left[(\mathcal{E}|\mathcal{S}'_4|\mathcal{F}_{\mathsf{Fwd}})(1^\lambda) \to 1\right] = \Pr\left[(\mathcal{E}|\mathcal{P}_{\mathsf{AAFT}})(1^\lambda) \to 1\right]$. Hence, we have $\mathcal{P}_{\mathsf{AAFT}} \leq \mathcal{F}_{\mathsf{AAFT}}$.

4.3 Implementation of $\mathcal{P}_{\mathsf{AAFT}}$

Our open-source proof-of-concept written in Rust confirms the practicality of our protocol [**?**]. We have chosen the Schnorr signature over the curve25519 curve as our EUF-CMA digital signature. Since our cryptographic hash function handles potentially large files, we have chosen to construct a parallelized Merkle-tree-based hash function using the standard SHA-256 cryptographic hash function as the underlying building block. To construct our IND-CCA2-secure ABE, we have applied the Fujisaki-Okamoto transform [**?**] on the Agrawal-Chase IND-CPA Ciphertext-Policy ABE scheme [**?**].We have used AES-256-CTR as the secret-key encryption within the transform. Benchmarks have been performed on an Ubuntu, embedding a 64 bits Intel Core i5-6500 processor cadenced at 3.20GHz including four cores, and embedding 16Gb of memory. In Fig. **??** is depicted the execution time of the sending and receiving procedures depending on the input file size. We directly observe that both procedures are mlinear. For a 450 megabytes file the sending procedure requires approximately 216 milliseconds, whereas the file receiving procedure expects 258 milliseconds. The difference between the sending and reception is explained by the Fujisaki-Okamoto transform, executing during the reception the encryption algorithm, used to reject malformed ciphertext. In Fig. **??**, we compare the amount of data sent by a sender to a receiver through the low-rate medium between the *naive* corresponding to the situation where the encrypted file is directly sent to the receiver (for example with OpenPGP [**?**]), and the approach motivated in the $\mathcal{P}_{\mathsf{AAFT}}$ protocol.

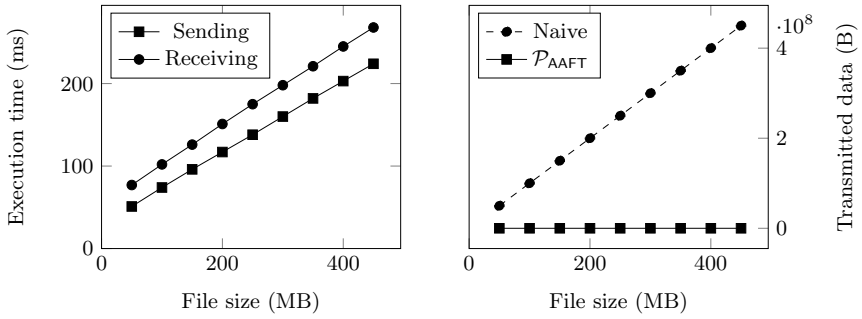

Fig. 12: Evaluation of the sending and receiving execution time (left) and the exchanged data size between users (right).

Compared to the naive approach, our solution provides a *constant* amount of transferred data of 96 bytes corresponding to the link (*i.e.,* the hash of the encrypted file) and the signature σ_l. This constant communication size is explained by the encrypted file being sent over the distributed storage network instead of being directly transferred, still with the guarantee to have confidentiality and integrity of the file but also authentication of the sender.

Acknowledgments. We thank the anonymous referees for their useful suggestions and remarks. This work was partially supported by the DataLake-For-Nuclear (D4N) project funded by the BPI institute.