Indirect Revocable KP-ABE with Revocation Undoing Resistance

Marco Rasori[1], Pericle Perazzo[1], Gianluca Dini[1], and Shucheng Yu[2]

[1] *Department of Information Engineering, University of Pisa*
[2] *Department of Electrical and Computer Engineering, Stevens Institute of Technology, Hoboken, USA*

April 7, 2021

Abstract

Lately, many cloud-based applications proposed Attribute-Based Encryption (ABE) as an all-in-one solution for achieving confidentiality and access control. Within this paradigm, data producers store the encrypted data on a semi-trusted cloud server, and users, holding decryption keys issued by a key authority, can decrypt data according to some access control policy. To be used in practical cases, any ABE scheme should implement a key revocation mechanism which assures that a compromised decryption key cannot be used anymore to decrypt data. Yu et al. (INFOCOM 2010) introduced an ABE scheme with revocation capabilities that enjoys several unique advantages, such as reactivity and efficiency. In the scheme, the cloud server is entitled to update keys and ciphertexts in order to achieve revocation. Unfortunately, the cloud server retains the power to undo the revocation of a key (revocation undoing attack) so endangering confidentiality. In this paper, we propose a revocable ABE scheme that still ensures the advantages of Yu et al.'s scheme, but it also resists to the revocation undoing attack. We formally prove the security of our scheme and show through simulations that the user experiences a slightly higher computational cost with respect to Yu et al.'s scheme.

1 Introduction

Attribute-Based Encryption (ABE) is a type of public-key cryptography able to provide confidentiality and fine-grained access control over the encrypted data. It is particularly suitable for applications in which the data is stored on semi-trusted storage services and should be accessed by users with different access privileges. ABE has been widely employed in disparate scenarios, e.g., IoT [1–6], digital health [7–9], blockchain [10, 11], and many others.

In ABE systems, a *key authority* generates *public parameters* for encryption and *decryption keys* for decryption. We refer to an encrypting entity as *data owner* and a decrypting entity as *user*. In practical applications, data owners store the encrypted data on some server, while users retrieve ciphertexts from it. Often, the storage service is provided by a third-party server, e.g., a *cloud server*, which is usually considered in the literature to be *honest-but-curious*, meaning that it does not deviate from the specified protocol, but it is also interested in accessing the encrypted data stored on it.

ABE schemes typically fall into two categories: Key-Policy Attribute-Based Encryption (KP-ABE) and Ciphertext-Policy Attribute-Based Encryption (CP-ABE). In KP-ABE schemes [12], ciphertexts are labeled with a set of attributes, and decryption keys embed an access policy defined over the attributes. Conversely, in CP-ABE schemes [13], ciphertexts embed the access policy, and decryption keys are labeled with a set of attributes.

In order to be useful in a practical case, any ABE scheme should implement a key revocation mechanism, which assures that a given decryption key cannot be used anymore to decrypt data. This is mandatory for example in case of key compromise. In a real-life scenario, the key authority revokes a decryption key when a user detects or suspects the compromise of his/her key, or when it becomes aware of an (alleged) compromise from third parties or public sources. Over the years, plenty of *revocable* ABE schemes have been proposed [14–18]. Existent ABE key revocation mechanisms can be classified into two categories: *direct revocation* and *indirect revocation*. Direct revocation is performed by maintaining the list of revoked users on each data owner, which encrypts data so as to exclude revoked keys from being capable of decrypting it. The drawback of this technique is that all data owners must have an updated copy of the revocation list. In indirect revocation, the key authority issues key update material through which only non-revoked users' decryption keys are updated. In the present paper, we focus on indirect revocation mechanisms.

Yu et al. [17] introduced an indirect revocable KP-ABE scheme in which revocation is achieved by a system-wide update of the attributes present in the key to revoke. Only the non-revoked decryption keys having at least an attribute in common with the revoked one are updated, while the revoked key is excluded from the update process. The advantage of this method with respect to others in the literature, e.g., [14–16], is threefold: (i) as long as no revocation occurs, the key authority can stay offline and perform no action; (ii) when a revocation occurs, not all the users –but only a portion of them, that we call *affected users*– need to update their decryption keys; and (iii) the revocation comes into force with immediate effect, and there is no need to wait for the end of some predefined time epoch.

In Yu et al.'s scheme, the task of updating affected users' decryption keys is delegated to the cloud server. To this aim, the cloud server is given some secret quantities, called *re-encryption keys*, through which it updates both affected users' decryption keys and ciphertexts (*re-encryption*) previously stored on it. Crucially, in this scheme, the re-encryption keys can also be used to update revoked keys, enabling them to decrypt data again. Whoever comes into pos-

session of the re-encryption keys can privately "undo" key revocations. This is a realistic threat that may occur whenever the curious cloud server comes into possession of a revoked key, or if a revoked user breaches the cloud server. A successful *revocation undoing attack* would lead to a breach of confidentiality. Indeed, it would allow an adversary to decrypt all the ciphertexts that the key was authorized to access, thus retrieving the sensitive or valuable information that was intended to be read only by authorized users.

In this paper, we propose a revocable KP-ABE scheme that enjoys all the advantages of the Yu et al.'s scheme but also ensures data confidentiality even if the cloud server comes into possession of a revoked key, thus guaranteeing *revocation undoing resistance.* The main idea is to distribute the task of updating affected users' decryption keys to both the cloud server and the users. That is, in our scheme a decryption key update is accomplished by a double update through two *different* re-encryption keys held by the cloud server and by the non-revoked users, and called *cloud re-encryption key* and *user re-encryption key*, respectively. In our construction, we show how the user re-encryption keys can be efficiently distributed only to non-revoked users. This assures that even if the cloud server comes into possession of a revoked key, it cannot undo the key revocation because it lacks the user re-encryption keys. We then formally prove that the proposed scheme is secure in the Selective-Set model under the Decisional Bilinear Diffie-Hellman assumption. Finally, we analyze storage, communication, and computational costs, and we perform simulations to show that in our scheme the user experiences a slightly higher computational cost with respect to the Yu et al's scheme [17]. Considering an IoT scenario, for example a smart city application like in [6,19], and assuming 8000 users equipped with Raspberry Pi boards and revocations occurring on average every six hours, a one-year-long simulation reveals that the average user experiences a load (in terms of computation time) 7.37 % higher than Yu et al.'s scheme. Also, the simulations show that our scheme is scalable with the number of users, i.e., the more users are in the system, and the less load introduced to each user.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces some background on ABE and other related techniques that we use in our scheme. Section 4 describes our model, assumptions, and construction. Section 5 analyzes the security of our scheme with formal proofs. Section 6 analyzes the cost of each scheme procedure and evaluates the performance through simulations. Section 7 concludes the paper.

2 Related Work

Plenty of ABE schemes have been introduced in the recent past. These schemes mainly aim at lowering the computational cost of encryption/decryption primitives either by outsourcing part of them to trusted parties [20–23], or by redefining them [24, 25], or by using new constructions [26]. Other schemes [14–18, 27–30] aim at adding a key revocation mechanism, which is essential in order for ABE to be practical and therefore adopted on a large scale. In this section,

we review some relevant indirect revocation schemes.

Indirect revocation can be further divided into two categories: (i) *user-wise* indirect revocation, and (ii) *attribute-wise* indirect revocation. In user-wise indirect revocation, the key authority can only revoke a *whole* decryption key associated with a user. Each user is given a decryption key and an *identifier*. During revocation, the key authority revokes an identifier, and then it issues key update material through which only non-revoked users' decryption keys are updated. In attribute-wise indirect revocation, the granularity of revocation is finer. That is, the key authority can revoke a subset of attributes embedded in a user's decryption key. This is desirable, for instance, if some user's access privileges change, and the key authority wants to provide the user with a more restrictive access policy. Notably, as in user-wise revocation, the key authority can always revoke a whole decryption key associated with a user.

Ref.	**Paradigm**	**Key authority goes online**	**Immediate Revocation**	**Revocation mode**	**Revocation undoing resistance**
[14]	KP-ABE	Periodically	No	User-wise	Yes
[15]	CP-ABE	Periodically	No	User-wise	Yes
[16]	CP-ABE	Periodically	No	User-wise	Yes
[27]	KP-ABE and CP-ABE	Periodically	No	User-wise	Yes
[28]	CP-ABE	Periodically	No	User-wise	Yes
[29]	CP-ABE	Only in case of revocation	Yes	User-wise	No
[17]	KP-ABE	Only in case of revocation	Yes	Attribute-wise	No
[18]	CP-ABE	Always	Yes	Attribute-wise	Yes
[30]	KP-ABE	Only in case of revocation	Yes	Attribute-wise	No
Our	KP-ABE	Only in case of revocation	Yes	Attribute-wise	Yes

Table 1: Comparison with existing indirect revocable ABE schemes.

Boldyreva et al. [14] proposed a user-wise indirect revocable KP-ABE scheme in which the key authority makes some key update material public. By using the information contained in the key update material, only the non-revoked users are capable of updating their decryption keys, while the revoked user is not. This scheme is slotted, i.e., the time is divided into epochs, and the key update process is performed by the key authority at the beginning of every new epoch. This means that the key authority must go online periodically to publish the update material, even if no key revocation has occurred in that epoch. Moreover, all the users must download the update material at each epoch, and this operation may be burdensome. Finally, a key cannot be promptly revoked after its compromise, but the key revocation comes into force only after the end of the current epoch. In [15] and [16], the authors extended the techniques of Boldyreva et al. [14] in the CP-ABE setting. These schemes delegate the majority of the workload to an untrusted cloud server. However, also in these schemes the key authority must go online at the beginning of every new epoch and a key cannot be promptly revoked after its compromise. Also in [27] and [28], the revocation approach is time slotted and at user level. Therefore, an immediate and fine-grained revocation cannot be reached. Differently from all these schemes, in our scheme a key revocation comes into force with immediate effect, and the key authority performs no operation as long as no decryption key must be revoked. In addition, since our revocation is performed at attribute level, only a portion of users are affected by each key revocation.

Yang et al. [18] proposed an attribute-wise indirect revocable CP-ABE scheme in which the key revocation has an immediate effect. However, to have his/her key updated, every affected user must establish a secure channel with the key authority. Therefore, the key authority must be always online because key update requests from users can happen anytime. On the contrary, in our scheme the key authority just stores secret quantities on the cloud server, and then it can go back offline.

In 2019, Ma et al. [29] proposed a user-wise indirect revocable CP-ABE scheme in which the revocation has an immediate effect. The cloud server holds secret quantities, called *cloud-side keys*, which it uses to re-encrypt the ciphertexts. The cloud server uses the cloud-side key to partially decrypt the ciphertext to be sent to the user. Crucially, if the cloud server comes into possession of a revoked key, it can use the cloud-side key to undo the revocation.

Hur and Noh [30] proposed an attribute-wise indirect revocable KP-ABE scheme in which the key revocation has an immediate effect. The cloud server holds secret quantities, called *attribute group keys*, which it distributes only to non-revoked users for key update purposes. Crucially, if the cloud server comes into possession of a revoked key, it can use the attribute group keys to undo the revocation.

Yu et al. [17] proposed an attribute-wise indirect revocable KP-ABE scheme in which the key authority goes online only at key revocation events, and the key revocation has an immediate effect. When a key must be revoked, the key authority performs a system-wide update of the attributes embedded in that key, it stores on the cloud server some secret quantities (re-encryption keys), and finally, it can go back offline. The cloud server updates affected users' decryption keys without learning anything of the decryption keys themselves. The disadvantage of this scheme is that the curious cloud server knows the re-encryption keys that are able to update also any revoked key. Our scheme, which is based on [17], introduces additional security with limited additional cost. In particular, it avoids that the cloud server in possession of a revoked key is able to undo a key revocation and, thus, authorize the revoked key to access the data again.

3 Background

3.1 KP-ABE: The GPSW Scheme

In KP-ABE, ciphertexts are labeled with a set of attributes, and decryption keys embed an access policy defined over some attributes. The *access policy* is a Boolean formula and can be expressed as a tree. If the access policy is satisfied by the attributes labeling the ciphertext, the decryption key is able to decrypt the ciphertext. The attributes present both in the ciphertext and in the decryption key evaluate to true, and the access policy is satisfied if the Boolean formula evaluates to true. To give an example, the access policy $A \,\text{AND}\, (E \,\text{OR}\, C)$ is satisfied by the set $\{A, E, F\}$, but it is not satisfied by the set $\{C, E, B\}$.

Let an access policy $\mathcal{T}$ be defined as a tree. Each node x of the tree is described by a threshold gate of type k_x-of-n_x, where k_x is its threshold value, and n_x is the number of child nodes of x. A generic k_x-of-n_x threshold gate evaluates to true, i.e., is satisfied, if at least k_x of its n_x children are satisfied. A threshold gate of type 1-of-n_x implements an OR gate, while a threshold gate of type n_x-of-n_x implements an AND gate. All the leaf nodes represent attributes and are described by a threshold value $k_x = 1$.

Let $\mathbb{G}_1$ be a multiplicative cyclic group whose group operation is efficiently computable. Let p be its prime order, and let g be a generator of $\mathbb{G}_1$. Let $e : \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_T$ be a bilinear map that is efficiently computable and has the properties of bilinearity and non-degeneracy [12].

The first KP-ABE scheme introduced in literature [12] (henceforth, the GPSW scheme) is defined by four primitives, namely GPSW.Setup, GPSW.Encrypt, GPSW.KeyGen, and GPSW.Decrypt.

(MK, PK) = GPSW.Setup($\mathcal{U}$). Defined the universe of the attributes $\mathcal{U} = \{1, \ldots, n\}$, choose $y, t_1, \ldots, t_n$ uniformly at random in $\mathbb{Z}_p$. Then compute $T_1 = g^{t_1}, \ldots, T_n = g^{t_n}$ and $Y = e(g, g)^y$. Output the public parameters as $PK = (Y, T_1, \ldots, T_n)$ and the master key as $MK = (y, t_1, \ldots, t_n)$.

CT = GPSW.Encrypt(M, γ, PK). To encrypt the message $M \in \mathbb{G}_T$ under the set of attributes γ (*encryption attributes*), choose a number s uniformly at random in $\mathbb{Z}_p$. Then compute $\hat{ct} = MY^s$, and for each attribute $i \in \gamma$ compute the *ciphertext component* $ct_i = T_i^s$. Output the *ciphertext* as $CT = (\hat{ct}, \{ct_i\}_{i \in \gamma})$.

DK = GPSW.KeyGen($\mathcal{T}, MK$). Let $\mathcal{T}$ be an access policy defined over a set of attributes λ (*access policy attributes*). The algorithm defines a random polynomial q_x of degree $deg_x = k_x - 1$ for each node x, starting from the root node and proceeding in a top-down manner. For the root node ρ, set $q_\rho(0) = y$ and other deg_ρ random points to define it completely. Then, associate each child node x with a unique number index(x). Proceed for any other node x, setting $q_x(0) = q_{\text{parent}(x)}(\text{index}(x))$ and other deg_x random points to define the polynomial completely. The polynomial for the leaf nodes are defined only by $q_x(0)$. For each leaf node, define the *decryption key component* $dk_i = g^{\frac{q_x(0)}{t_i}}$, where the node x identifies the attribute i. Output the *decryption key* as $DK = \{dk_i\}_{i \in \lambda}$.

M = GPSW.Decrypt(CT, DK). The algorithm proceeds in a bottom-up manner. For each attribute $i \in \gamma \cap \lambda$ compute $e(dk_i, ct_i) = e(g, g)^{q_x(0) \cdot s}$. Then, by polynomial interpolation at the exponent, compute $e(g, g)^{q_z(0) \cdot s}$, where z is the parent node. Proceeding in a bottom-up fashion, if the polynomial interpolation is feasible at each level of the access policy up to the root ρ, we obtain $e(g, g)^{q_\rho(0) \cdot s} = e(g, g)^{y \cdot s} = Y^s$. Output the message $M = \frac{\hat{ct}}{Y^s}$ if the access policy is satisfied, $\perp$ otherwise.

3.2 Proxy Re-Encryption for KP-ABE

Yu et al. [17] proposed a KP-ABE scheme (henceforth, the YWRL scheme) which extends the GPSW one to include a key revocation mechanism. This is

achieved by adding the features of *proxy re-encryption* and *key update*. Proxy re-encryption allows a third party to re-encrypt ciphertexts without accessing the content itself, thus an honest-but-curious cloud server perfectly fits this role. The re-encryption of ciphertexts stored on the cloud server prevents a revoked key from decrypting them. However, this mechanism alone is not sufficient: after the re-encryption, other legitimate keys may be unable to decrypt those ciphertext too. Hence, YWRL scheme implements also a key update mechanism for such keys, which is operated by the cloud server. For each decryption key, the cloud server is given all the decryption key components but one, the one related to a special attribute (*dummy attribute*, Att_D), which is ANDed at the root of every access policy. The dummy attribute is also present in every ciphertext. Without knowing the decryption key component dk_{Att_D}, the cloud server cannot decrypt any ciphertext, but it can perform key update.

In the YWRL scheme, every attribute i, except for the dummy attribute, is identified by a version number. The version of an attribute i is set to zero at setup time, and it is increased by one every time the attribute is updated. Re-encryption and key update are performed by means of *re-encryption keys* issued by the key authority. A re-encryption key $rk_{i^{(v)} \leftrightarrow i^{(v+1)}}$ is a cryptographic quantity able to update both a ciphertext component and a decryption key component from the version v to the version $v+1$. All the re-encryption keys for the attribute i are ordered and maintained in a dynamic structure called *re-encryption key list* RKL_i. When a new re-encryption key is issued, an element containing the latest re-encryption key is appended to the re-encryption key list. Note that the element at index v contains the re-encryption key to update either a ciphertext component or a decryption key component from the version $v-1$ to the version v. To create a new version of the attribute $i^{(v)}$, the key authority updates the component of the master key $t_i^{(v)}$ to $t_i^{(v+1)}$, and the component of the public parameters $T_i^{(v)}$ to $T_i^{(v+1)}$. Next, it issues a re-encryption key $rk_{i^{(v)} \leftrightarrow i^{(v+1)}}$ which can be used to transform either a ciphertext component $ct_i^{(v)}$ to $ct_i^{(v+1)}$ or a decryption key component $dk_i^{(v)}$ to $dk_i^{(v+1)}$. The scheme allows re-encryption and key update from *any* old version to the latest version. To simplify the notation, we avoid indicating the version of the attributes when is not necessary. We will use i to indicate a generic version of the attribute, and i' to indicate the next version of it. The primitives of YWRL scheme have the following syntax:

$(t'_i, T'_i, rk_{i \leftrightarrow i'}) = \text{YWRL.UpdateAtt}(i, MK)$. Choose t'_i uniformly at random in $\mathbb{Z}_p$ and compute $T'_i = g^{t'_i}$. Then compute the *re-encryption key* $rk_{i \leftrightarrow i'} = t'_i / t_i$. Update the component of the master key t_i to t'_i, and the component of the public parameters T_i to T'_i. Output the re-encryption key $rk_{i \leftrightarrow i'}$.

$ct_i^{(l)} = \text{YWRL.UpdateCT}(i, ct_i, RKL_i)$. Obtain the current version of the attribute i from ct_i and locate the entry associated with such a version in RKL_i. If the selected entry is not the last one, retrieve all the entries from the next one to the last one. Compute $rk_{i \leftrightarrow i^{(l)}} = rk_{i \leftrightarrow i'} \cdot rk_{i' \leftrightarrow i''} \cdots rk_{i^{(l-1)} \leftrightarrow i^{(l)}} = t_i^{(l)} / t_i$. Next, compute $ct_i^{(l)} = (ct_i)^{rk_{i \leftrightarrow i^{(l)}}} = g^{t_i^{(l)} \cdot s}$. Output the updated ciphertext

component $ct_i^{(l)}$.
$dk_i^{(l)} = \text{YWRL.UpdateDK}(i, dk_i, RKL_i)$. Obtain the current version of the attribute i from dk_i and locate the entry associated with such a version in RKL_i. If the selected entry is not the last one, retrieve all the entries from the next one to the last one. Compute $rk_{i \leftrightarrow i^{(l)}} = rk_{i \leftrightarrow i'} \cdot rk_{i' \leftrightarrow i''} \cdots rk_{i^{(l-1)} \leftrightarrow i^{(l)}} = t_i^{(l)}/t_i$. Next, compute $dk_i^{(l)} = (dk_i)^{(rk_{i \leftrightarrow i^{(l)}})^{-1}} = g^{\frac{q_x(0)}{t_i^{(l)}}}$. Output the updated decryption key component $dk_i^{(l)}$.

The YWRL scheme is vulnerable to the revocation undoing attack, which can be successfully executed by the cloud server. Let $DK^\star = \left(\{dk_i^{(v)}\}_{i \in \lambda}, dk_{Att_D}\right)$ be a revoked key, where version v is lower than the latest version l. If the cloud server comes into possession of $DK^\star$, it can run $dk_i^{(l)} = \text{YWRL.UpdateDK}(i, dk_i^{(v)}, RKL_i)$ on each decryption key component, thus obtaining an updated decryption key, which is able to decrypt data again. The revocation undoing attack can be performed on any decryption key. Notably, a new revocation of the same decryption key is useless since the cloud server can use the new re-encryption keys to perform the same attack again.

4 Proposed Scheme

Our main idea is to create a robust revocation mechanism that prevents the honest-but-curious cloud server from being able to undo key revocations, that is, to be resistant to the revocation undoing attack. To this aim, we use PRE techniques and distribute the task of keys and ciphertexts update to both the cloud server and the users. In particular, the update of either a decryption key component or a ciphertext component to a new version is accomplished by a double update through two different re-encryption keys, namely *cloud re-encryption key* (crk) and *user re-encryption key* (urk). The former is available only to the cloud server, the latter must be available only to non-revoked users. We now introduce a mechanism to distribute it only to them.

To exclude revoked users from obtaining user re-encryption keys, we encrypt them through GPSW encryption in such a way that only non-revoked users' keys are able to decrypt them. We can efficiently achieve this by means of a binary tree structure in which an attribute is assigned to every node, except for the root. Attributes assigned to leaf nodes are identities for current or future users, so the number of leaves determines the maximum number of users. In every user's decryption key is embedded a sub-policy called *update sub-policy* that is an OR between the attributes in the path of the binary tree from the leaf corresponding to that user to its first-level ancestor (a root's child). In the example of Fig. 1, the update sub-policy of the decryption key of user u_4 is $(x_1 \,\text{OR}\, x_4 \,\text{OR}\, u_4)$. An *update ciphertext* containing one or more urk's is labeled with the minimal set of nodes covering all the non-revoked users. We call this set NRAttrs. In the example of Fig. 1, to exclude the revoked user u_4 from being able to decrypt the update ciphertext, we encrypt the urk's with the attributes

in NRAttrs, i.e., $\{x_2, x_3, u_3\}$.

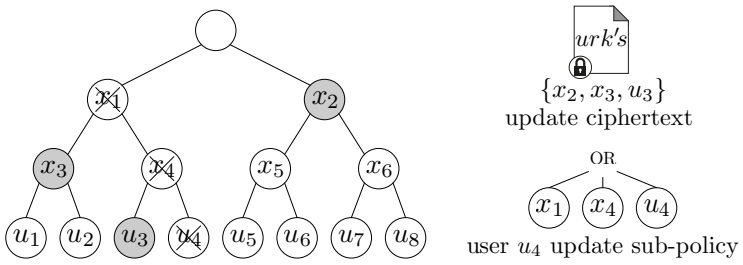


Figure 1: Example of binary tree used for distributing user re-encryption keys. User u_4 is revoked.

Note that none of these attributes is present in user u_4's update sub-policy, and therefore he/she cannot decrypt the update ciphertext. Moreover, note that every other non-revoked user has in his/her update sub-policy exactly one attribute in common with the NRAttrs set, and therefore each of them is able to decrypt the update ciphertext. When another user has to be revoked, we compute a new NRAttrs set, and we encrypt the new urk's with the attributes in the new NRAttrs set.

In the following, we describe the system model and the assumptions, and then we present our construction.

4.1 Model and Assumptions

Similar to [17], we assume a single entity to be responsible for key management and data production. In other words, we assume a single *data owner*, which also acts as *key authority*. Nevertheless, our scheme can be easily extended to split these roles and to foresee many data owners. To do this, it suffices a mechanism to distribute the new public parameters in authenticated fashion, as in [6]. We thus assume that the system is composed of the following parties: the data owner, many users, and a cloud server. In our system, the data owner produces the data, encrypts it, and stores it on the cloud server. Users are data consumers and hold decryption keys; they retrieve the data from the cloud server and decrypt it. The data owner can revoke a decryption key at any time. The cloud server is a server owned by some PaaS provider supplying storage and computational capabilities, that runs software on behalf of the data owner. We assume the cloud server to have generous storage space capacity and computational resources. Moreover, we assume the cloud server can ensure high availability and can be accessed at any time by the data owner and the users.

4.1.1 Security Model

Similarly to other works, e.g., [17, 18, 22], we assume the cloud server to be honest but curious, meaning that it honestly carries out all its designated tasks,

i.e., storage, communication, and computational tasks, but it is also interested in accessing the content of the encrypted data stored on it. Also, we assume the data owner to be fully trusted. On the contrary, we assume a user to be interested in accessing unauthorized data. To this aim, users and/or revoked users can collude by trying somehow to combine their decryption keys in order to access unauthorized data that each decryption key is unable to decrypt singly. Moreover, we assume that the data owner owns a public/private key pair to perform digital signature algorithms, e.g., ECDSA keys, and that each user owns a public/private key pair for asymmetric encryption, e.g., RSA keys. We further assume secure communication channels among all the communicating parties, which can be obtained by using some secure protocol, e.g., TLS.

4.1.2 Definitions and Notation

The data owner creates two different types of ciphertexts: (i) *data ciphertexts* CT_D, which are labeled with a set of encryption attributes γ and the dummy attribute Att_D, contain the actual data; and (ii) *update ciphertexts* $CT_U^{(r)}$, which are labeled with the attributes in NRAttrs and used for revocation purposes, contain the user re-encryption keys. An update ciphertext is parsed as $\left(\hat{ctu}, \{ctu_k\}_{k\in \mathrm{NRAttrs}}\right)^{(r)}$. The superscript notation (r) denotes a quantity related to the r-th occurred revocation.

The universe of attributes $\mathcal{U}$ can be logically divided into: (i) the *data sub-universe* $\mathcal{U}_D$ that includes all the attributes used to create data ciphertext; (ii) the *update sub-universe* $\mathcal{U}_U$ that includes all the attributes associated to binary tree nodes and used to create update ciphertexts; and (iii) the dummy attribute.

Each user is provided with a decryption key DK with an access policy $\mathcal{T}$ that can be logically divided into two sub-policies as shown in Fig. 2: (i) the *data sub-policy* $\mathcal{T}_D$ that, together with the dummy attribute, authorizes the user to decrypt data ciphertexts. It is defined over a set of attributes λ and is ANDed with the dummy attribute; and (ii) the *update sub-policy* that authorizes the user to decrypt update ciphertexts. Such a sub-policy is an OR between the attributes in the path of the binary tree from the leaf node corresponding to that user to its first-level ancestor.

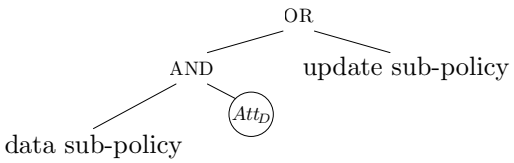


Figure 2: Structure of an access policy in our scheme.

Each user maintains a list for each attribute $i \in \lambda$ called $URKL_i$ (*user re-encryption key list*). Each element of this list is a user re-encryption key. Similarly, the cloud server maintains a list for each attribute $i \in \mathcal{U}_D$ called $CRKL_i$ (*cloud re-encryption key list*). Each element of this list is a cloud re-encryption

key. The cloud server must not have access to the user re-encryption keys, but it must keep track of which user re-encryption keys are contained in an update ciphertext. To do that, the cloud server maintains a list called CT_UL which contains all the update ciphertexts. Finally, the cloud server maintains a list for each attribute $i \in \mathcal{U}_D$ called PKL_i (*public parameter list for the attribute* i). Each element of this list is a component of the public parameters.

The data owner maintains a local copy of $CRKL_i$, $URKL_i$, and PKL_i for all attributes in the data sub-universe. The data owner also maintains a binary tree BT, a revocation list rl, and the set of attributes NRAttrs. The revocation list contains all the leaf nodes of the binary tree that identify users which have been revoked, as well as all the leaf nodes not yet assigned to any user. In other words, we consider a future user that has not joined yet the system as "revoked". This avoids that a new user can decrypt update ciphertexts produced before his/her joining time. To determine the NRAttrs set, we use a function called FindSet which takes as input the binary tree and the revocation list and outputs the minimal set of nodes NRAttrs that covers all the leaf nodes not in the revocation list. Finally, we define a primitive called UpdateAtt which allows us to generate cloud and user re-encryption keys.
$(t'_i, T'_i, crk_{i\leftrightarrow i'}, urk_{i\leftrightarrow i'}) = \text{UpdateAtt}(i, MK)$. Given an attribute $i \in \mathcal{U}_D$, choose α_i, β_i uniformly at random in $\mathbb{Z}_p$ and compute $t'_i = \alpha_i \cdot \beta_i$ and $T'_i = g^{t'_i}$. Then compute the *cloud re-encryption key* $crk_{i\leftrightarrow i'} = \alpha_i/t_i$ and set the *user re-encryption key* $urk_{i\leftrightarrow i'} = \beta_i$. Update the component of the master key t_i to t'_i, and the component of the public parameters T_i to T'_i. Output the cloud re-encryption key $crk_{i\leftrightarrow i'}$ and the user re-encryption key $urk_{i\leftrightarrow i'}$.

4.2 Procedures

In the following, we describe the procedures of our scheme, namely *Setup*, *User Join*, *Data Production*, *Key Revocation*, and *Data Consumption*.
Setup. This procedure initializes the scheme. The data owner chooses a maximum number of users N and creates a binary tree BT of height $\lceil \log_2(N) \rceil$. Next, it chooses the universe of the attributes $\mathcal{U} = \mathcal{U}_D \cup \mathcal{U}_U \cup \{Att_D\}$ and assigns each element of $\mathcal{U}_U$ to a node of the binary tree except the root node. Then, the data owner executes the GPSW.Setup($\mathcal{U}$) primitive which outputs the master key MK and the public parameters PK. The data owner keeps the master key MK secret, while it stores the public parameters PK on the cloud server. Finally, the data owner initializes a revocation list rl containing all the leaf nodes of the binary tree.
User Join. This procedure provides a new joining user with a decryption key DK. First, the data owner selects the left-most leaf node u_k of the binary tree not assigned to any user and removes it from the revocation list. If all the leaf nodes have already been assigned to users, the user cannot be added to the system. Next, it creates the update sub-policy, which is defined over the set of attributes π from the leaf node u_k to its first-level ancestor. Then, the data owner chooses the data sub-policy $\mathcal{T}_D$ for the user, which is defined over a set of

attributes λ. Finally, it structures the access policy $\mathcal{T}$ as shown in Fig. 2, and it executes GPSW.KeyGen$(\mathcal{T}, MK)$ to create the decryption key DK.

For each $i \in \lambda$, the data owner computes $e(dk_i, T_i)$; this result will be used by the user to verify the correct update of decryption key components. The data owner composes the message $\left(\{e(dk_i, T_i), URKL_i\}_{i\in\lambda}, DK\right)$, signs it, and encrypts it with the user's public key. Finally, it sends to the cloud server this encrypted message together with the attributes in π and all the decryption key components $\{dk_i\}_{i\in\lambda}$. Note that the decryption key components related to the dummy attribute and to the attributes in the update sub-policy are not sent to the cloud server. Upon receiving this message, the cloud server stores $(\pi, \{dk_i\}_{i\in\lambda})$ and sends the encrypted message to the user. The user decrypts it with his/her private key and retrieves the decryption key DK as well as $\{e(dk_i, T_i)\}$ and $\{URKL_i\}$ for all $i \in \lambda$.

Data Production. This procedure is executed when the data owner produces a new *data* D and wants to make it available to the users. First, the data owner signs the data D, and next, it chooses a unique *data identifier* DID. Then, the data owner chooses a set of encryption attributes $\gamma \in \mathcal{U}_D$, and it executes GPSW.Encrypt$(D|\text{Sign}(D), \gamma \cup \{Att_D\}, PK)$ to generate the data ciphertext CT_D. Finally, the data owner stores the couple (DID, CT_D) on the cloud server.

Key Revocation. This procedure is started by the data owner to revoke a decryption key $DK^\star$ which has been compromised. The procedure is divided into two phases. The first one is called *attributes update phase* and prevents $DK^\star$ from decrypting ciphertexts produced after the key revocation. The second phase is called *components update phase* and includes both the re-encryption of old ciphertexts and the update of affected users' decryption keys. While the former phase is immediately executed after the key compromise, the latter is distributed over time in a lazy fashion upon data requests by affected users. The messages exchanged between the entities during attribute update phase and data consumption procedure are depicted in Fig. 3.

Attributes Update Phase. This phase is started by the data owner to revoke a decryption key of a user $u_\star$. Let $DK^\star$ be the decryption key to be revoked and $\lambda^\star$ the data sub-policy attributes of such a key. The data owner selects a minimum set of attributes $\mu^\star \subseteq \lambda^\star$ without which the data sub-policy $\mathcal{T}_D^\star$ will never be satisfied (*updatee set*). For each $i \in \mu^\star$, the data owner executes UpdateAtt(i, MK) and updates the component of the master key t_i to t_i' and the component of the public parameters T_i to T_i'. The primitive outputs the cloud re-encryption key $crk_{i\leftrightarrow i'}$ and the user re-encryption key $urk_{i\leftrightarrow i'}$. The data owner creates an update ciphertext $CT_U^\star$ which contains all the user re-encryption keys in the updatee set and cannot be decrypted by any revoked user. To do that, the data owner first adds $u_\star$ to the revocation list rl and executes FindSet(BT, rl) to obtain the new NRAttrs set; next, it signs $\{urk_{i\leftrightarrow i'}\}_{i\in\mu^\star}$ and executes GPSW.Encrypt$(\{urk_{i\leftrightarrow i'}\}_{i\in\mu^\star}|\text{Sign}(\{urk_{i\leftrightarrow i'}\}_{i\in\mu^\star}), \text{NRAttrs}, PK)$. Then, the data owner signs $\{T_i'\}_{i\in\mu^\star}$. Finally, it composes the message $(\{crk_{i\leftrightarrow i'}\}_{i\in\mu^\star}$, $CT_U^\star, \{T_i'\}_{i\in\mu^\star}, \text{Sign}(\{T_i'\}_{i\in\mu^\star}))$ –message m_1 in Fig. 3– and sends it to the cloud server. Upon receiving the message m_1 from the data owner, the cloud server stores each quantity as last element of the related list.

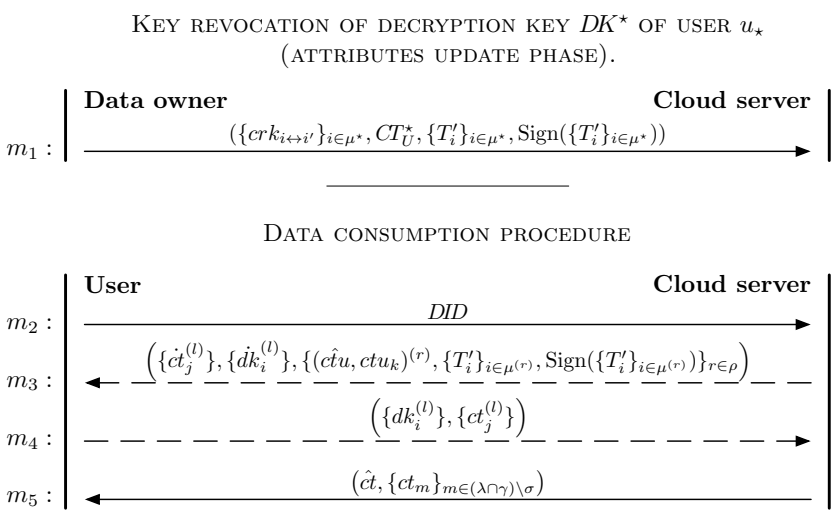


Figure 3: Messages exchanged during attribute update phase and data consumption procedure. Components update phase (dashed arrows) is executed only if ciphertext or decryption key components are outdated.

Components Update Phase. This phase is executed during each data consumption procedure. Let a data ciphertext requested by a legitimate user be CT_D and its identifier be *DID*, and let the decryption key of such user be *DK*. During this phase, the cloud server updates the decryption key components of *DK* and the ciphertext components of CT_D as follows. For each $i \in \lambda \cap \gamma$, the cloud server first determines whether the decryption key component and/or the ciphertext component are *outdated* by checking their version against the latest one. For each outdated attribute $i \in \lambda$, the cloud server executes YWRL.UpdateDK$(i, dk_i, CRKL_i)$ to obtain a *partially updated* decryption key component $\dot{dk}_i^{(l)}$. For each outdated attribute $j \in \sigma \subseteq \lambda \cap \gamma$, the cloud server executes YWRL.UpdateCT$(j, ct_j, CRKL_j)$ to obtain a *partially updated* ciphertext component $\dot{ct}_j^{(l)}$.

Data Consumption. This procedure is started by a user who wants to access a piece of data. To this aim, the user requests a data ciphertext to the cloud server by specifying the data identifier *DID* (message m_2). First, the cloud server retrieves the couple (DID, CT_D) and executes the components update phase, thus obtaining the partially updated decryption key components and the partially updated ciphertext components. Next, it determines the set of key revocations ρ –occurred since the user's last data consumption execution– in which he/she was an affected user. Then, for each revocation $r \in \rho$, the cloud server retrieves the update ciphertext $CT_U^{(r)}$ from CT_UL, parses it as $\left(\hat{ctu}, \{ctu_k\}_{k\in\mathrm{NRAttrs}}\right)^{(r)}$, and selects only the ciphertext component $ctu_k \mid k \in \pi \cap \mathrm{NRAttrs}$. Finally, the cloud server sends the message m_3 $(\{\dot{ct}_j^{(l)}\}, \{\dot{dk}_i^{(l)}\}, \{(\hat{ctu}, ctu_k)^{(r)}, \{T_i'\}_{i\in\mu^{(r)}},$

$\text{Sign}(\{T'_i\}_{i\in\mu^{(r)}})\}_{r\in\rho})$ to the user, where i refers to *outdated attributes* in the decryption key, and j refers to *outdated attributes* in the data ciphertext *and* present in the decryption key, i.e., $j \in \sigma$. Note that the cloud server sends only the ciphertext component corresponding to an attribute that is present in the update sub-policy of the decryption key. This optimization saves communication overhead.

Upon receiving the message m_3 from the cloud server, for each $r \in \rho$, the user verifies the data owner's signature on the public key components, executes GPSW.Decrypt$((\hat{ctu}, ctu_k)^{(r)}, DK)$, thus retrieving the user re-encryption keys $\{urk_{i\leftrightarrow i'}\}_{i\in\mu^{(r)}}$ and the signature on them. The user verifies the signature and appends each $urk_{i\leftrightarrow i'} \mid i \in \lambda$ to the related $URKL_i$. When the user re-encryption key list $URKL_i$ is updated to the last version, the user selects it together with the received partially updated decryption key component $\dot{dk}_i^{(l)}$, and executes YWRL.UpdateDK$(i, \dot{dk}_i^{(l)}, URKL_i)$, thus obtaining the *updated* decryption key component $dk_i^{(l)}$. The user verifies that the update of the decryption key components has been performed correctly by checking that $e\left(dk_i, T_i\right) = e\left(dk_i^{(l)}, T_i^{(l)}\right)$. In addition, for each attribute j, the user selects the user re-encryption key list $URKL_j$ and the received partially updated ciphertext component $\dot{ct}_j^{(l)}$, and executes YWRL.UpdateCT$(j, \dot{ct}_j^{(l)}, URKL_j)$, thus obtaining the *updated* ciphertext component $ct_j^{(l)}$. Then, the user composes and sends the message m_4 $\left(\{dk_i^{(l)}\}, \{ct_j^{(l)}\}\right)$ to the cloud server.

Upon receiving the message m_4 from the user, the cloud server verifies that each ciphertext component ct_j has been correctly updated by the user. To do that, the cloud server checks that $e\left(ct_j, T_j^{(l)}\right) = e\left(ct_j^{(l)}, T_j\right)$. If all the comparisons are correct, the cloud server updates the data ciphertext CT_D with the ciphertext components $\{ct_j^{(l)}\}_{j\in\sigma}$, and the decryption key components for the user with the received ones. Then, the cloud server composes the message m_5 $\left(\hat{ct}, \{ct_m\}_{m\in(\lambda\cap\gamma)\setminus\sigma}\right)$ and sends it to the user. Note that the cloud server sends only the ciphertext components that were already up to date *and* present in the user's decryption key. This optimization saves communication overhead.

Upon receiving the message m_5 from the cloud server, the user composes the data ciphertext $CT_D = (\hat{ct}, \{ct_j^{(l)}\}_{j\in\{\lambda\cap\gamma\}})$ and decrypts it by executing GPSW.Decrypt(CT_D, DK), thus retrieving the data D and the signature on it. Finally, the user verifies the signature, and, if everything is correct, accepts the message.

5 Security Analysis

In this section, we provide security proofs for our scheme. Our aim is to show that our scheme is not less secure than Goyal et al. [12] scheme (GPSW scheme), which has been proved secure in the Selective-Set model under the Decisional Bilinear Diffie-Hellman (DBDH) assumption. We first define a game (Game 1)

which models a chosen-plaintext attack by a group composed of (i) colluding users whose keys cannot decrypt a given ciphertext and (ii) colluding users whose keys have been revoked. We formally prove that our scheme is resistant against such an adversary under the DBDH assumption. Then, we define a second game (Game 2) which models a chosen-plaintext attack by a cloud server that comes into possession of any quantity of revoked keys. We formally prove that our scheme is resistant against such an adversary under the DBDH assumption. Note that, in order to achieve chosen-ciphertext security, we can apply an efficient random oracle technique such as the Fujisaki-Okamoto transformation [31].

Game 1

Init The adversary declares the universe of attributes $\mathcal{U} = \mathcal{U}_D \cup \mathcal{U}_U \cup \{Att_D\}$, the challenge set $\Gamma = \gamma \cup \{Att_D\}$ with $\gamma \subseteq \mathcal{U}_D$, the number of key revocations n, and the data sub-policies of the decryption keys to revoke, i.e., $\mathcal{T}_D^{(j)}, \forall j \in [1, n]$. For each key revocation, the adversary also chooses an attribute $u_i \in \mathcal{U}_U$, which is the identity of the user to revoke.
Setup The challenger creates a binary tree BT with the attributes in $\mathcal{U}_U$, and it runs the GPSW.Setup primitive, thus creating the public parameters PK_0. Next, for each key revocation j, the challenger determines the updatee set μ_j from $\mathcal{T}_D^{(j)}$ and runs the UpdateAtt primitive for each attribute $i \in \mu_j$, thus creating the user re-encryption keys $\{urk_{i^{(j-1)} \leftrightarrow i^{(j)}}\}_{i \in \mu_j}$. Then, the challenger creates the public parameters PK_j, determines the NRAttrs set, and encrypts the user re-encryption keys in the updatee set by executing GPSW.Encrypt$(\{urk_{i^{(j-1)} \leftrightarrow i^{(j)}}\}_{i \in \mu_j}, \text{NRAttrs}, PK_j)$. Finally, the challenger gives the $n + 1$ sets of the public parameters, i.e., $\{PK_0, \ldots, PK_n\}$, and the update ciphertexts to the adversary.
Phase 1 The adversary is allowed to issue queries for: (i) decryption keys of any version whose access policy $\mathcal{T}_D$ is not satisfied by the challenge set; and (ii) decryption keys with data sub-policy equal to any $\mathcal{T}_D^{(j)}$, but version less than j. The *version* $v \in [0, n]$ specifies which master key the challenger uses to create the decryption key requested.
Challenge The adversary submits two distinct messages m_0 and m_1 of equal length. The challenger sets $b \leftarrow \{0, 1\}$ uniformly at random, runs the GPSW.Encrypt primitive on m_b with the challenge set, and sends the ciphertext $CT^\star$ to the adversary.
Phase 2 Phase 1 is repeated.
Guess The adversary outputs a guess $b^\star$ of b.

The advantage of an adversary $\mathcal{A}$ in this game is defined as $\Pr[b^\star = b] - \frac{1}{2}$.

Proof of Security

We prove that no Probabilistic Polynomial-Time (PPT) adversary can play Game 1 against our scheme with a non-negligible advantage under the DBDH assumption. To do this, we prove that breaking our scheme reduces to breaking

the GPSW scheme, which in turn is proved to be infeasible for a PPT adversary (see [12]). More formally, we state the following:

Theorem 1. *If a PPT adversary can play Game 1 against our scheme with a non-negligible advantage, then a simulator can be built to play the Selective-Set game (Game 0) against the GPSW scheme with the same advantage.*

Proof. Suppose there exists a PPT adversary $\mathcal{A}$ able to win Game 1 against our scheme with advantage ϵ. We build a simulator $\mathcal{B}$ that plays Game 0 against the GPSW scheme with the same advantage ϵ.

Init The simulator $\mathcal{B}$ starts Game 1 against the adversary $\mathcal{A}$. $\mathcal{A}$ chooses the universe of attributes $\mathcal{U}$, the challenge set $\Gamma = \gamma \cup \{Att_D\}$ with $\gamma \subseteq \mathcal{U}_D$, the number of key revocations n, the data sub-policies of the decryption keys to revoke, i.e., $\mathcal{T}_D^{(j)}, \forall j \in [1, n]$, and for each key revocation the identity $u_i \in \mathcal{U}_U$ of the user to revoke.

Setup In this phase, the simulator $\mathcal{B}$ builds an alternative universe of attributes $\mathcal{U}'$ and an alternative challenge set Γ', which will be used in Game 0 against the GPSW scheme. The alternative universe will be composed of an attribute i_0 for each attribute i in the original universe, plus an attribute i_j for each attribute i in the updatee set of each key revocation j. In formulas: $\mathcal{U}' = \{i\_0 \mid i \in \mathcal{U}\} \cup \{i\_j \mid i \in \mu_j, \forall j \in [1, n]\}$. Note that the alternative universe contains the all the attributes in $\mathcal{U}_U$ and the dummy attribute Att_D. Since these attributes are not subject to versioning and are never included in an updatee set, the alternative universe contains only one version of them, i.e., i_0.

To simplify the creation of the alternative challenge set, we define the following function:
$i\_x \leftarrow \text{FindCurrVer}(i, \mathcal{U}', v)$ This function takes as input an attribute named $i \in \mathcal{U}$, the universe of attributes $\mathcal{U}'$, and a version number v. The function outputs the attribute i_x, where $x = \max_{x \leq v}\{x \mid i\_x \in \mathcal{U}'\}$.

The alternative challenge set will be composed of the outputs of $\text{FindCurrVer}(i, \mathcal{U}', n)$ for each attribute i in the original challenge set. In formulas: $\Gamma' = \{\text{FindCurrVer}(i, \mathcal{U}', n) \mid i \in \Gamma\}$. The simulator $\mathcal{B}$, which acts as adversary in Game 0, selects the universe of attributes $\mathcal{U}'$ and the challenge set Γ', and runs Game 0 Init phase. The challenger replies with the public parameters $PK' = \{T_{i\_v} \mid i\_v \in \mathcal{U}'\}$. The simulator creates the $n+1$ sets of public parameters $\{PK_0, \ldots, PK_n\}$ for $\mathcal{A}$ as follows: it sets $PK_0 = \{T_{i\_0} \mid i \in \mathcal{U}\}$; next, for $j \in [1, n]$, it sets $PK_j = \{T_{i\_j} \in PK_{j-1} \mid i \notin \mu_j\} \cup \{T_{i\_j} \in PK' \mid i \in \mu_j\}$. For each key revocation j, the simulator $\mathcal{B}$ selects the identity corresponding to they key to be revoked and adds it to the revocation list rl. Then, it executes $\text{FindSet}(\text{BT}, rl)$ thus retrieving the new NRAttrs set, selects $|\mu_j|$ random elements in $\mathbb{Z}_p$ as the user re-encryption keys, and executes $\text{GPSW.Encrypt}(\{urk_{i^{(j-1)} \leftrightarrow i^{(j)}}\}_{i \in \mu_j}, \text{NRAttrs}, PK_j)$ to create the update ciphertexts. Finally, the simulator $\mathcal{B}$ gives the $n+1$ sets of public parameters and the update ciphertexts to the adversary $\mathcal{A}$.

Phase 1 For each user u_i's decryption key with data sub-policy $\mathcal{T}_D$ of version v that the adversary $\mathcal{A}$ requests, the simulator $\mathcal{B}$ first determines the update sub-policy by ORing the attributes in the path of the binary tree from the leaf

corresponding to the user u_i to its first-level ancestor. Then, the simulator computes the corresponding policy $\mathcal{T}'$ in the alternative universe by replacing each attribute $i \in \mathcal{T}$ with the attribute $i\_x \leftarrow \text{FindCurrVer}(i, \mathcal{U}', v)$. Finally, the simulator makes a decryption key request to the GPSW scheme and forwards the obtained decryption key to the adversary $\mathcal{A}$.

Challenge The simulator $\mathcal{B}$ receives the messages m_0 and m_1 from the adversary $\mathcal{A}$, selects them as its own challenge messages with respect to the GPSW scheme, and runs Game 0 Challenge phase. The simulator propagates the obtained ciphertext $CT^\star$ to the adversary $\mathcal{A}$.

Phase 2 The simulator acts as it did in Phase 1.

Guess The simulator $\mathcal{B}$ receives the guess $b^\star$ from the adversary $\mathcal{A}$ and selects it as its own guess with respect to the GPSW scheme.

As shown, the simulator can map any attribute of any version of our scheme onto an attribute of the GPSW scheme, and therefore it can simulate the versioning of the attributes with the adversary $\mathcal{A}$. The simulator translates the attributes from the universe $\mathcal{U}$ to $\mathcal{U}'$ and maps each decryption key request from $\mathcal{A}$ to an equivalent request compliant with the GPSW scheme. Note that the decryption keys, as well as all the other quantities, returned by the simulator are indistinguishable from those generated by our scheme. Hence, the advantage of the simulator $\mathcal{B}$ in Game 0, i.e., in the Selective-Set game, against the GPSW scheme is the same as the one of the adversary $\mathcal{A}$ in Game 1 against our scheme. ■

Game 2

Init The adversary declares the universe of attributes $\mathcal{U} = \mathcal{U}_D \cup \mathcal{U}_U \cup \{Att_D\}$, the challenge set $\Gamma = \gamma \cup \{Att_D\}$ with $\gamma \subseteq \mathcal{U}_D$, the number of key revocations n, and the data sub-policies of the decryption keys to revoke, i.e., $\mathcal{T}_D^{(j)}, \forall j \in [1, n]$. For each key revocation, the adversary also chooses an attribute $u_i \in \mathcal{U}_U$, which is the identity of the user to revoke. Note that the universe of attributes, the challenge set, and the access policies include the dummy attribute Att_D.

Setup The challenger creates a binary tree BT with the attributes in $\mathcal{U}_U$, and it runs the GPSW.Setup primitive, thus creating the public parameters PK_0. Next, for each key revocation j, the challenger determines the updatee set μ_j from $\mathcal{T}_D^{(j)}$ and runs the UpdateAtt primitive for each attribute $i \in \mu_j$, thus creating the cloud re-encryption keys $\{crk_{i^{(j-1)} \leftrightarrow i^{(j)}}\}$ and the user re-encryption keys $\{crk_{i^{(j-1)} \leftrightarrow i^{(j)}}\}$. Then, the challenger creates the public parameters PK_j, determines the NRAttrs set, and encrypts the user re-encryption keys in the updatee set by executing $\text{GPSW.Encrypt}(\{urk_{i^{(j-1)} \leftrightarrow i^{(j)}}\}_{i \in \mu_j}, \text{NRAttrs}, PK_j)$. Finally, the challenger gives the $n + 1$ sets of the public parameters, the cloud re-encryption keys, and the update ciphertexts to the adversary.

Phase 1 The adversary is allowed to issue queries for: (i) *incomplete decryption keys*, i.e., decryption keys which do not include the decryption key component relative to the update sub-policy and to the dummy attribute (dk_{Att_D}); and (ii) decryption keys with data sub-policy equal to any $\mathcal{T}_D^{(j)}$, but version less

than j.

Challenge The adversary submits two distinct messages m_0 and m_1 of equal length. The challenger sets $b \leftarrow \{0,1\}$ uniformly at random, runs the GPSW.Encrypt primitive on m_b with the challenge set, and sends the ciphertext $CT^\star$ to the adversary.

Phase 2 Phase 1 is repeated.

Guess The adversary outputs a guess $b^\star$ of b.

The advantage of an adversary $\mathcal{A}$ in this game is defined as $\Pr[b^\star = b] - \frac{1}{2}$.

Proof of Security

We prove that no PPT adversary can play Game 2 against our scheme with a non-negligible advantage under the DBDH assumption. To do this, we prove that breaking our scheme with Game 2 rules reduces to breaking our scheme with Game 1 rules, which we just proved to be infeasible for a PPT adversary (see Theorem 1). More formally, we state the following:

Theorem 2. *If a PPT adversary can play Game 2 against our scheme with a non-negligible advantage, then a simulator can be built to play Game 1 against our scheme with the same advantage.*

Proof. Suppose there exists a PPT adversary $\mathcal{A}$ able to win Game 2 against our scheme with advantage ϵ. We build a simulator $\mathcal{B}$ that plays Game 1 against our scheme with the same advantage ϵ.

Init The simulator $\mathcal{B}$ starts Game 2 against the adversary $\mathcal{A}$. $\mathcal{A}$ chooses the universe of attributes $\mathcal{U}$ which includes the dummy attribute Att_D, the challenge set $\Gamma = \gamma \cup \{Att_D\}$ with $\gamma \subseteq \mathcal{U}_D$, the number of key revocations n, the data sub-policies of the decryption keys to revoke, i.e., $\mathcal{T}_D^{(j)}, \forall j \in [1,n]$, and for each key revocation the identity $u_i \in \mathcal{U}_U$ of the user to revoke.

Setup In this phase, the simulator $\mathcal{B}$ adds another dummy attribute Att'_D to the universe of attributes, selects Γ as its challenge set, $\mathcal{T}_D^{(j)}, \forall j \in [1,n]$ as the access policies of the decryption keys to revoke, and runs Game 1 Init phase. The challenger replies with the sets of public parameters $\{PK_0, \ldots, PK_n\}$ and the update ciphertexts. The simulator removes the component of the public parameters relative to the dummy attribute Att'_D in every set of public parameters. Then, it chooses $\sum_j \mid \mu_j \mid$ random numbers in $\mathbb{Z}_p$ to create all the cloud re-encryption keys. Finally, the simulator gives the $n+1$ sets of public parameters, the cloud re-encryption keys, and the update ciphertexts to the adversary $\mathcal{A}$.

Phase 1 For each revoked decryption key with data sub-policy $\mathcal{T}_D^{(j)}$ and version $v < j$ that the adversary $\mathcal{A}$ requests, the simulator $\mathcal{B}$ makes a decryption key request to the Game 1 challenger and forwards the obtained decryption key to the adversary.

For each incomplete decryption key with policy $\mathcal{T}_D$ and version v that the adversary $\mathcal{A}$ requests, the simulator substitutes the dummy attribute Att_D with the new dummy attribute Att'_D and makes a decryption key request to the

Game 1 challenger. Then, the simulator removes the component $dk_{Att'_D}$ and the components relative to the update sub-policy from the obtained decryption key and sends such an incomplete decryption key to the adversary.
Challenge The simulator $\mathcal{B}$ receives the messages m_0 and m_1 from the adversary $\mathcal{A}$, selects them as its own challenge messages and runs Game 1 Challenge phase. The simulator propagates the obtained ciphertext $CT^\star$ to the adversary $\mathcal{A}$.
Phase 2 Phase 1 is repeated.
Guess The simulator $\mathcal{B}$ receives the guess $b^\star$ from the adversary $\mathcal{A}$, selects it as its own guess, and runs Game 1 Guess phase.

As shown, the simulator can accommodate every decryption key request from the adversary. Notably, the simulator is able to provide the adversary with *any* incomplete decryption key. Indeed, during Phase 1, it may happen that the adversary requests incomplete decryption keys relative to decryption keys whose access policy is satisfied by the challenge set. In such a case, by substituting Att_D with Att'_D, the simulator ensures that the access policy is not satisfied by the challenge set and that the request will be compliant with those accepted by the Game 1 challenger. Note that the decryption keys, as well as all the other quantities, returned by the simulator are indistinguishable from those generated by our scheme. Therefore, the advantage of the simulator $\mathcal{B}$ in Game 1 against our scheme is the same as the one of the adversary $\mathcal{A}$ in Game 2 against our scheme. ■

6 Performance Evaluation

6.1 Cost Analysis

In this section, we compare in detail our scheme against the YWRL scheme. We first analyze the computational cost of the various procedures in terms of pairing-based operations, namely (i) operation in $\mathbb{G}_1$ (i.e., point-scalar multiplication), (ii) operation in $\mathbb{G}_T$ (i.e., modular exponentiation), and (iii) bilinear pairing. Table 2 shows the complexity of the various procedures executed by each entity in both schemes. The cost of the user join procedure for the data owner is slightly higher in our scheme than in the YWRL scheme, namely, $|\lambda|$ pairings. This is necessary to allow the user to verify that the cloud server partially updated his/her key correctly. Notably, the computational cost of the data production procedure, which is the most frequently executed by the data owner, is the same both in the YWRL scheme and in ours. As for the cloud server, the additional computational cost in our scheme is at most $|\gamma|$ pairings during the data consumption procedure. This is necessary to verify that the user updated the ciphertext correctly. As for the user, the additional computational cost in our scheme is only present in case of components update. In such a case, the additional cost is due to the decryption of the update ciphertexts ($|\rho|$ pairings and $|\rho|$ operations in $\mathbb{G}_T$) and to the update of decryption key and ciphertext (up to $|\lambda| + |\gamma|$ operations in $\mathbb{G}_1$). Considering that key revocation

Data owner

Procedure	Scheme	Bilinear Pairings	Operations in $\mathbb{G}_1$	Operations in $\mathbb{G}_T$
Setup	Our	1	$\lvert\mathcal{U}_D\rvert + 1$	–
	YWRL	1	$\lvert\mathcal{U}_D\rvert + \lvert\mathcal{U}_U\rvert + 1$	–
User Join	Our	$\lvert\lambda\rvert$	$\lvert\lambda\rvert + \lvert\pi\rvert$	–
	YWRL	–	$\lvert\lambda\rvert + 1$	–
Data Production	Our	–	$\lvert\gamma\rvert + 1$	1
	YWRL	–	$\lvert\gamma\rvert + 1$	1
Key Revocation	Our	–	$\lvert\mu\rvert + \lvert\text{NRAttrs}\rvert$	1
	YWRL	–	$\lvert\mu\rvert$	–

Cloud server

Procedure	Scheme	Bilinear Pairings	Operations in $\mathbb{G}_1$	Operations in $\mathbb{G}_T$
Data Consumption (with components update)	Our	$2\lvert\gamma\rvert$	$\lvert\lambda\rvert + \lvert\gamma\rvert$	–
	YWRL	–	$\lvert\lambda\rvert + \lvert\gamma\rvert$	–

User

Procedure	Scheme	Bilinear Pairings	Operations in $\mathbb{G}_1$	Operations in $\mathbb{G}_T$
Data Consumption (without components update)	Our	$\lvert\lambda\rvert + 1$	–	$\lvert\lambda\rvert + 1$
	YWRL	$\lvert\lambda\rvert + 1$	–	$\lvert\lambda\rvert + 1$
Data Consumption (with components update)	Our	$2\lvert\lambda\rvert + \lvert\rho\rvert + 1$	$\lvert\lambda\rvert + \lvert\gamma\rvert$	$\lvert\lambda\rvert + \lvert\rho\rvert + 1$
	YWRL	$2\lvert\lambda\rvert + 1$	–	$\lvert\lambda\rvert + 1$

Table 2: Computational costs comparison.

is a far less frequent event than data consumption and data production, and considering also that only a portion of users is affected by a key revocation, the additional cost introduced by our scheme should be acceptable in order to provide the additional security properties.

Table 3 shows a comparison of the size of the cryptographic information held by each entity. Symbols $|\mathbb{G}_1|$, $|\mathbb{G}_T|$, and $|\mathbb{Z}_p|$ refer to the size in bit of a $\mathbb{G}_1$, a $\mathbb{G}_T$, and a $\mathbb{Z}_p$ element, respectively. "Yes" means that the entity stores the whole information and "No" means that it does not store the information. The size is specified if the entity stores only a portion of such information. We note that, in terms of storage, the schemes are similar. The additional storage cost for the user in our scheme is due to the update sub-policy in the decryption key and to the user re-encryption keys. As for the data owner, the additional cost is due to the attributes of the update sub-universe and to the user re-encryption keys.

Table 4 shows a comparison of the communication cost for key revocation and data consumption procedures. The additional cost of message m_1 is all due to the update ciphertext. However, the major communication cost of our scheme is introduced with messages m_3 and m_4, when the cloud server sends the user the update ciphertext containing the user re-encryption keys, and the user replies with the updated ciphertext components.

	Scheme	Size	Data owner	Cloud server	User
PK	Our	$(\lvert\mathcal{U}_D\rvert + \lvert\mathcal{U}_U\rvert + 1)\lvert\mathbb{G}_1\rvert + \lvert\mathbb{G}_T\rvert$	Yes	$\lvert\mathcal{U}_D\rvert\lvert\mathbb{G}_1\rvert$	$\lvert\lambda\rvert\lvert\mathbb{G}_1\rvert$
	YWRL	$(\lvert\mathcal{U}_D\rvert + 1)\lvert\mathbb{G}_1\rvert + \lvert\mathbb{G}_T\rvert$	Yes	$\lvert\mathcal{U}_D\rvert\lvert\mathbb{G}_1\rvert$	$\lvert\lambda\rvert\lvert\mathbb{G}_1\rvert$
MK	Our	$(\lvert\mathcal{U}_D\rvert + \lvert\mathcal{U}_U\rvert + 2)\lvert\mathbb{Z}_p\rvert$	Yes	No	No
	YWRL	$(\lvert\mathcal{U}_D\rvert + 2)\lvert\mathbb{Z}_p\rvert$	Yes	No	No
DK	Our	$(\lvert\lambda\rvert + 1 + \lvert\pi\rvert)\lvert\mathbb{G}_1\rvert$	No	$\lvert\lambda\rvert\lvert\mathbb{G}_1\rvert$	Yes
	YWRL	$(\lvert\lambda\rvert + 1)\lvert\mathbb{G}_1\rvert$	No	$\lvert\lambda\rvert\lvert\mathbb{G}_1\rvert$	Yes
CT_D	Our	$(\lvert\gamma\rvert + 1)\lvert\mathbb{G}_1\rvert + \lvert\mathbb{G}_T\rvert$	No	Yes	No
	YWRL	$(\lvert\gamma\rvert + 1)\lvert\mathbb{G}_1\rvert + \lvert\mathbb{G}_T\rvert$	No	Yes	No
CT_U	Our	$(\lvert\text{NRAttrs}\rvert + 1)\lvert\mathbb{G}_1\rvert + \lvert\mathbb{G}_T\rvert$	No	Yes	No
	YWRL	–	–	–	–
$crk_{i\leftrightarrow i'}$	Our	$\lvert\mathbb{Z}_p\rvert$	Yes	Yes	No
	YWRL	$\lvert\mathbb{Z}_p\rvert$	Yes	Yes	No
$urk_{i\leftrightarrow i'}$	Our	$\lvert\mathbb{Z}_p\rvert$	Yes	No	Yes
	YWRL	–	–	–	–

Table 3: Storage costs comparison.

Message	Scheme	Message size
m_1	Our	$2\lvert\mu\rvert\lvert\mathbb{Z}_p\rvert + (\lvert\mu\rvert + \lvert\text{NRAttrs}\rvert)\lvert\mathbb{G}_1\rvert + \lvert\mathbb{G}_T\rvert$
	YWRL	$\lvert\mu\rvert\lvert\mathbb{Z}_p\rvert + \lvert\mu\rvert\lvert\mathbb{G}_1\rvert$
$m_3 + m_5$	Our	$(2\lvert\lambda\rvert + \lvert\gamma\rvert + 1 + \lvert\rho\rvert)\lvert\mathbb{G}_1\rvert + (\lvert\rho\rvert + 1)\lvert\mathbb{G}_T\rvert + \lvert\rho\rvert\lvert\mu\rvert\lvert\mathbb{Z}_p\rvert$
	YWRL	$(2\lvert\lambda\rvert + \lvert\gamma\rvert + 1)\lvert\mathbb{G}_1\rvert + \lvert\mathbb{G}_T\rvert$
m_4	Our	$(\lvert\lambda\rvert + \lvert\gamma\rvert)\lvert\mathbb{G}_1\rvert$
	YWRL	–
m_5	Our	$(\lvert\gamma\rvert + 1)\lvert\mathbb{G}_1\rvert + \lvert\mathbb{G}_T\rvert$
	YWRL	$(\lvert\gamma\rvert + 1)\lvert\mathbb{G}_1\rvert + \lvert\mathbb{G}_T\rvert$

Table 4: Communication cost comparison of key revocation and data consumption procedures. Messages refer to Fig. 3.

6.2 Performance Evaluation by Simulation

In this section, we evaluate the performance of the YWRL scheme and ours by simulating them. The aim is to practically understand the price of the additional security provided by our scheme. We developed a Matlab simulator that simulates a data owner, a cloud server, and a number of users that perform the math operations of the two schemes for a period of time. User join, data production, key revocation, and data consumption events are generated randomly. The simulator does not actually perform the math operations, but rather it keeps track of the type and the number of operations that a real system should perform. Such operations include operations in $\mathbb{G}_1$, operations in $\mathbb{G}_T$, and bilinear pairings, as well as the other cryptographic operations, namely, signature, signature verification, asymmetric encryption, and asymmetric decryption. The simulator also simulates the messages exchanged between the entities, and it keeps track of the total traffic in bytes, both in upload and download for each entity in the system. For the data ciphertexts, only the encryption overhead is considered, that is the size of the ciphertext size minus the size of the actual (in-the-clear) data. By doing this, the simulator actually measures the com-

putational and traffic costs with respect to a "baseline" insecure system that always stores and transmits data in the clear.

The simulator initializes the YWRL scheme with a universe of $|\mathcal{U}_D| + 1$ attributes, and our scheme with a universe of $|\mathcal{U}_D| + |\mathcal{U}_U| + 1$ attributes. The version of each attribute subject to versioning is initially set to 0. Then, the simulator generates `nInitUsers` users as follows. A data sub-policy with `nKeyAttrs` randomly chosen attributes from the data sub-universe is assigned to each user. The version of each decryption key component is initially set to 0. The data sub-policy is defined as a disjunctive normal form (DNF), i.e., an OR of AND gates. The OR gate is the root of the data sub-policy and has $\lfloor \texttt{nKeyAttrs} / \lfloor \sqrt{\texttt{nKeyAttrs}} \rfloor \rfloor$ child nodes. Each child node is an AND gate with $\texttt{nKeyAttrs} - (\lfloor \texttt{nKeyAttrs} / \lfloor \sqrt{\texttt{nKeyAttrs}} \rfloor \rfloor - 1) \cdot \lfloor \sqrt{\texttt{nKeyAttrs}} \rfloor$ child nodes. In this way, we obtain a data sub-policy with approximately the same number of AND gates and children for each AND gate. For example, with $\texttt{nKeyAttrs} = 10$, we obtain a data sub-policy with 3 AND gates, with 3, 3, and 4 children, respectively. Fig. 4 shows an example of access policy of our scheme.

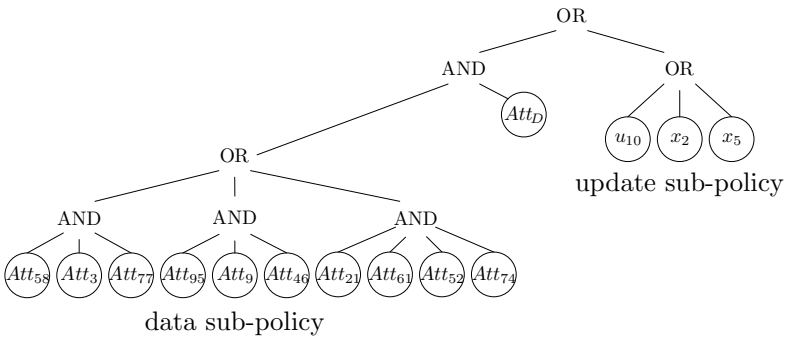


Figure 4: Example of access policy in our simulated scheme. The data sub-policy is in DNF and is composed of $\texttt{nKeyAttrs} = 10$ attributes.

Finally, the simulator generates an initial database of `nInitCtxs` data ciphertexts. Each data ciphertext is labeled with `nCtxAttrs` randomly chosen attributes from the data sub-universe, plus the dummy attribute. The version of each ciphertext component is initially set to 0.

After these preliminary operations, the simulator starts generating the events and starts recording the math operations and the traffic overhead. The simulator models the user join, data production, key revocation, and data consumption events as Poisson processes with different mean times, namely, `UJMeantime`, `DPMeantime`, `KRMeantime`, and `DCMeantime`. Within the user join event, a new user is created, and an access policy is randomly generated according to the method described earlier. The version of each decryption key component of the data sub-policy is set to the latest version. Within the data production event, a new data ciphertext is created and randomly labeled according to the method described earlier. The version of each ciphertext component, except for the dummy attribute, is set to the latest version. Within the key revocation

event, the simulator chooses a user to revoke at random. The simulator runs for a simulation time `SimTime`, and, when the simulation ends, it outputs the number of operations and the traffic overhead experienced by each entity in both schemes.

In the first set of simulations, we evaluate the computational load for users and data owner in terms of time. For each type of cryptographic operation, we multiply the number of operations for the time needed to perform that operation. We obtained the time needed to perform each cryptographic operation by running a benchmark on a device. In this set of simulations, we set the simulation time `SimTime` equal to one year; a user join event occurring on average every six hours, i.e., `UJMeantime` = 6 h; a key revocation event occurring on average every six hours, i.e., `KRMeantime` = 6 h; a data production event occurring on average every hour, i.e., `DPMeantime` = 1 h; a data consumption event occurring on average every hour per user, i.e., `DCMeantime` = 1 h; an initial number of users `nInitUsers` = 1000; a data sub-universe of $|\mathcal{U}_D| = 100$ attributes; an update sub-universe of $|\mathcal{U}_U| = 2\hat{}(\log_2((\texttt{nInitUsers} + \texttt{SimTime}/\texttt{UJMeantime}) \cdot 2))$ to have enough identities for all the users; an initial database of `nInitCtxs` = 10 000 data ciphertexts; a number of attributes in each data sub-policy `nKeyAttrs` = 10; and a number of attributes labeling each data ciphertext `nCtxAttrs` = 30, plus the dummy attribute. For the KP-ABE scheme, we considered a 512-bit elliptic curve with embedding degree 2 that admits an efficiently computable symmetric pairing. For digital signature algorithm and asymmetric encryption, we considered ECDSA with the secp160r1[1] curve, and RSA with 1024-bit key, respectively. With these settings, the scheme provides for 80-bit security strength.

In order to evaluate the computational load for users and data owner in terms of time, we considered an IoT scenario in which these entities are equipped with a single-board computer, namely a Raspberry Pi 3 Model B+. This scenario is typical in IoT. For example, the IoT smart city application described in [6, 19] includes some IP cameras that record the traffic on the streets and send videos to a central gateway (our data owner). The gateway stores on the cloud server multiple short video files in an encrypted form, in such a way to implement a sort of encrypted streaming. Many smart vehicles (our users) want to access videos to detect possible traffic congestion in advance and choose more convenient routes. Such smart vehicles can be authorized to access videos produced only by a given set of cameras at some given days/hours, depending on the service subscription type. Note that the ECUs of vehicles are typically single-board computers, whose computation capabilities are comparable to the Raspberry Pi that we considered. We ran benchmarks on the Raspberry Pi to determine the time needed to perform each operation, using the PBC library[2] for pairing-based operations and OpenSSL 1.0.1k library[3] for ECDSA and RSA. Table 5 shows the results of these benchmarks.

In Figs. 5a and 5b we report simulation results for our scheme, and we show the

[1] https://www.secg.org/SEC2-Ver-1.0.pdf

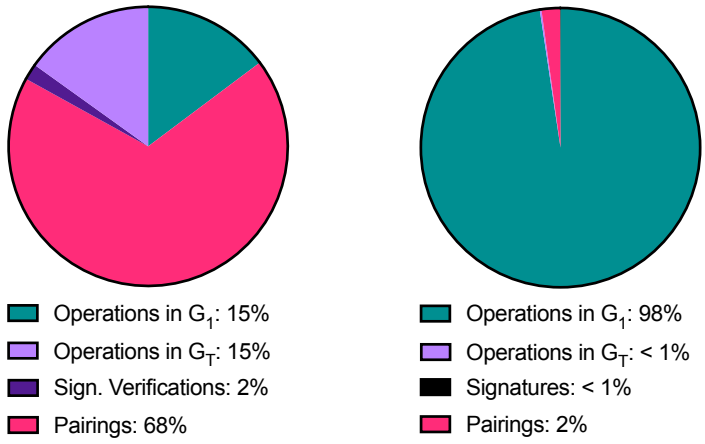
[2] https://crypto.stanford.edu/pbc

[3] https://www.openssl.org/source/old/1.0.1

Operation Type	Time (ms) Mean	95 % CI
Bilinear Pairing	15.663	±0.003 71
Operation in $\mathbb{G}_1$	9.495	±0.0326
Operation in $\mathbb{G}_T$	2.037	±0.0103
Signature	0.445	±0.000 163
Verification	1.656	±0.001 27
Asymmetric Encryption	0.208	±0.000 057 4

Table 5: Raspberry Pi 3 Model B+ benchmark. Table shows mean values averaged on 1000 independent repetitions and 95 % confidence intervals.

portions of time spent by the average user and data owner performing cryptographic operations, averaged on 30 independent repetitions.



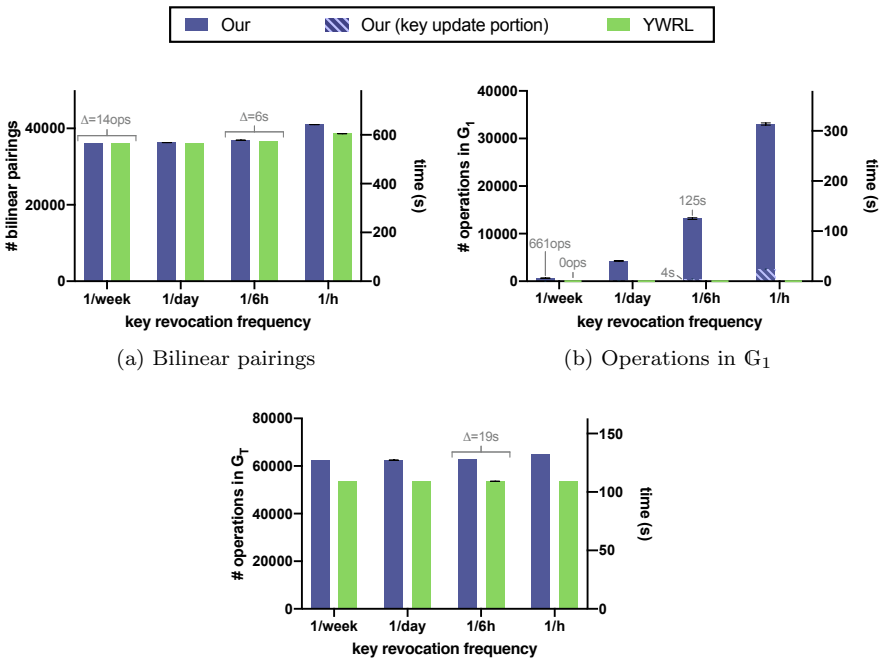
(a) User

(b) Data Owner

Figure 5: Time percentage spent on each cryptographic operation in our scheme.

The user spends 98 % of the time performing pairing-based operations. Operations in $\mathbb{G}_1$ are performed during key update and re-encryption, while bilinear pairing and operations in $\mathbb{G}_T$ are performed during decryption of both data and update ciphertexts. Despite the number of operations in $\mathbb{G}_T$ is greater than the number of bilinear pairings, the latter metrics is indisputably the most burdensome for the user with 68 % out of the total time. On the data owner, the load due to pairing-based operations is even more evident. Indeed, more than 99 % of the time is employed for KP-ABE encryption. Digital signature algorithm and asymmetric cryptography operations take only 2 % of the time for the user and less than 1 % of the time for the data owner, and this confirms that the pairing-based operations have the highest impact on the performance of the scheme.

In the second set of simulations, we compare the performance of our scheme with the YWRL one by varying the frequency of a key revocation event, i.e., by varying 1/`KRMeantime`. As already pointed out, for providing more security our scheme introduces some costs which mainly weigh on the user, who is actively involved in key update and re-encryption tasks. The more we raise the key revocation frequency, the more the computational load on the user will increase as he/she will be required to update his/her decryption key and/or ciphertexts more often. We set the user join frequency to be equal to the key revocation one, so, on average, the number of users remains constant throughout the simulation. We ran this set of simulations for four different frequencies, and we report results regarding the average user's load in Fig. 6.



(a) Bilinear pairings

(b) Operations in $\mathbb{G}_1$

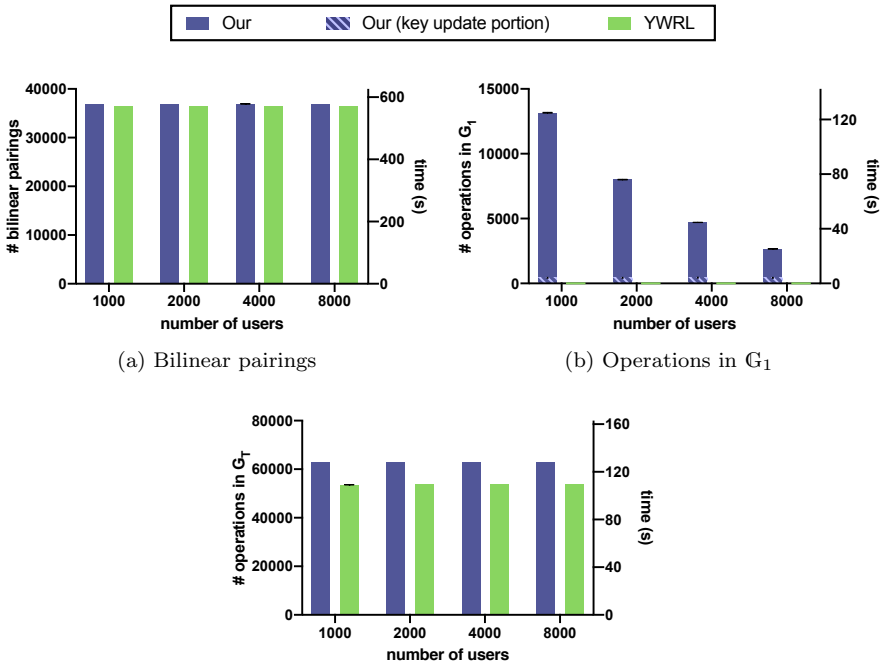
(c) Operations in $\mathbb{G}_T$

Figure 6: User load due to pairing-based operations in a one-year-long simulation. Graphs show how the user load varies with regards to the key revocation frequency. The initial number of users is 1000. Graphs show 95 % confidence intervals.

For key revocations occurring on average every week, the difference of load between the schemes is negligible, namely about 660 operations in $\mathbb{G}_1$, 50 pairings, 8775 operations in $\mathbb{G}_T$ in one year. Despite the high number of operations in $\mathbb{G}_T$, this type of operation is very fast, and the difference between the schemes is almost constant for all the tested frequencies, varying in the range of 18 s and 23 s. For key revocations occurring on average every day and every six hours,

the difference of load between the schemes is still limited. Indeed, the overhead is about 6 s in a year for the additional pairings and about 125 s for the operations in $\mathbb{G}_1$. Notably, the operations in $\mathbb{G}_1$ can be divided into operations performed for key update and operations performed for re-encryption. In the case re-encryption is not a peculiar feature for an application, the time spent performing operations in $\mathbb{G}_1$ drops from 125 s to about 4 s. For key revocations occurring on average every hour, the overhead introduced by our scheme grows by a significant amount, mainly due to re-encryption operations. However, systems in which a key revocation is so frequent should be uncommon.

In the third set of simulations, we aim at showing the behavior of the schemes for different average number of users. To this purpose, we varied the initial number of users `nInitUsers` from 1000 to 8000. Moreover, we set `KRMeantime` = 6 h, and `UJMeantime` to the same value. This allows to maintain an average number of users close to the initial number of users throughout the simulation. As before, each simulation is averaged on 30 independent repetitions. In Fig. 7, we show that our scheme scales with the number of users. Indeed, in these



(a) Bilinear pairings

(b) Operations in $\mathbb{G}_1$

(c) Operations in $\mathbb{G}_T$

Figure 7: User load due to pairing-based operations in a one-year-long simulation. Graphs show how the user load varies with regards to the initial number of users. Key revocation frequency is 1/6hours. Graphs show 95 % confidence intervals.

graphs we point out that the average user's load decreases with the number

of users in the system. We report results for a key revocation frequency of 1/6hours, but this trend holds independently of the key revocation frequency value. In Figs. 7a and 7c, we observe that the number of pairings and operations in $\mathbb{G}_T$ remain almost constant, independently of the number of users, while the number of operations in $\mathbb{G}_1$ (Fig. 7b) decreases significantly. This is because the re-encryption of ciphertexts is distributed among more users which participate in updating the ciphertexts, and, hence, the number of operations per user decreases. Compared to the YWRL scheme, where the load experienced by each user is not influenced by the the number of users in the system, in our scheme an additional cost weighs on the user. However, the difference of computational cost for pairings and operations in $\mathbb{G}_T$ is limited. Moreover, the difference of number of operations in $\mathbb{G}_1$ per user decreases with the number of users. Its lower bound tends to the number of operations executed only for the key update task. As shown with the previous set of simulations this cost is negligible.

In Table 6, we show a comparison of the traffic overhead experienced by each entity in both schemes. The simulator was configured as in the first set

Entity	**Scheme**	**Download** (MB)		**Upload** (MB)	
		Mean	95 % CI	Mean	95 % CI
Data owner	Our	0.0000	0.0000	81.6680	±0.5746
	YWRL	0.0000	0.0000	22.5177	±0.0757
Cloud server	Our	920.5880	±4.7833	5067.4901	±61.8700
	YWRL	22.5177	±0.0757	4895.8708	±60.9210
User	Our	5.0832	±0.0026	0.8425	±0.0110
	YWRL	4.9110	±0.0095	0.0000	0.0000

Table 6: Traffic overhead comparison. Table shows mean values averaged on 30 independent repetitions and 95 % confidence intervals.

of simulations. For convenience, we recall that the simulation time is one year, the initial number of users is 1000 and the key revocation frequency is 1/6 h. In both schemes, the data owner experiences only upload overhead as result of the execution of data production and key revocation procedures. In our scheme, the upload overhead is about four times the one in the YWRL scheme. This is because in the key revocation procedure, besides the cloud re-encryption keys, our scheme includes the overhead due to the transmission of user re-encryption keys and update ciphertexts. However, we stress that computational and communication costs for the data production procedure are equivalent in both the schemes. This means that when data owner and key authority are different entities, the additional cost of our scheme weighs only on the key authority. In Table 6, we observe that the cloud server is inevitably the most loaded entity as it communicates with all the users and with the data owner. Though in our scheme the cloud server generates a high download overhead, we note that the upload overhead is about the same in both the schemes. The additional download overhead is mainly due to the communication with the users, which update their keys and ciphertexts and upload them on the cloud server. However, by looking at the upload overhead experienced by the single

user, we notice that is negligible, namely 0.84 MB per year. Also the download overhead experienced by the user is limited, and the difference between the two schemes is negligible, namely 0.17 MB per year. As we showed that our scheme scales with the number of users, these differences are further reduced when more users are part of the system.

7 Conclusions

In this paper, we proposed a revocable KB-ABE scheme that ensures data confidentiality and fine-grained access control even if the cloud server comes into possession of revoked keys. We then formally proved that the proposed scheme is secure in the Selective-Set model under the Decisional Bilinear Diffie-Hellman assumption. Finally, we analyzed the computational costs and performed simulations to show that in our scheme the user experiences a slightly higher computational cost with respect to the Yu et al's scheme [17], which does not provide for revocation undoing resistance. Considering an IoT scenario, for example a smart city application like in [6, 19], and assuming 8000 users equipped with Raspberry Pi boards and revocations occurring on average every six hours, a one-year-long simulation revealed that the average user experiences a load (in terms of computation time) 7.37 % higher than Yu et al.'s scheme. Also, the simulations showed that our scheme is scalable with the number of users, i.e., the more users are in the system, and the less load introduced to each user.

Acknowledgements

This work was supported by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence) and by the European Union's Horizon 2020 research and innovation programme "European Processor Initiative" under grant agreement No. 826647.

References

[1] N. Oualha and K. T. Nguyen, "Lightweight attribute-based encryption for the internet of things," in *2016 25th Int. Conf. on Computer Communication and Networks*, 2016, pp. 1–6.

[2] Q. Huang, L. Wang, and Y. Yang, "DECENT: Secure and fine-grained data access control with policy updating for constrained IoT devices," *World Wide Web*, vol. 21, no. 1, pp. 151–167, 2018.

[3] M. La Manna, P. Perazzo, M. Rasori, and G. Dini, "fABElous: An attribute-based scheme for industrial internet of things," in *2019 IEEE Int. Conf. on Smart Computing*, 2019, pp. 33–38.

[4] B. Girgenti, P. Perazzo, C. Vallati, F. Righetti, G. Dini, and G. Anastasi, "On the feasibility of attribute-based encryption on constrained IoT devices for smart systems," in *2019 IEEE Int. Conf. on Smart Computing*. IEEE, 2019, pp. 225–232.

[5] M. Ambrosin, A. Anzanpour, M. Conti, T. Dargahi, S. R. Moosavi, A. M. Rahmani, and P. Liljeberg, "On the feasibility of attribute-based encryption on internet of things devices," *IEEE Micro*, vol. 36, no. 6, pp. 25–35, 2016.

[6] M. Rasori, P. Perazzo, and G. Dini, "A lightweight and scalable attribute-based encryption system for smart cities," *Computer Communications*, vol. 149, pp. 78–89, 2020.

[7] J. A. Akinyele, M. W. Pagano, M. D. Green, C. U. Lehmann, Z. N. Peterson, and A. D. Rubin, "Securing electronic medical records using attribute-based encryption on mobile devices," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 75–86.

[8] L. Ibraimi, M. Asim, and M. Petković, "Secure management of personal health records by applying attribute-based encryption," in *Proceedings of the 6th International Workshop on Wearable, Micro, and Nano Technologies for Personalized Health*, 2009, pp. 71–74.

[9] J. Eom, D. H. Lee, and K. Lee, "Patient-controlled attribute-based encryption for secure electronic health records system," *Journal of medical systems*, vol. 40, no. 12, p. 253, 2016.

[10] S. Wang, Y. Zhang, and Y. Zhang, "A blockchain-based framework for data sharing with fine-grained access control in decentralized storage systems," *IEEE Access*, vol. 6, pp. 38 437–38 450, 2018.

[11] A. Arena, P. Perazzo, and G. Dini, "Virtual private ledgers: embedding private distributed ledgers over a public blockchain by cryptography," in *Proceedings of the 23rd International Database Applications & Engineering Symposium*, 2019, pp. 1–9.

[12] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM conference on Computer and communications security*. ACM, 2006, pp. 89–98.

[13] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007, pp. 321–334.

[14] A. Boldyreva, V. Goyal, and V. Kumar, "Identity-based encryption with efficient revocation," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 417–426.

[15] H. Cui, R. H. Deng, Y. Li, and B. Qin, "Server-aided revocable attribute-based encryption," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 570–587.

[16] B. Qin, Q. Zhao, D. Zheng, and H. Cui, "Server-aided revocable attribute-based encryption resilient to decryption key exposure," in *Int. Conf. on Cryptology and Network Security*. Springer, 2017, pp. 504–514.

[17] S. Yu, C. Wang, K. Ren, and W. Lou, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *Infocom, 2010 proceedings IEEE*. IEEE, 2010, pp. 1–9.

[18] K. Yang, X. Jia, and K. Ren, "Attribute-based fine-grained access control with efficient revocation in cloud storage systems," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013, pp. 523–528.

[19] M. Rasori, P. Perazzo, and G. Dini, "ABE-cities: An attribute-based encryption system for smart cities," in *2018 IEEE Int. Conf. on Smart Computing*. IEEE, 2018, pp. 65–72.

[20] J. Li, C. Jia, J. Li, and X. Chen, "Outsourcing encryption of attribute-based encryption with mapreduce," in *Int. Conf. on Information and Communications Security*. Springer, 2012, pp. 191–201.

[21] L. Touati, Y. Challal, and A. Bouabdallah, "C-CP-ABE: Cooperative ciphertext policy attribute-based encryption for the internet of things," in *2014 Int. Conf. on Advanced Networking Distributed Systems and Applications (INDS)*, 2014, pp. 64–69.

[22] L. Touati and Y. Challal, "Collaborative KP-ABE for cloud-based internet of things applications," in *2016 IEEE Int. Conf. on Communications (ICC)*, 2016, pp. 1–7.

[23] C. Zuo, J. Shao, G. Wei, M. Xie, and M. Ji, "CCA-secure ABE with outsourced decryption for fog computing," *Future Generation Computer Systems*, vol. 78, pp. 730–738, 2018.

[24] X. Yao, Z. Chen, and Y. Tian, "A lightweight attribute-based encryption scheme for the internet of things," *Future Generation Computer Systems*, vol. 49, pp. 104–112, 2015.

[25] S. Ding, C. Li, and H. Li, "A novel efficient pairing-free CP-ABE based on elliptic curve cryptography for IoT," *IEEE Access*, vol. 6, pp. 27 336–27 345, 2018.

[26] B. Waters, "Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization," in *International Workshop on Public Key Cryptography*, 2011, pp. 53–70.

[27] A. Sahai, H. Seyalioglu, and B. Waters, "Dynamic credentials and ciphertext delegation for attribute-based encryption," in *Annual Cryptology Conference*. Springer, 2012, pp. 199–217.

[28] S. Xu, G. Yang, and Y. Mu, "Revocable attribute-based encryption with decryption key exposure resistance and ciphertext delegation," *Information Sciences*, vol. 479, pp. 116–134, 2019.

[29] H. Ma, R. Zhang, S. Sun, Z. Song, and G. Tan, "Server-aided fine-grained access control mechanism with robust revocation in cloud computing," *IEEE Trans. on Services Computing*, 2019.

[30] J. Hur and D. K. Noh, "Attribute-based access control with efficient revocation in data outsourcing systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 22, no. 7, pp. 1214–1221, 2011.

[31] E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes," in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 537–554.