

SEA-BREW: A Scalable Attribute-Based Encryption Revocable Scheme for Low-Bitrate IoT Wireless Networks

Michele La Manna^{†*}, · Pericle Perazzo^{*}, · Gianluca Dini^{*}

Received: date / Accepted: date

Abstract Attribute-Based Encryption (ABE) is an emerging cryptographic technique that allows one to embed a fine-grained access control mechanism into encrypted data. In this paper we propose a novel ABE scheme called SEA-BREW (Scalable and Efficient ABE with Broadcast REvocation for Wireless networks), which is suited for Internet of Things (IoT) and Industrial IoT (IIoT) applications. In contrast to state-of-the-art ABE schemes, ours is capable of securely performing key revocations with a single short broadcast message, instead of a number of unicast messages that is linear with the number of nodes. This is desirable for low-bitrate Wireless Sensor and Actuator Networks (WSANs) which often are the heart of (I)IoT systems. In SEA-BREW, sensors, actuators, and users can exchange encrypted data via a cloud server, or directly via wireless if they belong to the same WSAN. We formally prove that our scheme is IND-CPA secure also in case of an untrusted cloud server that colludes with a set of users, under the generic bilinear group model. We show by simulations that our scheme requires a constant computational overhead on the cloud server with respect to the complexity of the access control policies. This is in contrast to state-of-the-art solutions, which require instead a linear computational overhead.

Keywords Attribute-Based Encryption · ABE · Ciphertext-Policy Attribute-Based Encryption · CP-ABE · Key Revocation · Broadcast.

[†] Department of information Engineering (DINFO)
University of Florence, Italy.
E-mail: m.lamanna@unifi.it

^{*}Department of information Engineering (DII)
University of Pisa, Italy.
E-mail: michele.lamanna@phd.unipi.it;
E-mail: pericle.perazzo@unipi.it;
E-mail: gianluca.dini@unipi.it

1 Introduction

In the Internet of Things (IoT) vision [3, 12, 28, 15], ordinary “things” like home appliances, vehicles, industrial robots, etc. will communicate and coordinate themselves through the Internet. By connecting to Internet, things can provide and receive data from users or other remote things, both directly or via cloud. Cloud-based services are in turn provided by third-party companies, such as Amazon AWS or Microsoft Azure, usually through pay-per-use subscription. On the other hand, outsourcing sensitive or valuable information to external servers exposes the data owner to the risk of data leakage. Think for example of an industrial IoT network that communicates and processes business-critical information. A data leakage could expose a company or an organization to industrial espionage, or it can endanger the privacy of employees or customers. Encrypting data on cloud servers is a viable solution to this problem. An emerging approach is *Attribute-Based Encryption* (ABE) [27, 14, 4, 35, 25, 34], which is a cryptographic technique that embeds an access control mechanism within the encrypted data. ABE describes data and decrypting parties by means of *attributes*, and it regulates the “decryptability” of data with *access policies*, which are Boolean formulas defined over these attributes. In ABE, encrypting parties use an *encryption key*, which is public and unique, whereas any decrypting party uses a *decryption key*, which is private and different for each of them.

Unfortunately, state-of-the-art ABE schemes are poorly suitable for the majority of IoT applications. The biggest problem is not computational power as one may think, since ABE technology and elliptic curve operations have proven to be well-supportable by mobile devices [2, 1] and modern IoT devices [13, 30]. The most

problematic aspect is the recovery procedure in case of key compromise, which requires to send an update message to all the devices [35]. Sending many update messages could be quite burdensome for wireless networks with a limited bitrate, like those employed in IoT [10, 22]. Indeed modern IoT networks use low-power communication protocols like Bluetooth LE, IEEE 802.15.4, and LoRA, which provide for low bitrates (230Kbps for BLE [31], 163Kbps for 802.15.4 [19], 50Kbps for LoRA [11]).

In this paper, we propose *SEA-BREW* (Scalable and Efficient ABE with Broadcast REvocation for Wireless networks), an ABE revocable scheme suitable for low-bitrate Wireless Sensor and Actuator Networks (WSANs) in IoT applications. *SEA-BREW* is highly scalable in the number and size of messages necessary to manage decryption keys. In a WSAN composed of n decrypting nodes, a traditional approach based on unicast would require $\mathcal{O}(n)$ messages. *SEA-BREW* instead, is able to revoke or renew multiple decryption keys by sending a single broadcast message over a WSAN. For instance, if $n = 50$ and considering a symmetric pairing with 80-bit security, the traditional approach requires 50 unicast messages of 2688 bytes each, resulting in about 131KB of total traffic. *SEA-BREW* instead, requires a single 252-byte broadcast message over a WSAN. Also, our scheme allows for per-data access policies, following the *Ciphertext-Policy Attribute-Based Encryption* (CP-ABE) paradigm, which is generally considered flexible and easy to use [4, 20, 2]. In *SEA-BREW*, things and users can exchange encrypted data via the cloud, as well as directly if they belong to the same WSAN. This makes the scheme suitable for both remote cloud-based communications and local delay-bounded ones. The scheme also provides a mechanism of *proxy re-encryption* [35, 36, 37] by which old data can be re-encrypted by the cloud to make a revoked key unusable. We formally prove that our scheme is adaptively IND-CPA secure also in case of an untrusted cloud server that colludes with a set of users, under the generic bilinear group model. We finally show by simulations that the computational overhead is constant on the cloud server, with respect to the complexity of the access control policies.

The rest of the paper is structured as follows. In Section 2 we review the current state of the art. In Section 3 we explain our system model and we mathematically describe the *SEA-BREW* primitives. In Section 4 we introduce our threat model, and we formally prove the security of *SEA-BREW*. In Section 5 we evaluate our scheme both analytically and through simulations. Finally, in Section 6 we conclude the paper.

2 Related Work

Attribute-Based Encryption has been applied to protect confidentiality and assure fine-grained access control in many different application scenarios like cloud computing [21, 35, 33, 16], e-health [23], wireless sensor networks [34], Internet of Things [32, 29], smart cities [25], smart industries [18], online social networks [17], and so on.

Bethencourt et al. [4] proposed the first CP-ABE scheme. Their scheme lacks functionalities for key revocation and ciphertext re-encryption, which we provide in our scheme. Notably, a naive key revocation mechanism can be realized on the Bethencourt et al.'s scheme, but it requires to send a new decryption key for each user in the system, resulting in $\mathcal{O}(n)$ point-to-point messages where n is the number of users. In contrast, *SEA-BREW* is able to revoke or renew a decryption key by sending a single $\mathcal{O}(1)$ -sized broadcast message over a wireless network, and it also provides a re-encryption mechanism delegated to the untrusted cloud server.

Yu et al. [34] realized the first distributed fine-grained data access control for Wireless Sensor Networks (WSNs) using ABE. Their proposed architecture is composed of one trusted network controller, several users and several nodes. Every user has a secret key generated by the network controller according to a policy that describes the type of data it can access. Every node is preloaded with a set of attributes and their relative public quantities, generated by the network controller. In addition, their scheme uses *Key-Policy Attribute-Based Encryption* (KP-ABE) [14]. We adopt instead CP-ABE, which is generally considered more flexible and easier to use than KP-ABE [2, 4, 20]. Moreover, their system does not permit the WSN devices to receive and decrypt ABE-encrypted data, but only to encrypt and send it. This precludes the possibility to use their scheme in complex Industrial IoT scenarios in which some devices also receive encrypted data, for example in case of actuators. Finally, their scheme does not provide for a re-encryption mechanism, while ours does.

Picazo-Sanchez et al. [23] proposed a secure publish-subscribe protocol for medical Wireless Body Area Networks (WBANs) using ABE. The proposed architecture is a star-topology network where a smartphone (or a similar device) manages the communication among various nodes placed over/inside the user's body, monitoring his/her health conditions. Each node can publish its data and subscribe to data generated from other nodes. Singh et al. [29] proposed a secure MQTT for IoT, with the possibility of using ABE. The proposed

architecture is composed of one public key generator, one broker and several devices, which can act both as subscriber and publisher. Every device subscribes to the data it needs for its proper functioning. Both Picazo-Sanchez et al.'s and Singh et al.'s schemes follow the CP-ABE paradigm like ours, but they do not provide a key revocation mechanism, nor a re-encryption one.

Touati et al. [32] proposed an ABE system for IoT which implements a key revocation procedure based on time slots. In their work, time is divided in slots, and policies can be modified only at the beginning of a slot. This approach is efficient only if key revocations and policy changes are known a priori. An example is an access privilege that expires after one year. Unfortunately, in many systems there is not the possibility to know beforehand when and which access privilege should be revoked. For example, in case a decryption key gets compromised the system must revoke it as soon as possible. Our scheme gives this possibility. Cui et al. [8], and Qin et al. [24] proposed two revocable CP-ABE schemes which do not require to communicate with data producers during a revocation process. However, their schemes require all data producers to be time-synchronised in a secure manner. This could be quite difficult to achieve and hard to implement in a WSN where data producers are often very resource constrained sensors. Also, their schemes require the cloud server to perform a ciphertext transformation for *each single* data request, even if no revocation occurred. Such a transformation is needed to make the ciphertext decryptable by the data consumer. Instead, in SEA-BREW no ciphertext update is needed if no revocation occurred. Furthermore, in case of key revocation, it requires the cloud server to perform a ciphertext update only once.

Rasori et al. [25] proposed a system for smart cities using ABE. The application offers a service for real-time road monitoring in a smart city scenario. Sensors (e.g., cameras) are placed along the roads and store their sensed data on a cloud server. Users can pay a subscription and obtain an ABE decryption key with which they can retrieve the information (e.g., videostream of the traffic) in real time. Their scheme provides also a proxy re-encryption mechanism. Notably, its revocation mechanism is not efficient for WSN, as it requires $\mathcal{O}(n)$ different messages where n is the number of decrypting parties. On the other hand, SEA-BREW is able to revoke or renew a decryption key by sending a single $\mathcal{O}(1)$ -sized broadcast message over the wireless network.

3 System Model

Figure 1 shows our reference system model. We assume

a low-bitrate WSN, composed of a set of sensors and actuators, which upload and download encrypted data to/from a *cloud server*. Sensors and actuators access the cloud server through an Internet-connected *WSN gateway* node, belonging to the WSN. Sensors and actuators inside the WSN can also communicate directly, without passing through the cloud server. We assume that some sensors and some actuators are outside the WSN, and they can also upload and download encrypted data to/from the cloud server, but they cannot communicate directly. In addition, human users outside the WSN can upload and download encrypted data to/from the cloud server. The encrypted data received by an actuator could be a command that the actuator must execute, as well as a measurement from a sensor that the actuator can use to take some decision. The cloud server is an always-on-line platform managed by an untrusted third-party company which offers storage and computational power to privates or other companies. Finally, a fully trusted *key authority* is in charge of generating, updating and distributing cryptographic keys.

In the following, we will call *producers* all those system entities that produce and encrypt data. This includes sensors internal or external to the WSN, which sense data, as well as users that produce data or commands for actuators. Similarly, we will call *consumers* all those system entities that decrypt and consume data. This includes actuators internal or external to the WSN, which request data and which receive commands, as well as users that request data. For the sake of simplicity, we keep the “producer” and the “consumer” roles separated, however SEA-BREW allows a single device or a single user to act as both. Producers that are inside the WSN will be called *WSN producers*, while those outside the WSN will be called *remote producers*. Similarly, consumers that are inside the WSN will be called *WSN consumers*, while those outside the WSN will be called *remote consumers*.

As an use-case example, consider a smart factory with many sensors and actuators which must communicate in a delay-bounded way to implement a real-time application [6]. Given the strict requirements, sensors and actuators must communicate directly through the WSN, without losing time in remote communications with the cloud. The WSN inside the smart factory use IEEE 802.15.4 as a link-layer protocol, which is low-energy and low-bitrate. As a consequence, communications and key management operations must be as lightweight as possible. In addition, employees, external sensors and external actuators involved for remote applications will upload or download data to/from the cloud server.

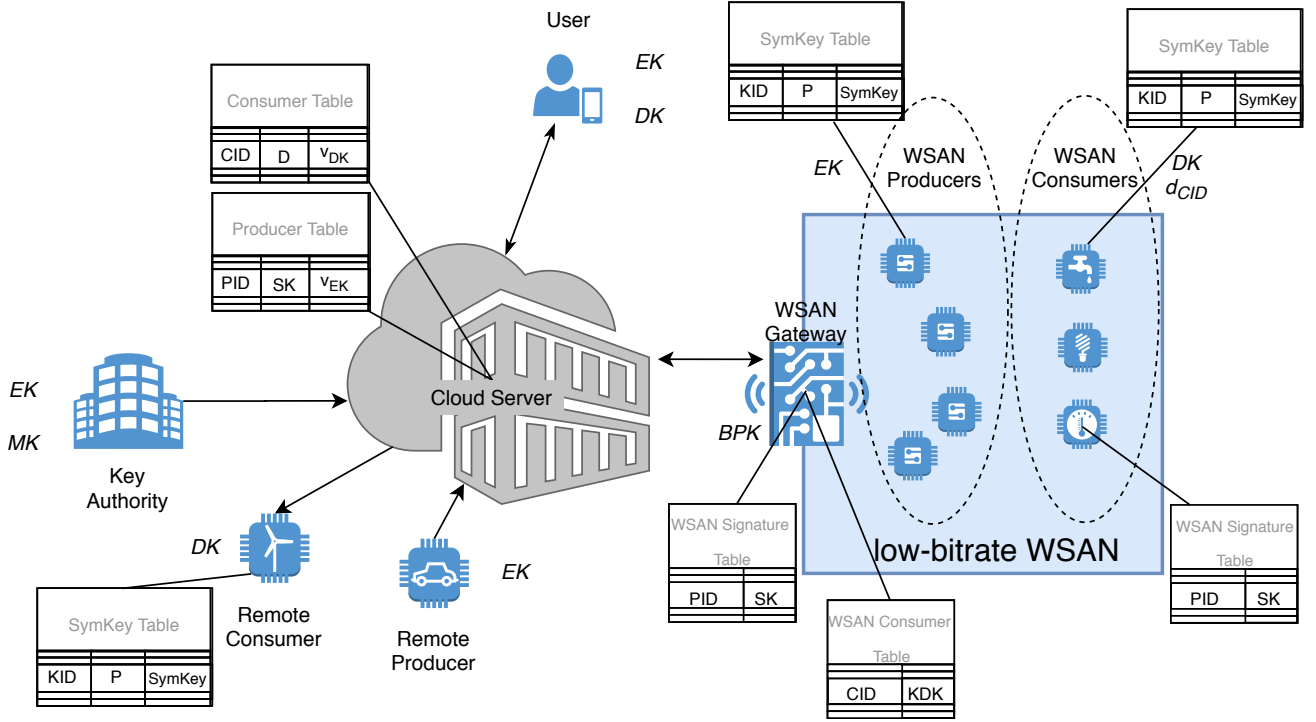


Fig. 1 SEA-BREW system model.

Each producer encrypts data by means of an *encryption key (EK)*. Each consumer decrypts data by means of a *decryption key (DK)*. The encryption key is public and unique for all the producers, whereas the decryption key is private and specific of a single consumer. A single piece of encrypted data is called *ciphertext (CP)*. Each consumer is described by a set of attributes (γ), which are cryptographically embedded into its decryption key. The access rights on each ciphertext are described by an *access policy (P)*. We assume that the key authority, the cloud server, and the WSN gateway have their own pair of asymmetric keys used for digital signature and encryption (e.g., RSA or ECIES keys). In addition, each producer and each consumer has a unique identifier called, respectively, *producer identifier (PID)* and *consumer identifier (CID)*, which are assigned by the key authority. If a device acts as both producer and consumer, then it will have both a producer identifier and a consumer identifier.

When a decryption key needs to be revoked (e.g., because it is compromised or because a consumer has to leave the system), the key authority must ensure that such a decryption key will not be able to decrypt data anymore. This is achieved by *Proxy Re-Encryption (PRE)*. Re-Encryption consists in modifying an existing ciphertext such that a specific decryption key can no longer decrypt it. In SEA-BREW, as in other schemes [35], the Re-Encryption is “proxied” because it is del-

egated to the cloud server, which thus acts as a full-resource proxy for the producers. The cloud server, however, does this blindly, that is without accessing the plaintext of the messages. This makes our scheme resilient to possible data leakage on the cloud server. Our PRE mechanism is also “lazy”, which means that the ciphertext is modified not immediately after the key revocation, but only when it is downloaded by some consumer. This allows us to spread over time the computational costs sustained by the cloud server for the PRE operations. We implement the lazy PRE scheme by assigning a version to the encryption key, to each decryption key, and to each ciphertext. When a key is revoked, the key authority modifies the encryption key, increments its version, and uploads some update quantities to the cloud server. The set of these update quantities is called *update key*. The update key is used by the cloud server to blindly re-encrypt the ABE ciphertexts and increment their version before sending them to the requesting consumers. The cloud server also uses the update key to update the encryption key used by producers, and the decryption keys used by consumers. Inside the low-bitrate WSN, instead, the update of the WSN consumers’ decryption keys is achieved with a *constant-ciphertext broadcast encryption scheme*, like the one shown in Boneh et al.’s work [5]. The broadcast encryption scheme allows the WSN gateway to broadcast the update key encrypted in such a way to exclude

EK	Encryption key
MK	Master key
DK	Decryption key
KDK	Key distribution key
PID	Producer identifier
SK	Signature verification key
CID	Consumer identifier
KID	Symmetric key identifier
$SymKey$	Symmetric key
\mathcal{P}	Access policy
γ	Attribute set
BPK	Broadcast public key
d_{CID}	Broadcast private key
CP	Ciphertext
U	Update key
M	Message

Table 1 Table of Symbols

one or more WSAN consumers from decrypting it. To do this, the WSAN gateway uses a *broadcast public key* (BPK), and each WSAN consumer uses its own *broadcast private key* (d_{CID}). Table 1 lists the symbols used in the paper.

3.1 SEA-BREW System Procedures

Our system makes use of a set of *cryptographic primitives* (from now on, simply primitives), which are the following ones.

$(MK, EK) = \mathbf{Setup}(\kappa)$: This primitive initializes the cryptographic scheme. It takes a security parameter κ as input, and outputs a *master key* MK and an associated *encryption key* EK .

$CP = \mathbf{Encrypt}(M, \mathcal{P}, EK)$: This primitive encrypts a plaintext M under the policy \mathcal{P} . It takes as input the message M , the encryption key EK , and the policy \mathcal{P} . It outputs the ciphertext CP .

$DK = \mathbf{KeyGen}(\gamma, MK)$: This primitive generates a decryption key. It takes as input a set of attributes γ which describes the consumer, and the master key MK . It outputs a decryption key DK , which is composed of two fields for each attribute in γ , plus a field called D , useful to update such a key.

$M = \mathbf{Decrypt}(CP, DK)$: This primitive decrypts a ciphertext CP . It takes the ciphertext CP and the consumer's decryption key DK as input, and outputs the message M if decryption is successful, \perp otherwise. The decryption is successful if and only if γ satisfies \mathcal{P} , which is embedded in CP .

The following primitives use symbols with a superscript number to indicate the version of the associated quantity. For example, $MK^{(i)}$ indicates the i -th version of the master key, $DK^{(i)}$ indicates the i -th version of a given decryption key, etc.

$(MK^{(i+1)}, U^{(i+1)}) = \mathbf{UpdateMK}(MK^{(i)})$: This primitive updates the master key from a version i to the version $i + 1$ after a key revocation. It takes as input the old master key $MK^{(i)}$, and it outputs an updated master key $MK^{(i+1)}$, and the $(i + 1)$ -th version of the update key $U^{(i+1)}$. Such an update key is composed of the quantities $U_{EK}^{(i+1)}$, $U_{DK}^{(i+1)}$, $U_{CP}^{(i+1)}$, which will be used after a key revocation respectively to update the encryption key, to update the decryption keys, and to re-encrypt the ciphertexts.

$EK^{(n)} = \mathbf{UpdateEK}(EK^{(i)}, U_{EK}^{(n)})$: This primitive updates an encryption key from a version i to the latest version n , with $n > i$, after a key revocation. The primitive takes as input the old encryption key $EK^{(i)}$ and $U_{EK}^{(n)}$, and it outputs the updated encryption key $EK^{(n)}$.

$D^{(n)} = \mathbf{UpdateDK}(D^{(i)}, U_{DK}^{(i)}, U_{DK}^{(i+1)}, \dots, U_{DK}^{(n)})$: This primitive updates a decryption key from a version i to the latest version n , with $n > i$, after a key revocation. What is actually updated is not the whole decryption key, but only a particular field D inside the decryption key. This allows the cloud server to execute the primitive without knowing the whole decryption key, but only D which alone is useless for decrypting anything. The primitive takes as input the old field $D^{(i)}$ and $U_{DK}^{(i)}, U_{DK}^{(i+1)}, \dots, U_{DK}^{(n)}$, and it outputs the updated field $D^{(n)}$.

$CP^{(n)} = \mathbf{UpdateCP}(CP^{(i)}, U_{CP}^{(i)}, U_{CP}^{(i+1)}, \dots, U_{CP}^{(n)})$: This primitive updates a ciphertext from a version i to the latest version n , with $n > i$, after a key revocation. The cloud server executes this primitive to perform proxy re-encryption on ciphertexts. The primitive takes as input the old ciphertext $CP^{(i)}$, and $U_{CP}^{(i)}, U_{CP}^{(i+1)}, \dots, U_{CP}^{(n)}$. It outputs the updated ciphertext $CP^{(n)}$.

The implementation of these primitives will be described in detail in Section 3.2. In the following, we describe the procedures that our system performs.

3.1.1 System Initialization

The system initialization procedure is executed only once, to start the system, and it consists in the fol-

lowing steps.

Step 1. The key authority runs the Setup primitive, thus obtaining the first version of the master key ($MK^{(0)}$) and the first version of the encryption key ($EK^{(0)}$). We indicate with v_{MK} (*master key version*) the current version of the master key. The key authority initializes the master key version to $v_{MK} = 0$, and it sends the encryption key and the master key version to the cloud server with a signed message.

Step 2. The cloud server, in turn, sends the encryption key and the master key version to the WSAN gateway with a signed message.

Step 3. The WSAN gateway generates the broadcast public key (see Figure 1) for the broadcast encryption scheme.

3.1.2 Producer Join

The consumer join procedure is executed whenever a new producer joins the system. We assume that the producer has already pre-installed its own pair of asymmetric keys that it will use for digital signatures. Alternatively the producer can create such a pair at the first boot. We call *signature verification key* (SK , see Figure 1) the public key of such a pair. The procedure consists in the following steps.

Step 1. The producer sends the signature verification key to the key authority in some authenticated fashion. The mechanism by which this is done falls outside the scope of the paper. For example, in case the producer is a sensor, the human operator who is physically deploying the sensor can leverage a pre-shared password with the key authority.

Step 2. The key authority assigns a new producer identifier to the producer, and it sends such an identifier and the encryption key to the producer with a signed message. The encryption key embeds an *encryption key version* (v_{EK}), which represents the current version of the encryption key locally maintained by the producer. Initially, the encryption key version is equal to the master key version ($v_{EK} = v_{MK}$).

Step 3. The key authority also sends the producer's identifier, signature verification key and encryption key version to the cloud server with a signed message. The cloud server adds a tuple $\langle PID, SK, v_{EK} \rangle$ to a locally maintained *Producer Table* (PT, see Figure 1). Each tuple in the PT represents a producer in the system.

If the producer is remote, then the procedure ends here. Otherwise, if the producer is inside the WSAN, then the following additional steps are performed.

Step 4. The key authority sends the producer identifier and the signature verification key to the WSAN gateway with a signed message. The WSAN gateway adds

a tuple $\langle PID, SK \rangle$ to a locally maintained *WSAN Signature Table* (see Figure 1). Each tuple in the WSAN Signature Table represents a producer in the WSAN. Through this table, both the gateway and the consumers are able to authenticate data and messages generated by the producers in the WSAN.

Step 5. The WSAN gateway finally broadcasts the signed message received from the key authority to all the WSAN consumers. The WSAN consumers add the same tuple $\langle PID, SK \rangle$ to a locally maintained copy of the WSAN Signature Table.

3.1.3 Consumer Join

The consumer join procedure is executed whenever a new consumer, described by a given attribute set, joins the system. We assume that the consumer has already pre-installed its own pair of asymmetric keys that it will use for asymmetric encryption. Alternatively the consumer can create such a pair at the first boot. We call *key distribution key* (KDK , see Figure 1) the public key of such a pair. The procedure consists in the following steps.

Step 1. The consumer sends the key distribution key to the key authority in some authenticated fashion. Again, the mechanism by which this is done falls outside the scope of the paper.

Step 2. The key authority assigns a new consumer identifier to the consumer, and it generates a decryption key with the *KeyGen* primitive, according to the consumer's attribute set. The key authority sends the consumer identifier and the decryption key to the consumer with a signed message, encrypted with the consumer's key distribution key.

Step 4. The key authority sends the consumer identifier and the field D of the decryption key to the cloud server with a signed message. The cloud server initializes a *decryption key version* (v_{DK}), which represents the current version of the consumer's decryption key, to the value of the master key version. The cloud server adds a tuple $\langle CID, D, v_{DK} \rangle$ to a locally maintained *Consumer Table* (CT, see Figure 1). Each tuple in the CT represents a consumer in the system.

If the consumer is remote, then the procedure ends here. Otherwise, if the consumer is a WSAN consumer, then the following additional steps are performed.

Step 5. The key authority sends the consumer identifier and the key distribution key to the WSAN gateway with a signed message.

Step 6. The WSAN gateway sends the WSAN Signature Table to the consumer with a signed message, along with the broadcast public key and the consumer's broadcast private key which is appropriately encrypted

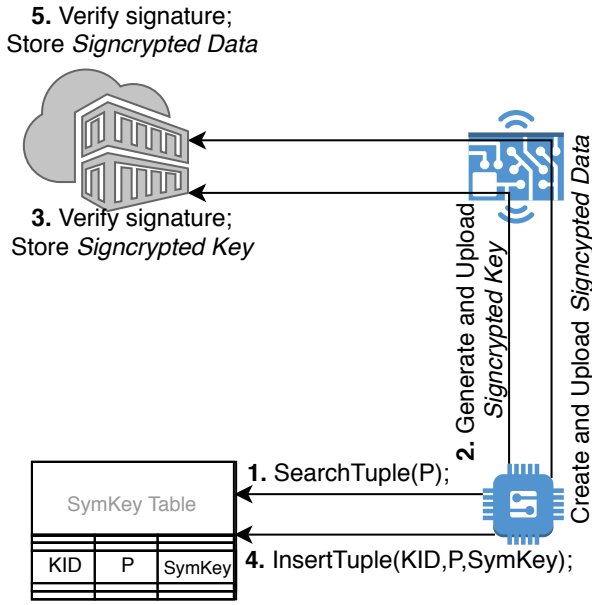


Fig. 2 Data upload by WSAN producers procedure.

with the consumer's key distribution key. Finally, the WSAN gateway adds a tuple $\langle CID, KDK \rangle$ to a locally maintained *WSAN Consumer Table*.

3.1.4 Data Upload by Remote Producers

The data upload procedure is executed whenever a producer wants to upload data to the cloud server. Remote producers and WSAN producers perform two different procedures to upload a piece of information to the cloud server. We explain them separately. The data upload procedure by remote producers consists in the following steps.

Step 1. Let \mathcal{P} be the access policy that has to be enforced over the data. The remote producer encrypts the data under such a policy using the *Encrypt* primitive. The resulting ciphertext has the same version number of the producer's locally maintained encryption key ($v_{CP} = v_{EK}$).

Step 2. The producer securely deletes the original data. Then it signs and uploads the ciphertext to the cloud server, along with its producer identifier.

Step 3. The cloud server verifies the signature, and then it stores the ciphertext.

Finally, if the ciphertext version is older than the master key version, the cloud server executes the remote producer update procedure (see Section 3.1.10).

3.1.5 Data Upload by WSAN Producers

SEA-BREW aims at saving bandwidth in the WSAN also during data upload. However, encrypting data directly with the *Encrypt* primitive introduces a lot of overhead in terms of data size, as it happens in the typical ABE scheme. Therefore, we want to obtain the access control mechanism provided by the *Encrypt* primitive, and at the same time producing the small ciphertexts typical of symmetric-key encryption. Broadly speaking, we achieve this by encrypting a symmetric key using the *Encrypt* primitive, and then using such a symmetric key to encrypt all the data that must be accessible with the same access policy. To do this, each WSAN producer maintains a *SymKey Table* (see Figure 1), which associates policies \mathcal{P} to symmetric keys *SymKey*. More specifically, the *SymKey Table* is composed of tuples in the form $\langle KID, \mathcal{P}, SymKey \rangle$, where *KID* is the *symmetric key identifier* of *SymKey*. The symmetric key identifier uniquely identifies a symmetric key in the whole system. The data upload procedure by WSAN producers consists in the following steps (Figure 2).

Step 1. Let \mathcal{P} be the access policy that has to be enforced over the data. The producer searches for a tuple inside its *SymKey Table* associated with the policy. If such a tuple already exists, then the producer jumps directly to Step 4, otherwise it creates it by continuing to Step 2.

Step 2. The producer randomly generates a symmetric key and a symmetric key identifier. The symmetric key identifier must be represented on a sufficient number of bits to make the probability that two producers choose the same identifier for two different symmetric keys negligible. The producer then encrypts the symmetric key under the policy using the *Encrypt* primitive, and it signs the resulting ciphertext together with the key identifier. The result is the *signcrypt key*. The producer uploads the signcrypt key and its producer identifier to the cloud server.

Step 3. The cloud server verifies the signature, and then it stores the signcrypt key in the same way it stores ordinary encrypted data produced by remote producers.

Step 4. The producer inserts (or retrieves, if steps 2 and 3 have not been executed) the tuple $\langle KID, \mathcal{P}, SymKey \rangle$ into (from) its *SymKey Table*, and it encrypts the data using the symmetric key associated to the policy. Then, the producer signs the resulting ciphertext together with the symmetric key identifier. The result is the *signcrypt data*. The producer uploads the signcrypt data and its producer identifier to the cloud server, and it securely deletes the original

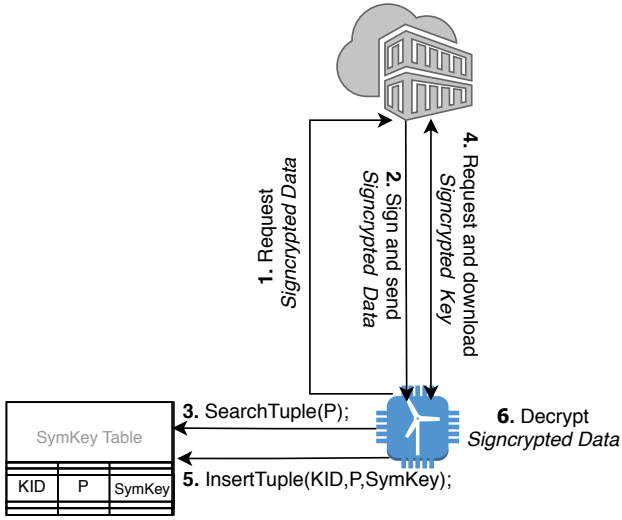


Fig. 3 Download signcrypted data procedure.

data.

Step 5. The cloud server verifies the signature, and then it stores the signcrypted data.

3.1.6 Data Download

The data download procedure is executed whenever a consumer wants to download data from the cloud server. Consumers perform two different procedures to download a piece of information from the cloud server, depending whether such piece of information has been produced by a remote producer or by a WSAN producer. We explain them separately. The download procedure of data produced by remote producers consists in the following steps.

Step 1. The consumer sends a data request along with its consumer identifier to the cloud server.

Step 2. The cloud server checks in the CT whether the decryption key version of the consumer is older than the master key version and, if so, it updates the decryption key by executing the remote consumer update procedure (see after). The cloud server identifies the requested ciphertext and checks whether its version is older than the master key version. If so, the cloud server updates the ciphertext by executing the UpdateCP primitive (see Section 3.2).

Step 3. The cloud server signs and sends the requested data to the consumer.

Step 4. The consumer verifies the server signature over the received message. Then, it executes the Decrypt primitive using its decryption key.

Now consider the case in which a consumer requests a data produced by a WSAN producer. Each consumer maintains a SymKey Table (see Figure 1), which asso-

ciates policies \mathcal{P} to symmetric keys $SymKey$. The download procedure of data produced by WSAN producers consists in the following steps (Figure 3).

Step 1. The consumer sends a data request along with its consumer identifier to the cloud server.

Step 2. The cloud server signs and sends the requested signcrypted data to the consumer.

Step 3. The consumer searches for a tuple with the same key identifier as the one contained in the received signcrypted data inside its SymKey Table. If such a tuple already exists, then the consumer jumps directly to Step 6, otherwise the consumer creates it by continuing to Step 4.

Step 4. The consumer performs a data download procedure, requesting and obtaining the signcrypted key associated to the received symmetric key identifier.

Step 5. The consumer decrypts the signcrypted key thus obtaining the symmetric key, and it adds the tuple $\langle KID, \mathcal{P}, SymKey \rangle$ to its SymKey Table.

Step 6. The consumer decrypts the signcrypted data with the symmetric key.

3.1.7 Direct Data Exchange

The direct data exchange procedure is executed whenever a producer wants to transmit data to one or more consumers in a low-latency fashion inside the WSAN. To obtain a low latency the producer broadcasts the data directly to the authorized consumers in an encrypted form, instead of uploading such data to the cloud server. Furthermore, to save WSAN bandwidth we want that the data exchanged is encrypted with symmetric-key encryption, under the form of signcrypted data as it happens for data uploaded by WSAN producers. To ease the reading we assume that the producer has already a tuple associated to the policy it wants to apply. Otherwise the producer should previously perform a data upload procedure to the cloud in which it uploads the signcrypted key it will use.

The procedure consists in the following steps.

Step 1. Let \mathcal{P} be the access policy that has to be enforced over the data. The producer retrieves the symmetric key associated to such policy inside its SymKey Table. The producer encrypts the data with such a symmetric key, and signs it together with the symmetric key identifier. It thus obtains the signcrypted data.

Step 2. The producer broadcasts the signcrypted data in the WSAN, and securely deletes the original data.

Step 3. Perform Steps 3-6 of the download procedure of data produced by WSAN producers.

3.1.8 Producer Leave

The producer leave procedure is executed whenever one or more producers leave the system. This happens in case that producers are dismissed from the system, or the private keys that they use for signatures are compromised. In all these cases, the private keys of the leaving producers must be revoked, so that data signed with such keys is no longer accepted by the cloud server. The procedure consists in the following steps.

Step 1. The key authority communicates to the cloud server the identifiers of the leaving producers with a signed message.

Step 2. The cloud server removes the tuples associated to such identifiers from the PT.

If at least one leaving producer was a WSAN producer, the following additional steps are performed.

Step 3. The key authority communicates the identifiers of the leaving WSAN producers to the WSAN gateway with a signed message.

Step 4. The WSAN gateway removes the tuples associated to such identifiers from the WSAN Signature Table, and it broadcasts the signed message received by the key authority to all the WSAN consumers.

Step 5. The WSAN consumers remove the tuples associated to such identifiers from their locally maintained copy of the WSAN Signature Table.

3.1.9 Consumer Leave

The consumer leave procedure is executed whenever one or more consumers leave the system, as depicted in figure 4. This happens in case that consumers are dismissed from the system, or their keys are compromised. In all these cases, the decryption keys of the leaving consumers must be revoked, in such a way that they cannot decrypt data anymore. The procedure consists in the following steps.

Step 1. The key authority increases the master key version, and it executes the UpdateMK primitive on the old master key, thus obtaining the new master key and the quantities $U_{EK}^{(v_{MK})}$, $U_{DK}^{(v_{MK})}$, and $U_{CP}^{(v_{MK})}$. Then, the key authority sends the identifiers of the leaving consumers and the quantities $U_{EK}^{(v_{MK})}$, $U_{DK}^{(v_{MK})}$, and $U_{CP}^{(v_{MK})}$ to the cloud server with a signed message, encrypted with the cloud server's public key.

Step 2. The cloud server verifies the signature, decrypts the message, retrieves the consumer identifier from the message, and removes the tuples associated to those identifiers from the CT. Note that the cloud server could now re-encrypt all the ciphertexts, by using the quantity $U_{CP}^{(v_{MK})}$ just received. However, the re-encryption of each ciphertext is deferred to the time

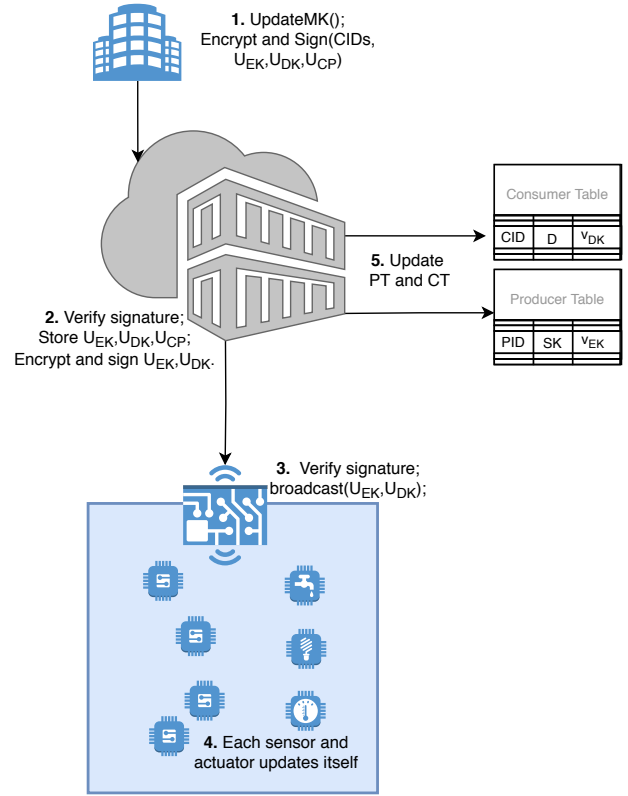


Fig. 4 Consumer leave procedure.

at which a consumer requests it (Lazy PRE). Then, the cloud server signs and encrypts $U_{EK}^{(v_{MK})}$ and $U_{DK}^{(v_{MK})}$ with asymmetric encryption, and it sends them to the gateway.

Step 3. The gateway broadcasts the quantity $U_{EK}^{(v_{MK})}$ and $U_{DK}^{(v_{MK})}$ over the local low-bitrate WSAN, so that all the producers and consumers that belong to it can immediately update their encryption key and decryption key, respectively. To do this the gateway sends a single broadcast message, composed as follows. The gateway encrypts the $U_{DK}^{(v_{MK})}$ quantity with the broadcast public key, in such a way that all the WSAN consumers except the leaving ones can decrypt it. This allows the gateway to share said quantity only with the WSAN consumers, excluding the compromised ones if there are any. The gateway then signs the concatenation of the quantity $U_{EK}^{(v_{MK})}$, and the quantity $U_{DK}^{(v_{MK})}$ (encrypted), and broadcasts said message over the WSAN.

Step 4. Each producer updates its encryption key upon receiving the broadcast message; each consumer then decrypts the received message using its broadcast private key d_{CID} , and executes the UpdateDK primitive using its old decryption key and the just received $U_{DK}^{(v_{MK})}$. The WSAN producers and the consumers delete their SymKey Tables.

Step 5. The cloud server updates inside the PT the versions of the encryption keys of all the WSAAN producers, and inside the CT the versions of the decryption keys of all the WSAAN consumers.

Note that SEA-BREW updates all the devices inside the low-bitrate WSAAN with a single $\mathcal{O}(1)$ -sized broadcast message (Step 3). This makes SEA-BREW highly scalable in the number and size of messages necessary to manage decryption keys. Note also that, regarding remote consumers and remote producers, the computational load of the consumer leave procedure is entirely delegated to the cloud server, leaving the producers and consumers free of heavy computation. This enables SEA-BREW to run on a broader class of sensors and actuators.

3.1.10 Remote Producer Update

The producer update procedure is executed by the data upload procedure by remote producers (see Section 3.1.4), and it consists in the following steps. **Step 1.** The cloud server signs and sends the last quantity U_{EK} received from the key authority to the remote producer that must be updated.

Step 2. The producer verifies the signature and retrieves U_{EK} . Then, it executes the UpdateEK primitive using its encryption key and the received quantity U_{EK} as parameters.

Step 3. The cloud server updates the producer's encryption key version to v_{MK} inside PT.

3.1.11 Remote Consumer Update

The consumer update procedure is executed as specified in the data download procedure (see Section 3.1.6), and it consists in the following steps.

Step 1. The cloud server executes the UpdateDK primitive using the consumer's decryption key and the last $(v_{MK} - v_{DK})$ quantities U_{DKs} received from the key authority. The cloud server encrypts and signs the output of that primitive, $D^{(v_{MK})}$ using the consumer's key-encryption key, and sends it to the consumer.

Step 2. The consumer verifies the signature and decrypts the message, thus obtaining back $D^{(v_{MK})}$. Then, the consumer replaces the old field D of its decryption key with the received quantity.

Step 3. The cloud server updates the consumer's decryption key version to v_{MK} inside CT.

3.2 SEA-BREW primitives

We now explain in detail how the CP-ABE primitives previously introduced at the beginning of Section 3.1

are realized.

$$(MK^{(0)}, EK^{(0)}) = \text{Setup}(\kappa)$$

The Setup primitive is executed by the key authority. This primitive computes:

$$EK^{(0)} = \{G_0, g, h = g^\beta, l = e(g, g)^\alpha, v_{EK} = 0\} \quad (1)$$

$$MK^{(0)} = \{\beta, g^\alpha, v_{MK} = 0\} \quad (2)$$

Where G_0 is a multiplicative cyclic group of prime order p with size κ , g is the generator of G_0 , $e : G_0 \times G_0 \rightarrow G_1$ is an efficiently-computable bilinear map with bi-linearity and non-degeneracy properties, and $\alpha, \beta \in \mathbb{Z}_p$ are chosen at random.

$$CP = \text{Encrypt}(M, \mathcal{P}, EK^{v_{EK}})$$

The Encrypt primitive is executed by a producer. From now on, \mathcal{P} is represented as a *policy tree*, which is a labeled tree where the non-leaf nodes implement *threshold-gate operators* whereas the leaf nodes are the attributes of the policy. A threshold-gate operator is a Boolean operator of the type *k-of-n*, which evaluates to true iff at least k (*threshold value*) of the n inputs are true. Note that a 1-of- n threshold gate implements an OR operator, whereas an n -of- n threshold gate implements an AND operator. For each node x belonging to the policy tree the primitive selects a polynomial q_x of degree equal to its threshold value minus one ($d_x = k_x - 1$). The leaf nodes have threshold value $k_x = 1$, so their polynomials have degree equal to $d_x = 0$. The polynomials are chosen in the following way, starting from the root node R . The primitive assigns arbitrarily an index to each node inside the policy tree. The index range varies from 1 to num , where num is the total number of the nodes. The function $\text{index}(x)$ returns the index assigned to the node x . Starting with the root node R the primitive chooses a random $s \in \mathbb{Z}_p$ and sets $q_R(0) = s$. Then, it randomly chooses d_R other points of the polynomial q_R to completely define it. Iteratively, the primitive sets $q_x(0) = q_{\text{parent}(x)}(\text{index}(x))$ for any other node x and randomly chooses d_x other points to completely define q_x , where $\text{parent}(x)$ refers to the parent of the node x . At the end, the ciphertext is computed as follows:

$$CP = \{\mathcal{P}, \tilde{C} = Me(g, g)^{\alpha s}, C = h^s, v_{CP} = v_{EK} \\ \forall y \in Y : C_y = g^{q_y(0)}, C'_y = H(\text{att}(y))^{q_y(0)}\}, \quad (3)$$

where Y is the set of leaf nodes of the policy tree. The function $\text{att}(x)$ is defined only if x is a leaf node, and it denotes the attribute associated with the leaf. H is

a hash function $H : \{0,1\}^* \rightarrow \mathbb{G}_0$ that is modeled as a random oracle. The encryption key version v_{EK} is assigned to the ciphertext version v_{CP} .

$$DK = \text{KeyGen}(MK^{v_{MK}}, \gamma)$$

The KeyGen primitive is executed by the key authority. This primitive randomly selects $r \in \mathbb{Z}_p$, and $r_j \in \mathbb{Z}_p$ for each attribute in γ . It computes the decryption key DK as:

$$DK = \{D = g^{\frac{(\alpha+r)}{\beta}}, v_{DK} = v_{MK} \mid \forall j \in \gamma : D_j = g^r \cdot H(j)^{r_j}, D'_j = g^{r_j}\} \quad (4)$$

$$M = \text{Decrypt}(CP, DK)$$

The Decrypt primitive is executed by a consumer. This primitive executes the sub-function *DecryptNode* on the root node. *DecryptNode*(DK, CP, x) takes as input the consumer's decryption key, the ciphertext and the node x . If the node x is a leaf node, let $i = \text{att}(x)$ and define the function as follows. If $i \in \gamma$, then:

$$\begin{aligned} \text{DecryptNode}(DK, CP, x) &= \frac{e(D_i, C_x)}{e(D'_i, C'_x)} \\ &= \frac{e(g^r \cdot H(i)^{r_i}, g^{q_x(0)})}{e(g^{r_i}, H(i)^{q_x(0)})} \\ &= \frac{e(g^r, g^{q_x(0)}) \cdot e(H(i)^{r_i}, g^{q_x(0)})}{e(H(i)^{r_i}, g^{q_x(0)})} \\ &= e(g, g)^{r \cdot q_x(0)} \end{aligned} \quad (5)$$

Otherwise, if $i \notin \gamma$, then *DecryptNode*(DK, CP, x) = \perp . When x is not a leaf node, the primitive proceeds as follows. First of all, let $\Delta_{i,S}$ be the Lagrange coefficient for $i \in \mathbb{Z}_p$ and let S be an arbitrary set of element in \mathbb{Z}_p : $\Delta_{i,S}(x) = \prod_{j \in S, j \neq i} \frac{x-j}{i-j}$. Now, for all nodes z that are children of x , it calls recursively itself and stores the result as F_z . Let S_x be an arbitrary k_x -sized set of children z such that $F_z \neq \perp \forall z \in S_x$. Then, the function computes:

$$\begin{aligned} F_x &= \prod_{z \in S_x} F_z^{\Delta_{i,S'_x}(0)} \\ &= \prod_{z \in S_x} (e(g, g)^{r \cdot q_z(0)})^{\Delta_{i,S'_x}(0)} \\ &= \prod_{z \in S_x} (e(g, g)^{r \cdot q_{\text{parent}(z)}(\text{index}(z))})^{\Delta_{i,S'_x}(0)} \\ &= \prod_{z \in S_x} e(g, g)^{r \cdot q_x(i) \cdot \Delta_{i,S'_x}(0)} \\ &= e(g, g)^{r \cdot q_x(0)} \end{aligned} \quad (6)$$

where $i = \text{index}(z)$, and $S_x = \{\text{index}(z) : z \in S_x\}$. The Decrypt(CP, DK) primitive first calls *DecryptNode*(DK, CP, R) where R is the root of the policy tree extracted by \mathcal{P} embedded in CP . Basically, the sub-function navigates the policy tree embedded inside the ciphertext in a top-down manner and if γ satisfies the policy tree it returns $A = e(g, g)^{rs}$. Finally, the primitive computes:

$$\begin{aligned} \tilde{C}/(e(C, D)/A) &= \tilde{C}/(e(h^s, g^{\frac{\alpha+r}{\beta}})/e(g, g)^{rs}) \\ &= Me(g, g)^{\alpha s} / (e(g, g)^{\beta s \cdot \frac{\alpha+r}{\beta}} / e(g, g)^{rs}) \\ &= \frac{Me(g, g)^{\alpha s}}{e(g, g)^{\alpha s}} = M \end{aligned} \quad (7)$$

$$(MK^{(v_{MK}+1)}, U^{(v_{MK}+1)}) = \text{UpdateMK}(MK^{(v_{MK})})$$

The UpdateMK primitive is executed by the key authority. This primitive increments v_{MK} by one, chooses at random a new $\beta^{(v_{MK})} \in \mathbb{Z}_p$, and computes:

$$\begin{aligned} U_{CP}^{(v_{MK})} &= \frac{\beta^{(v_{MK})}}{\beta^{(v_{MK}-1)}}; \\ U_{EK}^{(v_{MK})} &= g^{\beta^{(v_{MK})}}; \\ U_{DK}^{(v_{MK})} &= \frac{\beta^{(v_{MK}-1)}}{\beta^{(v_{MK})}}; \\ U^{(v_{MK})} &= \{U_{CP}^{(v_{MK})}, U_{EK}^{(v_{MK})}, U_{DK}^{(v_{MK})}\}. \end{aligned} \quad (8)$$

Then it updates the master key as:

$$MK^{(v_{MK})} = \{\beta^{(v_{MK})}, g^\alpha, v_{MK}\} \quad (9)$$

In order to avoid ambiguities, we specify that the first ever update key is $U^{(1)}$ and not $U^{(0)}$ as the value v_{MK} is incremented *before* the creation of U . The careful reader surely have noticed that U_{CP} and U_{DK} are reciprocal. In practice, we can use only one of these quantities and compute the other by inverting it. In this paper we chose to keep those quantity separated for the sake of clarity.

$$EK^{(v_{MK})} = \text{UpdateEK}(EK^{(v_{EK})}, U_{EK}^{(v_{MK})})$$

The UpdateEK primitive is executed by the producers. Regardless the input encryption key's version, this primitive takes as input only the last update key generated, namely $U_{EK}^{(v_{MK})}$. The primitive substitutes the field h inside the encryption key with the last update quantity, and updates the encryption key version to the latest master key version, thus obtaining:

$$EK^{(v_{MK})} = \{\mathbb{G}_0, g, h = U_{EK}^{(v_{MK})}, l = e(g, g)^\alpha, v_{EK} = v_{MK}\}. \quad (10)$$

$$D^{(v_{MK})} = \text{UpdateDK}(U_{DK}^{(v_{DK}+1)}, \dots, U_{DK}^{(v_{MK})}, D^{(v_{DK})})$$

The UpdateDK primitive is executed by the cloud server and by the WSAW consumers. The decryption key on input has been lastly updated with $U_{DK}^{(v_{DK})}$, and the overall latest update is $U_{DK}^{(v_{MK})}$, with, $v_{MK} > v_{DK}$. This primitive computes:

$$U'_{DK} = U_{DK}^{(v_{DK}+1)} \dots U_{DK}^{(v_{MK})}$$

$$D^{(v_{MK})} = (D^{(v_{DK})})^{U'_{DK}} = g^{\frac{r+\alpha}{\beta^{(v_{DK})}} \cdot \frac{\beta^{(v_{DK})}}{\beta^{(v_{MK})}}} = g^{\frac{r+\alpha}{\beta^{(v_{MK})}}}. \quad (11)$$

$$CP^{(v_{MK})} = \text{UpdateCP}(CP^{(v_{CP})}, U_{CP}^{(v_{CP}+1)}, \dots, U_{CP}^{(v_{MK})})$$

The UpdateCP primitive is executed by the cloud server. The ciphertext on input has been lastly re-encrypted with $U_{CP}^{(v_{CP})}$, and the overall latest update is $U_{CP}^{(v_{MK})}$, with, $v_{MK} > v_{CP}$. This primitive computes the re-encryption quantity U'_{CP} as the multiplication of all the version updates successive to the one in which the ciphertext has been lastly updated.

$$U'_{CP} = U_{CP}^{(v_{CP}+1)} \dots U_{CP}^{(v_{MK})} \quad (12)$$

Then, re-encryption is achieved with the following computation:

$$C^{(v_{MK})} = (C^{(v_{CP})})^{U'_{CP}} = g^{\beta^{(v_{MK})} s} \quad (13)$$

Finally, the primitive outputs the re-encrypted ciphertext CP' as:

$$CP^{(v_{MK})} = \{\mathcal{P}, \tilde{C}, C^{(v_{MK})}, v_{CP} = v_{MK},$$

$$\forall y \in Y : C_y = g^{q_y(0)}, C'_y = H(\text{att}(y))^{q_y(0)}\} \quad (14)$$

4 Security Analysis

4.1 Threat Model

In this section, we model a set of adversaries and we analyze the security of our system against them. In particular, we consider the following adversaries: (i) an *external adversary*, which does not own any cryptographic key except the public ones; (ii) a *device compromiser*, which can compromise sensors and actuators to steal secrets from them; (iii) a set of *colluding consumers*, which own some decryption keys; and (iv) a *honest-but-curious cloud server* as defined in [35, 25, 9], which does not tamper with data and correctly executes the procedures, but it is interested in accessing data. We

assume that the honest-but-curious cloud server might collude also with a set of consumers, which own some decryption keys. Note that the honest-but-curious cloud server models also an adversary capable of *breaching* the cloud server, meaning that he can steal all the data stored in it. In order to do this, he can leverage some common weakness, for example buffer overflows or code injections, or hardware vulnerabilities like Meltdown or Spectre [26]. We assume that who breaches the cloud server only steals data and does not alter its behavior in correctly executing all the protocols, basically because he tries to remain as stealth as possible during the attack. Note that this reflects real-life attacks against cloud servers¹. In the following we analyze in detail each adversary model.

The external adversary aims at reading or forging data. To do so, he can adopt several strategies. During the new producer join procedure, he can impersonate the key authority to communicate a false encryption key to the producer, so that the data encrypted by said producer will be accessible by the adversary. This attack is avoided because the encryption keys are signed by the key authority. Alternatively, during the new consumer join procedure, the external adversary can act as a man in the middle between the key authority and the joining consumer in order to steal the consumer's decryption key, with which he can later decrypt data. This attack is avoided because the decryption key is encrypted by the key authority with asymmetric encryption. Using the encryption key, which is public, the external adversary may also try to encrypt false data and upload it to the cloud server. This attack is avoided because he cannot forge a valid signature for the encrypted data, thus he cannot make the false data be accepted as valid by the legitimate consumers. To sum up, the external adversary cannot access legitimate data neither inject malicious data.

The device compromiser can compromise a producer or a consumer. If he compromises a producer, then he gains full control of such a device and full access to its sensed data and to its private key used for signatures. Note that he cannot retrieve any data sensed before the compromise, because the producer securely deletes data after having uploaded it to the cloud server. Nonetheless, he can indeed inject malicious data into the system, by signing it and uploading it to the cloud server, or by transmitting it directly to WSAW consumers if the compromised producer belongs to the WSAW. When the key authority finds out the compromise (the method by which this happens falls outside the scope of the paper), it executes a producer leave procedure. After that, the compromised producer cannot inject malicious data

¹ <https://www.bbc.com/news/technology-41147513>

anymore because the private key that it uses for signatures is not considered valid anymore by the consumers. On the other hand, if the adversary compromises a consumer, then he gains full access to its decryption key. The attacker can decrypt *some* data downloaded from the cloud server or, if the compromised a consumer belonging to the WSAN, transmitted directly by WSAN producers. Notably, the adversary can decrypt only data that the compromised consumer was authorized to decrypt. When the key authority finds out the compromise, it executes a consumer leave procedure. After that, the compromised consumer cannot decrypt data anymore. The reason for this is that our re-encryption mechanism updates the ciphertexts as if they were encrypted with a different encryption key, in particular with a different β parameter.

A set of colluding consumers can try to combine somehow their decryption keys to decrypt some data that singularly they cannot decrypt. However, even if the union of the attribute sets of said decryption keys satisfies the access policy of a ciphertext, the colluding consumers cannot decrypt such a ciphertext. In Section 4.2 we will capture this adversary model with the Game 1, and we will provide a formal proof that SEA-BREW is resistant against it.

The honest-but-curious cloud server does not have access to data because it is encrypted, but it can access all the update keys and all the field D of the consumers' decryption keys. The update keys alone are useless to decrypt data because the cloud server lacks of a (complete) decryption key. However, if the cloud server colludes with a set of consumers, then it can access all the data that the consumers are authorized to decrypt. Interestingly, if the honest-but-curious cloud server is modelling an adversary capable of breaching the cloud server, recovering the breach is easy. It is sufficient to execute a consumer leave procedure *of zero consumers*. This has the effect of updating the master key without revoking any particular decryption key, making thus all the stolen update keys useless. On the other hand, in case of an *actual* honest-but-curious cloud server, executing a consumer leave procedure does not solve the problem, because the cloud server knows the just generated update key and thus it can update the revoked decryption keys by executing the UpdateDK primitive. In any case, the honest-but-curious cloud server and the colluding consumers cannot combine somehow the update keys and decryption keys to decrypt some data that singularly the colluding consumers cannot decrypt. In Section 4.2 we will capture this adversary model with the Game 2, and we will provide a formal proof that SEA-BREW is resistant against it.

4.2 Security Proofs

In this section, we provide formal proofs of two security properties of our scheme, related to two adversary models described in the previous section. Namely, we prove our scheme to be secure against a set of colluding consumers (Theorem 1), and against a honest-but-curious cloud server colluding with a set of consumers (Theorem 2).

We state that a scheme is secure against an adaptive chosen plaintext attack (IND-CPA) by a set of colluding consumers if no probabilistic polynomial-time (PPT) adversary \mathcal{A} has a non-negligible advantage against the challenger in the following game, denoted as Game 1.

Setup The challenger runs the Setup primitive and generates $EK^{(0)}$, and sends it to the adversary.

Phase 1 The adversary may issue queries for:

- *encryption key update*: the challenger runs the primitive UpdateMK. The challenger sends the updated encryption key to the adversary.
- *generate decryption key*: the challenger runs the primitive KeyGen using as input an attribute set provided by the adversary. Then, the challenger sends the generated decryption key to the adversary.
- *decryption key update*: the challenger runs the primitive UpdateDK using as input a decryption key provided by the adversary. Then, the challenger sends the updated decryption key to the adversary.
- *ciphertext update*: the challenger runs the primitive UpdateCP using as input a ciphertext provided by the adversary. Then, the challenger sends the ciphertext updated to the last version to the adversary.

Challenge The adversary submits two equal length messages m_0 and m_1 and a challenge policy \mathcal{P}^* , which is not satisfied by any attribute set queried as *generate decryption key* during Phase 1. The challenger flips a fair coin and assigns the outcome to b : $b \leftarrow \{0, 1\}$. Then, the challenger runs the Encrypt primitive encrypting m_b under the challenge policy \mathcal{P}^* using $EK^{(n)}$ and sends the ciphertext CP^* to the adversary. The symbol n is the last version of the master key, i.e., the number of time the adversary queried for an encryption key update.

Phase 2 Phase 1 is repeated. However the adversary cannot issue queries for *generate decryption key* whose attribute set γ satisfies the challenge policy \mathcal{P}^* .

Guess The adversary outputs a guess b' of b . The advantage of an adversary \mathcal{A} in Game 1 is defined as $\Pr[b' = b] - \frac{1}{2}$.

Theorem 1 *SEA-BREW is secure against an IND-CPA by a set of colluding consumers (Game 1), under the generic bilinear group model.*

Proof Our objective is to show that SEA-BREW is not less secure than the CP-ABE scheme by Bethencourt et al. [4], which is proved to be IND-CPA secure under the generic bilinear group model. To do this, we prove that if there is a PPT adversary \mathcal{A} that can win Game 1 with non-negligible advantage ϵ against SEA-BREW, then we can build a PPT simulator \mathcal{B} that can win the CP-ABE game described in [4] (henceforth, Game 0) against the scheme of Bethencourt et al. with the same advantage. We will denote the challenger of Game 0 as \mathcal{C} . We describe the simulator \mathcal{B} in the following.

Setup In this phase \mathcal{C} gives to \mathcal{B} the public parameters EK of Game 0, that will be exactly $EK^{(0)}$ in Game 1. In turn, \mathcal{B} sends to \mathcal{A} the encryption key $EK^{(0)}$ of Game 1.

Phase 1 Let us denote with the symbol n the latest version of the master key at any moment. In addition let us denote with the symbol k a specific version of a key or a ciphertext lower than n , so that $k < n$ at any moment. The query that an adversary can issue to the simulator are the following.

- *encryption key update*: \mathcal{B} chooses $U_{DK}^{(n+1)}$ at random from \mathbb{Z}_p . Then, \mathcal{B} computes

$$h^{(n+1)} = (g^{\beta^{(n)}})^{\frac{1}{U_{DK}^{(n+1)}}}, \quad (15)$$

and sends $EK^{(n+1)}$ to \mathcal{A} . Finally, \mathcal{B} increments n . Please note that \mathcal{B} does not know $\beta^{(i)}$, $\forall i \in [0, n]$, but it does not need to. \mathcal{B} needs to know only the relationship between any two consecutive versions, which are exactly:

$$U_{DK}^{(i)} = \frac{\beta^{(i-1)}}{\beta^{(i)}}, \forall i \in [1, n] \quad (16)$$

- *generate decryption key*: when \mathcal{A} issues a query for $DK_j^{(n)}$ (i.e., a decryption key with a given attribute set γ_j , and latest version n) to \mathcal{B} , \mathcal{B} in turn issues a query for DK_j to \mathcal{C} , and receives $DK_j^{(0)}$. Then \mathcal{B} upgrades such a key to the latest version n executing the primitive UpdateDK, using as input said key and $U_{DK}^{(i)}$, $\forall i \in [1, n]$. Finally \mathcal{B} sends to \mathcal{A} the desired decryption key $DK_j^{(n)}$.

- *decryption key update*: when \mathcal{A} issues a query for upgrading an existing decryption key $DK_w^{(k)}$, \mathcal{B} upgrades such a key to the last version n executing the primitive UpdateDK, using as input said key and $U_{DK}^{(i)}$, $\forall i \in [k, n]$. Finally \mathcal{B} sends to \mathcal{A} the updated decryption key $DK_w^{(n)}$.
- *ciphertext update*: when \mathcal{A} issues a query for upgrading an existing ciphertext $CP^{(k)}$, \mathcal{B} upgrades such a ciphertext to the latest version n executing the primitive UpdateCP, using as input said ciphertext and $(U_{DK}^{(i)})^{-1}$, $\forall i \in [k, n]$. Finally \mathcal{B} sends to \mathcal{A} the updated ciphertext $CP^{(n)}$.

Challenge \mathcal{A} submits two equal length messages m_0 and m_1 and a challenge policy \mathcal{P}^* to \mathcal{B} , which in turn forwards them to \mathcal{C} . \mathcal{C} responds with CP^* to \mathcal{B} , that will be exactly $CP^{*(0)}$ of Game 1. Then, \mathcal{B} upgrades such a ciphertext to the latest version n executing the primitive UpdateCP, using as input said ciphertext and $(U_{DK}^{(i)})^{-1}$, $\forall i \in [1, n]$. Finally \mathcal{B} sends to \mathcal{A} the updated challenge ciphertext $CP^{*(n)}$.

Phase 2 Phase 1 is repeated.

Guess \mathcal{A} outputs b' to \mathcal{B} , which forwards it to \mathcal{C} .

Since a correct guess in Game 1 is also a correct guess in Game 0, then the advantage of the adversary \mathcal{A} in Game 1 is equal to that of the adversary \mathcal{B} in Game 0. This concludes our proof. ■

We now consider a honest-but-curious cloud server colluding with a set of consumers. We state that a scheme is secure against an IND-CPA by a honest-but-curious cloud server colluding with a set of consumers if no PPT adversary \mathcal{A} has a non-negligible advantage against the challenger in the following game, denoted as Game 2. Game 2 is the same as Game 1 except that: (i) for every *encryption key update* query in Phase 1 and Phase 2 the adversary is given also the update quantities $U_{DK}^{(i)}$, $\forall i \in [1, n]$; and (ii) during Phase 1 and Phase 2 the adversary can issue the following new type of query.

- *generate decryption key's D field*: the challenger runs the primitive KeyGen using as input an attribute set provided by the adversary. Then, the challenger sends the field D of generated decryption key to the adversary.

Note that differently from the *generate decryption key* query, when issuing a *generate decryption key's D field* query the adversary is allowed to submit an attribute set that *satisfies* the challenge policy \mathcal{P}^* .

Theorem 2 *SEA-BREW is secure against an IND-CPA by a honest-but-curious cloud server colluding with a set of consumers (Game 2), under the generic bilinear group model.*

Proof We prove that if there is a PPT adversary \mathcal{A} that can win Game 2 with non-negligible advantage ϵ against SEA-BREW, then we can build a PPT simulator \mathcal{B} that can win Game 1 against SEA-BREW with the same advantage. We can modify the simulator \mathcal{B} used in the proof of Theorem 1 to prove this theorem. In the Phase 1 and Phase 2, \mathcal{B} additionally gives to \mathcal{A} the update quantities $U_{DK}^{(i)}, \forall i \in [1, n]$, which \mathcal{B} creates at each *encryption key update* query. During Phase 1 and Phase 2, when \mathcal{A} issues a *generate decryption key's D field* query, \mathcal{B} treats it in the same way of a *generate decryption key* query with an empty attribute set $\gamma = \{\emptyset\}$. Note indeed that a decryption key component D_{γ_j} is indistinguishable from a *complete* decryption key with *no attributes*. Hence, we can say that the advantage of \mathcal{B} in Game 0 is the same as that of \mathcal{A} in Game 2. ■

5 Performance Evaluation

5.1 WSN Traffic Overhead

In this section we analytically estimate the traffic overhead that the key revocation mechanism of SEA-BREW generates in the WSN, compared to the simple key revocation mechanism that can be realized on the Bethencourt et al.'s scheme. This mechanism works as follows. The producer leave procedure works in the same way as SEA-BREW: the WSN gateway simply broadcasts a signed message containing the producer identifier to all the WSN consumers, which remove the tuples associated to such an identifier from their locally maintained copy of the WSN Signature Table. The consumer leave procedure requires the WSN gateway to send a signed broadcast message containing the new encryption key to all the WSN producers, and in addition an encrypted and signed message containing a new decryption key to each WSN consumer. This procedure results in $\mathcal{O}(n)$ point-to-point messages where n is the number of WSN consumers. In contrast, SEA-BREW is able to perform both a consumer leave procedure by sending a single $\mathcal{O}(1)$ -sized signed broadcast message over the WSN. We refer to such Bethencourt et al.'s scheme with the simple key revocation mechanism as “BSW-KU” scheme (Bethencourt-Sahai-Waters with Key Update).

In both SEA-BREW and BSW-KU schemes, for implementing G_0 , G_1 , and the bilinear pairing we consider

	Size of broadcast message (bytes)	Number/size of unicast messages (bytes)	Total (bytes)
SEA-BREW			
consumer leave	252	-	252
producer leave	48	-	48
BSW-KU			
consumer leave	256	50×2,688	134,656
producer leave	48	-	48

Table 2 Traffic overhead of key revocation procedures in the WSN.

a supersingular elliptic curve with embedding degree $k = 2$ defined over a finite field of 512 bits. For the signatures of the unicast and broadcast messages we consider a 160-bit ECDSA scheme. Moreover, for the selective broadcast encryption used in the SEA-BREW scheme we consider the Boneh et al. scheme [5] with the same supersingular elliptic curve as above. This gives to both schemes an overall security level of 80 bits. We assume that, in both SEA-BREW and BSW-KU schemes, all elliptic-curve points are represented in compressed format [7] when they are sent over wireless links. This allows us to halve their size from 1024 bits to 512 bits. We further assume a low-bitrate WSN composed of one gateway, 50 consumers, and 50 producers. Each consumer is described by an attribute set of 20 attributes. We assume that the consumer identifiers and the producer identifiers are both 64-bit long.

Table 2 shows the traffic overhead of consumer leave and producer leave procedures of SEA-BREW and BSW-KU schemes. In SEA-BREW, the broadcast message sent by the WSN gateway during the consumer leave procedure is composed by the ECDSA signature (40 bytes), U_{EK} (64 bytes), and U_{DK} encrypted with the broadcast public key (148 bytes). Here we assumed that U_{DK} is encrypted by one-time pad with a key encrypted by the Boneh et al.'s broadcast encryption scheme [5], so it is composed of 20 bytes (the one-time-padded U_{DK}) plus the broadcast encryption overhead (128 bytes). As can be seen from the table, inside a low-bitrate WSN, SEA-BREW produces the same traffic overhead as the BSW-KU scheme when performing producer leave procedure. However, the overhead is merely the 0.2% of that produced by the BSW-KU scheme when performing a consumer leave procedure. Indeed, SEA-BREW is able to revoke or renew multiple decryption keys by sending a single 252-byte (considering 80-bit security) broadcast message over the WSN, opposed to the one 256-byte broadcast message plus 50 unicast messages of 2688-byte each (total: ~131KB of traffic) necessary to update a network with 50 con-

Primitive	Pairings	G_0 exp.'s	G_1 exp.'s
SEA-BREW			
Encrypt	-	$2 \mathcal{P} $	1
KeyGen	-	$2 \gamma + 1$	-
Decrypt	$2 \mathcal{P} + 1$	-	$ \mathcal{P} + 2$
UpdateCP	-	1	-
UpdateDK	-	1	-
BSW-KU			
Encrypt	-	$2 \mathcal{P} $	1
KeyGen	-	$2 \gamma + 1$	-
Decrypt	$2 \mathcal{P} + 1$	-	$ \mathcal{P} + 2$
UpdateCP	-	(not available)	-
UpdateDK	-	$2 \gamma + 1$	-
YWRL [35]			
Encrypt	-	$ \gamma $	1
KeyGen	-	$ \mathcal{P} $	-
Decrypt	$ \mathcal{P} $	-	$ \mathcal{P} $
UpdateCP	-	$ \gamma \cap \mathcal{A}_{rev} $	-
UpdateDK	-	$ \mathcal{P} \cap \mathcal{A}_{rev} $	-

Table 3 Comparison between SEA-BREW, BSW-KU, and YWRL schemes in terms of the computational cost of the primitives. For the YWRL scheme, the UpdateCP and the UpdateDK primitives correspond respectively to the AUpdateAtt4File and AUpdateSK of the original paper.

sumers (each of them described by 20 attributes) in a traditional CP-ABE scheme. With bigger WSANs (more than 50 consumers) or bigger attribute sets (more than 20 attributes) the advantage of SEA-BREW with respect to the BSW-KU scheme grows even more. Moreover, SEA-BREW also provides a re-encryption mechanism delegated to the untrusted cloud server, which is absent in the BSW-KU scheme.

5.2 Computational Overhead

In Table 3 we compare the computational cost of the primitives of SEA-BREW with those of BSW-KU, in terms of number and type of needed operations. In addition, we compare also with the scheme proposed by Yu et al. [35], denoted as “YWRL”, which is a KP-ABE scheme. Although CP-ABE and KP-ABE schemes are not comparable strictly speaking, we find interesting the comparison with this particular scheme because it natively supports a Lazy PRE mechanisms. In the table, the symbol \mathcal{A}_{rev} indicates the set of attributes that have been revoked, therefore the attributes that need to be updated in ciphertexts and decryption keys. The symbol $|\mathcal{P}|$ is the number of attributes inside the policy \mathcal{P} , and the same applies for $|\gamma|$. The expression $|\gamma \cap \mathcal{A}_{rev}|$ is the number of attributes belonging to both γ and \mathcal{A}_{rev} , and the same applies to $|\mathcal{P} \cap \mathcal{A}_{rev}|$. The operations taken into account are pairings, exponentiations in G_0 , and exponentiations in G_1 . In all the three schemes, we consider the worst-case scenario for the Decrypt primitive, which corresponds to a policy with an AND root

having all the attributes in γ as children. This represents the worst case since it forces the consumer to execute the DecryptNode sub-primitive on every node of the policy, thus maximizing the computational cost.

From the table we can see that SEA-BREW and BSW-KU pay the flexibility of the CP-ABE paradigm in terms of computational cost, especially concerning the Encrypt and Decrypt operations. However, this computational cost is the same of that in Bethencourt et al.’s scheme [4], which has proven to be supportable by mobile devices [2] and constrained IoT devices [13]. Note that our UpdateCP and UpdateDK primitives have a cost which is independent of the number of attributes in the revoked decryption key. The cost of such primitives depend instead on the number of revocations executed from the last update of the ciphertext or the decryption key.

Since modern cloud services typically follow a “pay-as-you-go” business model, in order to keep the operational costs low it is important to minimize the computation burden on the cloud server itself. We investigated by simulations the cloud server computation burden of our Lazy PRE scheme compared to the YWRL one, which represents the current state of the art. We can see from Table 3 that in both SEA-BREW and YWRL, the cloud performs only exponentiations in G_0 .

The reference parameters for our simulations are the following ones. We simulated a system of 100k ciphertexts stored on the cloud server, over an operation period of 1 year. We fixed an attribute universe of 200 attributes. We fixed a number of 15 attributes embedded in policies and attribute sets. We modeled the requests with a Poisson process with average of 50k daily requests. Finally, we modeled that several consumer leave procedures are executed at different instants, following a Poisson process with average period of 15 days. In order to obtain more meaningful statistical results we performed 100 independent repetitions of every simulation.

Fig. 5 shows the average number of exponentiations in G_0 performed by the cloud server, with respect to the number of attributes in ciphertexts and decryption keys, which is a measure of the complexity of the access control mechanism.

As we can see from the figure, SEA-BREW scales better than the YWRL as the access control complexity grows. This is because in the YWRL scheme every attribute has an singular and independent version number, and the revocation of a decryption key requires to update all the single attributes in the key. The cloud server re-encrypts a ciphertext with a number of operations equal to the attributes shared between the ciphertext and the revoked key. Such a number of operations

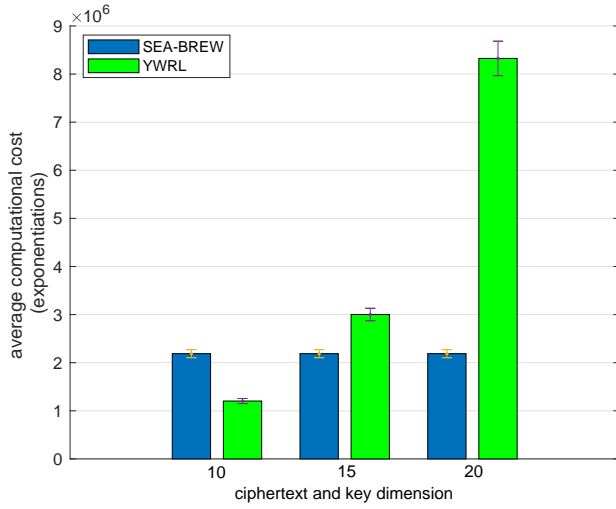


Fig. 5 Average number of exponentiations over a year, varying policies and attribute sets dimension. 95%-confidence intervals are displayed in error bars.

grows linearly with the average number of attributes in ciphertexts and decryption keys. On the other hand, in SEA-BREW the master key version number is unique for all the attributes, and the revocation of a decryption key requires to update only it. The cloud server re-encrypts a ciphertext with an operation whose complexity is independent of the number of attributes in the ciphertext and the revoked key.

Fig. 6 shows the average number of exponentiations in \mathbb{G}_0 performed by the cloud server with respect to the average daily requests, which is a measure of the system load. The number of attributes in ciphertexts and decryption keys is fixed to 15. As we can see from

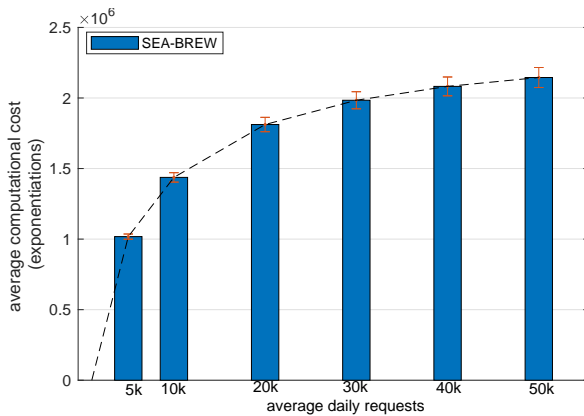


Fig. 6 Average number of exponentiation over a year, varying the average daily requests.

the figure the computational load on the cloud server

grows sub-linearly with respect to the increase of the requests. This behavior allows SEA-BREW to scale well also with high number of requests.

6 Conclusion

In this paper, we proposed SEA-BREW, a scheme suitable for IoT applications, which is able to securely revoke or renew a decryption key by sending a single $\mathcal{O}(1)$ -sized broadcast message over the wireless network. Unlike other ABE schemes in the literature, SEA-BREW allows for per-data access policies, following the Ciphertext-Policy Attribute-Based Encryption (CP-ABE) paradigm, which is generally considered flexible and usable. In our scheme, sensors, actuators, and users can exchange encrypted data via a cloud server, or directly if they belong to the same WSN. This makes the scheme suitable for best-effort communication, as well as for real-time communication. SEA-BREW also provides a mechanism for proxy re-encryption, by which old encrypted data can be re-encrypted by the cloud to make a revoked key unusable, without exposing data in clear to the cloud. We formally proved that SEA-BREW is IND-CPA secure also in case of an untrusted cloud server that colludes with a set of users, under the generic bilinear group model. We finally showed by simulations that the computational overhead is constant on the cloud server, with respect to the complexity of the access control policies.

Compliance with Ethical Standards

Funding The work presented herein has been partially funded by: (i) the European Processor Initiative (EPI) consortium, under grant agreement number 826646; (ii) the project PRA.2018.81 “Wearable sensor systems: personalized analysis and data security in healthcare” funded by the University of Pisa; and (iii) the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence).

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. Ambrosin, M., Anzanpour, A., Conti, M., Dargahi, T., Moosavi, S.R., Rahmani, A.M., Liljeberg, P.: On the feasibility of attribute-based encryption on Internet of Things devices. *IEEE Micro* **36**(6), 25–35 (2016)

2. Ambrosin, M., Conti, M., Dargahi, T.: On the feasibility of attribute-based encryption on smartphone devices. In: *Proceedings of the 2015 Workshop on IoT challenges in Mobile and Industrial Systems*, pp. 49–54. ACM (2015)
3. Atzori, L., Iera, A., Morabito, G.: The Internet of Things: A survey. *Computer networks* **54**(15), 2787–2805 (2010)
4. Bethencourt, J., Sahai, A., Waters, B.: Ciphertext-policy attribute-based encryption. In: *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pp. 321–334. IEEE (2007)
5. Boneh, D., Gentry, C., Waters, B.: Collusion resistant broadcast encryption with short ciphertexts and private keys. In: *Annual International Cryptology Conference*, pp. 258–275. Springer (2005)
6. Chen, F., Talanis, T., German, R., Dressler, F.: Real-time enabled IEEE 802.15.4 sensor networks in industrial automation. In: *Industrial Embedded Systems, 2009. SIES'09. IEEE International Symposium on*, pp. 136–139. IEEE (2009)
7. Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., Vercauteren, F.: *Handbook of elliptic and hyperelliptic curve cryptography*. Chapman and Hall/CRC (2005)
8. Cui, H., Deng, R.H., Li, Y., Qin, B.: Server-aided revocable attribute-based encryption. In: *European Symposium on Research in Computer Security*, pp. 570–587. Springer (2016)
9. Di Vimercati, S.D.C., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Over-encryption: management of access control evolution on outsourced data. In: *Proceedings of the 33rd international conference on Very large data bases*, pp. 123–134. VLDB endowment (2007)
10. Farrell, S.: Low-power wide area network (LPWAN) overview. RFC 8376 (2018)
11. Georgiou, O., Raza, U.: Low power wide area network analysis: Can lora scale? *IEEE Wireless Communications Letters* **6**(2), 162–165 (2017)
12. Gilchrist, A.: *Industry 4.0: the industrial Internet of Things*. Apress (2016)
13. Girgenti, B., Perazzo, P., Vallati, C., Righetti, F., Dini, G., Anastasi, G.: On the feasibility of attribute-based encryption on constrained IoT devices for smart systems. In: *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pp. 225–232 (2019). DOI 10.1109/SMARTCOMP.2019.00057
14. Goyal, V., Pandey, O., Sahai, A., Waters, B.: Attribute-based encryption for fine-grained access control of encrypted data. In: *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 89–98. Acm (2006)
15. Granjal, J., Monteiro, E., Silva, J.S.: Security for the Internet of Things: a survey of existing protocols and open research issues. *IEEE Communications Surveys & Tutorials* **17**(3), 1294–1312 (2015)
16. Hur, J.: Improving security and efficiency in attribute-based data sharing. *IEEE transactions on knowledge and data engineering* **25**(10), 2271–2282 (2013)
17. Jahid, S., Mittal, P., Borisov, N.: EASiER: Encryption-based access control in social networks with efficient revocation. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 411–415. ACM (2011)
18. La Manna, M., Perazzo, P., Rasori, M., Dini, G.: fABELous: An attribute-based scheme for industrial Internet of Things. In: *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pp. 33–38. IEEE (2019)
19. Latré, B., De Mil, P., Moerman, I., Van Dierdonck, N., Dhoedt, B., Demeester, P.: Maximum throughput and minimum delay in IEEE 802.15.4. In: *International Conference on Mobile Ad-Hoc and Sensor Networks*, pp. 866–876. Springer (2005)
20. Liu, Z., Cao, Z., Wong, D.S.: White-box traceable ciphertext-policy attribute-based encryption supporting any monotone access structures. *IEEE Transactions on Information Forensics and Security* **8**(1), 76–88 (2013)
21. Ming, Y., Fan, L., Jing-Li, H., Zhao-Li, W.: An efficient attribute based encryption scheme with revocation for outsourced data sharing control. In: *Instrumentation, Measurement, Computer, Communication and Control, 2011 First International Conference on*, pp. 516–520. IEEE (2011)
22. Montenegro, G., Kushalnagar, N., Hui, J., Culler, D.: Transmission of IPv6 packets over IEEE 802.15.4 networks. RFC 4944 (2007)
23. Picazo-Sanchez, P., Tapiador, J.E., Peris-Lopez, P., Suarez-Tangil, G.: Secure publish-subscribe protocols for heterogeneous medical wireless body area networks. *Sensors* **14**(12), 22619–22642 (2014)
24. Qin, B., Zhao, Q., Zheng, D., Cui, H.: Server-aided revocable attribute-based encryption resilient to decryption key exposure. In: *International Conference on Cryptology and Network Security*, pp. 504–514. Springer (2017)
25. Rasori, M., Perazzo, P., Dini, G.: ABE-Cities: An attribute-based encryption system for smart cities. In: *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*, pp. 65–72 (2018)
26. Reidy, K.M.: Complex cybersecurity vulnerabilities: Lessons learned from Spectre and Meltdown. <http://bit.ly/30prfJ8> (2018). Accessed: 2020-01-16
27. Sahai, A., Waters, B.: Fuzzy identity-based encryption. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 457–473. Springer (2005)
28. Sicari, S., Rizzardi, A., Grieco, L.A., Coen-Porisini, A.: Security, privacy and trust in Internet of Things: The road ahead. *Computer networks* **76**, 146–164 (2015)
29. Singh, M., Rajan, M., Shivraj, V., Balamuralidhar, P.: Secure MQTT for Internet of Things (IoT). In: *Communication Systems and Network Technologies (CSNT), 2015 Fifth International Conference on*, pp. 746–751. IEEE (2015)
30. Sowjanya, K., Dasgupta, M., Ray, S.: An elliptic curve cryptography based enhanced anonymous authentication protocol for wearable health monitoring systems. *International Journal of Information Security* **19**(1), 129–146 (2020)
31. Tosi, J., Taffoni, F., Santacatterina, M., Sannino, R., Formica, D.: Performance evaluation of Bluetooth Low Energy: a systematic review. *Sensors* **17**(12), 2898 (2017)
32. Touati, L., Challal, Y.: Batch-based CP-ABE with attribute revocation mechanism for the Internet of Things. In: *Computing, Networking and Communications (ICNC), 2015 International Conference on*, pp. 1044–1049. IEEE (2015)
33. Xu, Z., Martin, K.M.: Dynamic user revocation and key refreshing for attribute-based encryption in cloud storage. In: *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pp. 844–849. IEEE (2012)
34. Yu, S., Ren, K., Lou, W.: FDAC: Toward fine-grained distributed data access control in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems* **22**(4), 673–686 (2011)

35. Yu, S., Wang, C., Ren, K., Lou, W.: Achieving secure, scalable, and fine-grained data access control in cloud computing. In: Infocom, 2010 proceedings IEEE, pp. 1–9. Ieee (2010)
36. Yu, S., Wang, C., Ren, K., Lou, W.: Attribute based data sharing with attribute revocation. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, pp. 261–270. ACM (2010)
37. Zu, L., Liu, Z., Li, J.: New ciphertext-policy attribute-based encryption with efficient revocation. In: Computer and Information Technology (CIT), 2014 IEEE International Conference on, pp. 281–287. IEEE (2014)