APECS: A Distributed Access Control Framework for Pervasive Edge Computing Services

Sean Dougherty
Saint Louis University
St. Louis, U.S.
sean.dougherty@slu.edu

Reza Tourani
Saint Louis University
St. Louis, U.S.
reza.tourani@slu.edu

Gaurav Panwar
New Mexico State University
Las Cruces, U.S.
gpanwar@cs.nmsu.edu

Roopa Vishwanathan
New Mexico State University
Las Cruces, U.S.
roopav@nmsu.edu

Satyajayant Misra
New Mexico State University
Las Cruces, U.S.
misra@cs.nmsu.edu

Srikathyayani Srikanteswara
Intel Labs
Portland, U.S.
srikathyayani.srikanteswara@intel.com

ABSTRACT

Edge Computing is a new computing paradigm where applications operate at the network edge, providing low-latency services with augmented user and data privacy. A desirable goal for edge computing is pervasiveness, that is, enabling any capable and authorized entity at the edge to provide desired edge services–pervasive edge computing (PEC). However, efficient access control of users receiving services and edge servers handling user data, without sacrificing performance is a challenge. Current solutions, based on "always-on" authentication servers in the cloud, negate the latency benefits of services at the edge and also do not preserve user and data privacy. In this paper, we present *APECS*, an advanced access control framework for PEC, which allows legitimate users to utilize any available edge services without need for communication beyond the network edge. The *APECS* framework leverages multi-authority attribute-based encryption to create a federated authority, which delegates the authentication and authorization tasks to semi-trusted edge servers, thus eliminating the need for an "always-on" authentication server in the cloud. Additionally, *APECS* prevents access to encrypted content by unauthorized edge servers. We analyze and prove the security of *APECS* in the Universal Composability framework and provide experimental results on the GENI testbed to demonstrate the scalability and effectiveness of *APECS*.

CCS CONCEPTS

• **Security and privacy** → **Access control**; **Distributed systems security**; • **Networks** → *Security protocols.*

KEYWORDS

Distributed access control, authentication, authorization, attribute-based encryption, edge computing.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3484804

ACM Reference Format:
Sean Dougherty, Reza Tourani, Gaurav Panwar, Roopa Vishwanathan, Satyajayant Misra, and Srikathyayani Srikanteswara. 2021. APECS: A Distributed Access Control Framework for Pervasive Edge Computing Services. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3460120.3484804

1 INTRODUCTION

The number of wireless devices and connections are growing rapidly, the major drivers being the increasing number of smartphones and machine to machine communications from smart meters, autonomous vehicles, video cameras, and more [5]. As an example, real-time video analytics data from Internet of Things (IoT) devices such as surveillance cameras [4], estimated to be over 1 billion by the end of 2021[1], supports several practical, useful applications such as traffic control, autonomous driving, providing cognitive assistance to users [25, 31], and more. The video feed data generated by cameras needs to be processed quickly and in proximity to the user, which precludes transferring the data to the Cloud for processing. This need is particularly accentuated for latency-sensitive applications such as autonomous driving.

To address this challenge, various edge computing ecosystems have been proposed, including *cloudlets*, fog computing, and Multi-Access Edge Computing [21] with the premise of deploying powerful servers and gateways to serve users in regions with high computation demand. Recently, the notion of Pervasive Edge Computing (PEC) [23] has emerged, aiming to create an ecosystem in which the computation capability of every peer device at the edge, *e.g.,* smartphones, tablets, and vehicles, can be brought to bear to serve users' computation needs.

Motivation: Current access control enforcement solutions designed for cloud computing cannot be trivially ported to the distributed, multi-stakeholder PEC environment. In a PEC ecosystem, relying on an always-online cloud service for access control is undesirable for several reasons, such as high latency due to several rounds of direct client-server communication, the Cloud might become a single point of failure, and cloud server(s) going rogue and undermining user privacy and/or user data confidentiality. Furthermore, in the

[1] https://technology.informa.com/607069/video-surveillance-installed-base-report-2019

highly dynamic PEC environment with quick nodes turnover, lack of mutual authentication between the user and edge servers, and delayed revocation of rogue servers (particularly with high turnover rates) is an added challenge. This necessitates re-envisioning access control mechanisms without involving the Cloud.

Use Cases: We consider two application use cases to motivate the need for distributed and fine-grained access control for secure execution of edge services. The extensive deployment of security, traffic, and dash cameras motivated crowdsensing applications, including a vehicle-tracking AMBER Alert system [30, 31] and a parking spot locating service for dense urban areas [10]. These applications collect user generated video feed for low latency processing by the authorized edge servers, available in the data sources locale to identify occupied parking spots or vehicles using their license plates. Such data sharing applications, however, raise privacy and security concerns over how user generated data is collected, processed, and utilized. For instance, in the AMBER Alert example, parents may accept sharing their children's information with the authorities but not the larger public community, or a driver in the region of interest may be willing to share the video feed of her on-dash camera only with the police department. In another scenario, the police department might require the location (or the annotated image) of the alleged kidnapper's vehicle to only be shared with active duty officers in order to mitigate the risk of vigilantes.

The second use case is post disaster rescue, in which first responders and civilian volunteers spontaneously form rescue teams to collect information such as the video feed from body cameras and updates from cameras/sensors on disaster victims devices. The data will be opportunistically shared with the available and authorized edge servers (*e.g.,* vehicles, drones, or base stations) for processing and critical decision-making–often in this case there is no centralized cloud available. In this use case, only relevant information should be shared with each participant. For instance, a civilian volunteer should not be able to access the private health information of a victim while a paramedic at the same site should be able to obtain such information. These use cases elucidate the demand for distributed and fine-grained access control, enabling users and the dynamic edge infrastructure to mutually authenticate and authorize each other without relying on an always online authentication server, which will often be difficult to provision.

Unique Constraints in the PEC Environment: The PEC ecosystem is highly dynamic and is composed of mobile devices, *e.g.,* cars, smartphones, and PEC servers with high turnover. Providing services in such a fast-changing, evolving environment is a challenge.

This will be further compounded by the low-latency and high bandwidth requirements of the next generation services, *e.g.,* autonomous driving and industrial IoT, where significant amounts of data need to be transferred to a server quickly, processed rapidly, and delivered back to a customer, sometimes in mere milliseconds. This necessitates the need for quick authentication and rapid interchanges between the consumer and the servers before the connection is lost–potentially forever. Further, the personal nature of the user data, such as images, puts stringent privacy requirements on it. Providing personal data to an unauthorized or unauthenticated server for service becomes a high stakes operation and the impact of data falling in the wrong hands (especially if authentication is not rigorous) could be significant. The high dynamicity may not adequately equip the consumers to verify the servers' access rights and authenticity in the short time window available for interaction. These constraints are major motivating challenges.

Overview: We address these motivating challenges by proposing *APECS*–a distributed, multi-authority access control framework for dynamic PEC ecosystems. *APECS* enables the users and PEC servers to mutually authenticate and authorize each other via a federated access control model without relying on a centralized root of trust. Utilizing multiple trust authorities regulates access to users' personal data and prevents a malicious authority from breaching users' privacy by unilaterally accessing the user's personal data. To ensure that users can provide access right proofs and this access can be verified at the PEC server directly, *APECS* employs a token-based authorization scheme (similar to OAuth), which includes a novel authentication method for verifying token ownership–a feature not provided in OAuth. In addition, *APECS* removes the "always-on" authentication server requirement and allows asynchronous authentication of PEC servers. Thus, in the highly dynamic and intermittent edge environment, *APECS* allows the users to securely submit their data for processing, without edge server discovery and with implicit authenticity verification; ensuring that only available and authorized PEC servers can decipher the data for processing.

Contributions: In summary, the contributions of this paper include: **a)** Design of *APECS*, a distributed mutual access control enforcement framework that operates at the edge after bootstrapping by the Cloud. *APECS* uses multi-authority attribute based encryption [3], is agnostic of the underlying network architecture, thus is portable to future internet architectures. **b)** Design of *APECS PKC* (public key cryptosystem), an alternative *APECS* implementation using traditional public key cryptosystems. *APECS PKC* is suitable for static networks, in which PEC servers' availability is known to the users prior to service requests, allowing the users to establish secure channels to the desired PEC server. **c)** *APECS* has an efficient and quick revocation mechanism for edge entities that does not need immediate communication with the Cloud, and requires minimal (not system-wide) re-keying of the remaining entities and data re-encryption in the system. **d)** Rigorous security analysis of *APECS* using the Universal Composability framework and discussion of enhancements using traditional public key cryptosystems. **e)** *APECS* prototype implementation in the GENI testbed [1] and performance evaluation with the existing set-up and an IP-based potential design set-up.

The paper is organized as follows. We discuss the related work in Section 2. Section 3 includes our models and assumptions. Section 4 presents *APECS* building blocks and overview with detailed design in Section 5. Section 6 includes the security analysis of *APECS*. We discuss the reference implementation of *APECS* along with its evaluation in Section 7 and draw our conclusion in Section 8.

2 RELATED WORK

The majority of access control for services today happens far from the edge, either on the Cloud or the Content Provider premises. Recently, a few initiatives have proposed edge computing as a platform for providing security services at the edge [6, 12, 18]. However, access control enforcement at the network's edge has received little attention. Despite some similarities, edge-centric access control

enforcement requires additional considerations due to its decentralized, dynamic, and multi-stakeholder nature, which limits the effectiveness of conventional cloud-based solutions. We review the access control literature in both Cloud and edge ecosystems.

In the literature, attribute-based access control (ABAC), in which policies are used for granting access rights, has been widely studied [7, 8, 12, 19, 20, 26, 27, 33]. ABAC schemes can generically be realized by attribute-based encryption (ABE) [8, 20, 26], which have proven to be very beneficial in cloud architectures [26] for providing flexible and fine-grained data sharing frameworks. However, the use of ABAC for user authentication and authorization can be costly compared to capability based access control [7, 8, 12–14, 19, 20, 27, 33]. To alleviate this challenge, approaches have been proposed for reducing ABE's computation cost [7, 12, 33]. Xue *et al.* utilized hash-chains to limit the number of ABE operations in an information-centric setting [22, 27, 28]. The user's initial communication is authenticated using a single ABE operation while the follow-up requests are authorized by a series of bootstrapped chained hashes. Xue *et al.* proposed proof-of-attribute challenges to prevent Economic Denials of Service (EDoS) [26], in which the receiver proves the possession of the attributes by solving a challenge encrypted with those attributes prior to communication.

Capability-based access control (CapBAC) is an alternative access control model to ABAC systems. In CapBAC, unforgeable access tokens are given to subjects to represent subjects' access rights–enabling a more distributed and computationally cheap authentication and authorization [11, 24, 33]. In the majority of the token-based CapBAC implementations, such as OAuth [11] and Heracles [33], the cloud back-end is assumed to be the sole trusted authority, responsible for token generation and distribution. In OAuth, the client in the possession of the access token (referred to as "bearer token") can authorize themselves to the resource server by including the access token in the requests. However, the OAuth's bearer token does not include user specific information, allowing an intercepted token to be used by the attacker. Thus, undermining its effectiveness. Heracles [33] extended OAuth via a hybrid solution, in which both bulk operations and single target operations were accomplished via ID-based tokens and attribute based tokens, respectively. Despite Heracles' capability in promoting fine-grained access control, it remains vulnerable if an unfaithful subject shares its token. To remedy, FairAccess [19] proposed the utilization of a shared private blockchain at the edge to provide the accurate ledger of access tokens, their rights, and their possession.

Despite some initiatives for IoT device access control at the edge, no effort was able to build a holistic mutual access control system for computation offloading to the edge. We addressed this gap by building a distributed framework for mutual authentication and authorization of users and PEC servers. *APECS* provides a scalable and efficient access control enforcement in highly dynamic PEC environments, and enables federation with access control authorities for quick access revocation without system-wide re-keying.

3 MODELS AND ASSUMPTIONS

3.1 System Model

Our system model comprises the computing environment, service consumers, and service providers. The computing environment

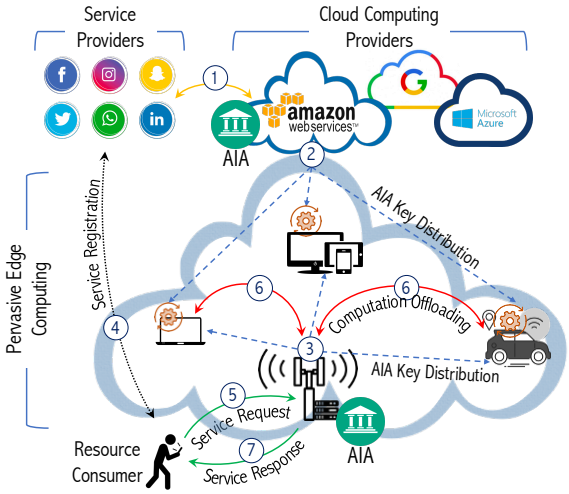


Figure 1: *APECS* system model including the parties involved in secure delivery of edge services.

includes Cloud providers and the PEC ecosystem [23]. The PEC ecosystem is itself composed of pre-deployed components such as the multi-access edge computing (MEC) infrastructure [21] and the users' devices that are joining the computing resource pool for executing requested services. In the rest of the paper, we refer to these users' devices as PEC servers. A service can either be static, *e.g.,* static videos or web content, or dynamic, *e.g.,* annotation of videos or images. Dynamic services may require an input data from the user (*e.g.,* a user's image/video for performing annotation) or other service providers (*e.g.,* the police department requesting the vehicle-tracking AMBER Alert information from other vehicles).

A service provider owns the requested static/dynamic service. We assume that the service providers are well-known. A service consumer is typically a user who requests a service. Given the fluid and highly dynamic nature of PEC, a user can have multiple roles at the same or different times, simultaneously acting as a service provider and consumer. In *APECS*, we employ multi-authority attribute-based encryption (MABE) [3] with service providers designated as one set of attribute-issuing authorities (AIAs), and base stations (part of internet service providers) being the second set of AIAs. Alternatively, MEC servers can act as base stations for the second AIA category.

System Entities Interactions. As shown in Step (1) of Figure 1, each service provider initiates its AIA, hosted as a virtual machine on the Cloud. The AIAs onboard PEC servers and provide them with attributes and secret keys for their registered services (Step (2)). Similarly, each base station initiates its AIA to onboard its local PEC servers (Step (3))[2]. At this stage, the PEC nodes are fully onboarded by both AIAs. A user, interested in a service, registers with the service provider and obtains an authentication/authorization token (Step (4)). To request a service, the user encrypts her data using the expected attributes of the service provider and her local base station, and sends the encrypted data (and her token) into the network via the base station (Step (5)). The base station relays the user's request to the existing PEC servers (Step (6)) for enforcing access control

[2] An ISP may run a system-wide AIA rather than one per base station; but that is an implementation decision, which we do not discuss.

and service execution. The PEC servers return the result of the service to the base station, which forwards it to the user (Step (7)).

3.2 Security and Computational Assumptions

We assume that all entities are capable of performing symmetric and asymmetric key cryptography, and have their clocks loosely synchronized. We assume the existence of a trusted public key infrastructure (PKI) by which all entities obtain certificates corresponding to their cryptographic key pairs from well-known authorities (*e.g.,* Verisign). For instance, a provider p obtains its certificate $Cert_p$ and a user u obtains their certificate $Cert_u$. We further assume that symmetric and asymmetric key operations and cryptosystems are secure. We assume the cloud providers and base stations are honest but curious participants in that they do not deviate from the protocols but try to learn information about the system. It is a common assumption when considering these entities as part of the infrastructure. We further assume attackers are Probabilistic Polynomial-time (PPT) adversaries and are computationally bounded.

Our scheme relies on assumptions based on the decisional Diffie Hellman (DDH) problem, the decisional Bilinear Diffie Hellman (BDH) problem, the k-decisional Diffie Hellman (k-DDH) problem, and the external Diffie Hellman (XDH) problem [3]. Please refer to Appendix 10 for formal definitions of these assumptions.

3.3 Threat Model

We consider the following six threats from the service consumers, the computing ecosystem (including PEC servers and the cloud computing providers), and malicious third parties, which may play the role of an edge server or a service consumer. An outsider may try to unlawfully use a service *(a)* without registering and obtaining a valid token for the service, or *(b)* by using a forged token (not generated by the service provider or containing invalid information). A legitimate user (service consumer) may try to *(c)* request a service with an expired token or a token with insufficient authorization level (similarly reusing a token from one service provider to give access to services of other providers), or *(d)* share their token with an unauthorized user to allow unauthorized service access.

An unauthorized PEC server may try *(e)* to mount a spoofing attack by impersonating an authorized server to hijack or obtain a user's data. An authorized but malicious PEC server may try to *(f)* collude with an unauthorized user to maliciously provide a service. This includes offering a static service (*i.e.,* content) or the execution of a dynamic service to an unauthorized user. We do not consider the situation where a malicious PEC server returns incorrect results, possibly for avoiding resource intensive computation. For addressing this, techniques for verifiable computing [32] can be used in conjunction with *APECS*.

4 *APECS* BLOCKS AND OVERVIEW

In this section, we give an overview of *APECS* and its building blocks. Table 1 presents the notations used in explaining *APECS*.

4.1 User Authentication and Authorization

In *APECS*, a user $u \in \mathbb{U}$ interested in using a service provider's $p \in \mathbb{P}$ services (*e.g.,* Instagram) has to register herself with p to obtain a customized and time-bounded token. The token allows u to authenticate herself to the corresponding PEC server $e \in \mathbb{E}$ providing service for p, when requesting the service, which can be either static or dynamic. We note that p has to sign all the issued tokens for integrity verification. Below, we elaborate on the token's structure, its components, and the rationale behind its components.

Table 1: Notations Used

Notation	Description
$\mathbb{P}$	Set of service providers.
$\mathbb{U}$	Set of clients.
$\mathbb{C}$	Access Control Cloud.
$\mathbb{E}$	Set of PEC servers.
$\mathbb{B}$	Set of base stations.
$\mathcal{T}_{pu}$	User u's token $\mathcal{T}$ from service provider p.
ID_x	Identifier of entity/service x.
$Cert_x$	Entity x certificate containing verification key VK_x.
L_x	Authorization level of entity/data x.
T_{exp}	Token's expiry time.
T_c	Current time.
M_{pk}	Public key of MABE system for ABE operations.
$[A_e]$	List of MABE decryption keys possessed by e.
$\mathcal{R}_{AC}$	Service provider p's registration request to $\mathbb{C}$.
SK_x	Signing key of entity x.
VK_x	Signature verification key of entity x.
σ_x	Signature on data x.
$revocTable$	List of revoked users' tokens stored at each $e \in \mathbb{E}$.
$serverTable$	List of PEC servers maintained by each AIA.
$userTable$	List of users and tokens maintained by each $p \in \mathbb{P}$.

Definition 4.1. **Authentication Token**
Token $\mathcal{T}_{pu}$ represents the unique JSON Web Token (JWT) that service provider p generates for user u. The JWT format provides greater functionality than traditional bearer tokens, such as those used in OAuth. $\mathcal{T}_{pu}$ is a unique token that includes the service provider's unique identifier, ID_p, the service identifier, ID_s, (or a list of service identifiers), the user's certificate, $Cert_u$, the user's authorization level, L_u, (or a list of authorization levels), and its expiry time, T_{exp}: $\mathcal{T}_{pu} := \langle ID_p, [ID_s], Cert_u, [L_u], T_{exp} \rangle$.

The service provider's identifier, ID_p, in $\mathcal{T}_{pu}$ enables the access control enforcers, *i.e.,* PEC servers, to fetch p's certificate for token signature verification, thus ensuring token's integrity and provenance. *We note that lack of token integrity and provenance verification is one of the major shortcomings of OAuth, which we address.* The service identifier, ID_s, indicates the name of service(s) that u is authorized to use. By including ID_s in $\mathcal{T}_{pu}$, the PEC servers prevent u's unauthorized access to other services p provides that requires independent membership per service. For each service (static or dynamic), L_u indicates u's authorization level, *i.e.,* Bronze, Silver, or Gold, to be matched against the required authorization level of the requested service. Token $\mathcal{T}_{pu}$ also includes u's certificate, $Cert_u$, which enables the PEC servers to verify the authenticity of u's signed request, thus preventing unauthorized users from using a hijacked token. Finally, $\mathcal{T}_{pu}$ includes an expiry time as a system parameter. At the conclusion of T_{exp}, u can request to renew her token, which is granted at the service provider's discretion.

4.2 Asynchronous Server Authentication

APECS is designed for a dynamic edge computing ecosystem where edge servers can leave and join at will. In such ecosystems, the traditional authentication mechanisms, in which the user has to first

discover the available PEC servers, create a secure connection, and authenticate the selected server would not scale. Thus, *APECS* devises an asynchronous PEC server authentication framework using the MABE scheme proposed in [3]. In *APECS*' asynchronous PEC server authentication framework, users encrypt their data (needed for service execution) using the MABE scheme, allowing any PEC server with the requisite set of attributes obtained from multiple attribute-issuing authorities to decrypt the data and execute the requested service without the need for server discovery, secure channel establishment, or synchronous interactions between the user and PEC servers. To obtain pertinent credentials (*e.g.,* secret keys and attributes), PEC servers should be associated with the corresponding service providers and a base station. Before presenting the MABE scheme [3], we note that broadcast encryption (BE) is another relevant technique for asynchronous authentication [16, 17]. Despite its simplicity, BE is not suitable for the PEC ecosystem since it does not work well for several one-to-one (consumer-edge server) communications [9], and falls short in performing dynamic revocation. Furthermore, a federated BE approach does not exist in the literature.

Definition 4.2. **Multi-authority Attribute-Based Encryption (MABE) [3]**
A key policy ABE scheme (KPABE) with n attribute-issuing authorities (AIAs) consists of the following four algorithms. All algorithms except decryption are randomized.
1) $(\text{sysparam}, (apk_1, ask_1), \ldots, (apk_n, ask_n)) \leftarrow \text{ABE.Setup}(1^\lambda, n)$*: This algorithm runs once in the beginning to setup the system parameters and the AIAs. It takes in a security parameter, λ, and number of AIAs n, as input, and outputs the system parameters,* sysparam*, and each AIA's public/private key pairs. The* sysparam *includes bilinear group information, and the threshold value d_k that denotes the minimum number of attributes each user needs to possess from an AIA $k; k \in [1 \ldots n]$. Set public key, $M_{pk} = (\text{sysparam}, apk_1 \ldots apk_n)$.*
2) $\mathbb{SK}_k \leftarrow \text{ABE.KeyGen}(M_{pk}, ask_k, id, \mathbb{A}_k)$*: This algorithm is run by AIA k, and takes as input M_{pk}, k's secret key, ask_k, a userid, id, and a set of attributes, $\mathbb{A}_k$, s.t., $|\mathbb{A}_k| \geq d_k$. It outputs the user's secret keys $\mathbb{SK}_k$.*
3) $\mathsf{C} \leftarrow \text{ABE.Encrypt}(M_{pk}, (\mathbb{A}_1, \ldots, \mathbb{A}_n), m)$*: This algorithm takes in M_{pk}, a subset of at least d_k attributes from an AIA $k; k \in [1 \ldots n]$, message m, and outputs ciphertext* C*.*
4) $\{m, \perp\} \leftarrow \text{ABE.Decrypt}(M_{pk}, (\mathbb{SK}_1, \ldots, \mathbb{SK}_n), \mathsf{C})$*: This algorithm takes in the public key M_{pk} and a set of secret keys from each AIA sufficient to decrypt the ciphertext* C*. If successful it outputs the plaintext message m, else outputs $\perp$. Decryption is successful whenever the overlap between the set of secret keys and the set of attributes associated with the ciphertext is above a threshold.*

4.3 *APECS* Overview

APECS consists of seven protocols that describe the interactions between the cloud provider, $\mathbb{C}$, service providers, $\mathbb{P}$, PEC servers, $\mathbb{E}$, and users, $\mathbb{U}$. *APECS* consists of two phases: *(i)* distributed user authentication and *(ii)* asynchronous server authentication and service execution.

In the first phase, PEC servers authenticate and authorize users' requests by validating the users' requests and corresponding tokens. For token verification, PEC servers use service providers' identifiers (included in the tokens) to fetch the corresponding certificates and validate tokens' signatures; preventing token forgery. For users' requests verification, PEC servers use the users' certificates included in the corresponding tokens to verify requests' signatures. Finally, the PEC servers use the other components of the tokens to authorize users for requested services. Token-based user authentication and authorization in *APECS* enables mobile users to utilize edge services while moving across base stations without the need for obtaining new tokens or updating cryptographic materials.

In the second phase, upon successful user authentication, PEC servers should fulfill service requests on a service provider's behalf. In order to protect the user's privacy, the user encrypts the data needed for her service execution using the set of attributes (from both AIAs) pertinent to the requested service. Following the MABE scheme mentioned in Definition 4.2, PEC servers use their attribute sets for data decryption. A successful MABE decryption process proves the authenticity of the PEC server for service execution.

APECS enables efficient revocation of users and PEC servers. For user revocation, service providers share their user revocation lists (revoked tokens) with the PEC servers. For PEC server revocation, instead of a system-wide re-keying of un-revoked users, the provider notifies the base station that is associated with the PEC server to revoke it. This localizes the PEC server revocation to the base station, which invariably has a much smaller number of connected PEC servers.

5 *APECS* DESIGN

This section includes *APECS* architectural design and details of protocols for system setup and registration, users service requests, PEC servers' service response (including mutual authentication), and user and PEC server revocation. We also discuss *APECS PKC*–an *APECS* construct using the traditional public key cryptosystem for scenarios where users and PEC servers can synchronously interact.

5.1 System Setup and Registration

5.1.1 Bootstrapping of AIAs and Provider Registration (Protocol 1). In *APECS*, service providers use the Cloud as a conduit for their interactions with the PEC ecosystem due to the Cloud's centrality. Thus, to delegate access control enforcement to PEC servers, the service provider ($p \in \mathbb{P}$) must register with the Cloud ($\mathbb{C}$) as the hosting environment for running its AIA and bootstrapping the PEC servers at the edge. Initially, as illustrated in Protocol 1, the providers and base stations run the system setup for the MABE protocol as defined in Definition 4.2 (Line 1). Provider registration begins with p forming and sending a request ($\mathcal{R}_{AC}$) to $\mathbb{C}$, containing p's certificate ($Cert_p$), followed by a challenge-response communication to prove the ownership of $Cert_p$ (Line 2). We note that $Cert_x$ contains the verification key of entity x (VK_x), which will be used for signature verification. Upon receipt of $\mathcal{R}_{AC}$ from p, $\mathbb{C}$ registers p by generating a profile and a provider identifier ID_p (Line 3), using either the unique subject identifier value stored in $Cert_p$ or its digest, and returns it to p (Line 4). This allows p to use ID_p when generating future access tokens and aids in confirming the validity of the tokens at the PEC servers.

5.1.2 Edge Server Registration (Protocol 2). As shown in Protocol 2, a PEC server ($e \in \mathbb{E}$) initiates its registration process by securely

Protocol 1 System Setup and Provider Registration

{**At AIAs (Provider & Base station)**}

1: $\left(M_{pk} = (\text{sysparam}, apk_1, \ldots, apk_n), ask_1, \ldots, ask_n\right) \leftarrow$ ABE.Setup($1^\lambda, n$)

{**At Provider**}

2: send $\mathcal{R}_{AC} = \{Cert_p\}$ to $\mathbb{C}$

{**At Cloud**}

3: $ID_p \leftarrow$ registerProvider($Cert_p$);
4: return ID_p to p;

sending it's certificate, ($Cert_e$), and the list of identifiers, ($[ID_s]$), of services it would like to provide to two AIAs–both p's AIA, hosted on the Cloud, and the base station that e is connected to (Line 1). Each AIA executes the MABE key generation algorithm (following Definition 4.2) to generate a list of secret keys $[A_e]$ for e, corresponding to the services $[ID_s]$ offered by e (Line 3). The AIA stores the generated keys in a *serverTable* and securely sends them to e (Lines 4-5), allowing it to decrypt any requests that it may receive from users, so long as e's attributes match the attributes in the user's request.

5.1.3 User Registration (Protocol 3). A new service consumer ($u \in \mathbb{U}$), interested in p's service(s), has to register with p to obtain a customized token for future service utilization. As shown in Protocol 3, user registration begins with u sending a request to p containing her certificate ($Cert_u$) and general user data (*user_data*) that are often used for creating user accounts, such as credentials, e-mail, and birth-date (Line 1). The *user_data* also contains any number of values necessary for the provider to complete registration of the user. Additionally, this metadata contains information related to the requested service, such as service tiers (*e.g.,* bronze, silver, and gold) or service types (*e.g.,* image annotation, gaming, or streaming).

Provider p verifies u's request and if the registration request is valid (Line 2), it retrieves a list of service identifiers ($[ID_s]$) corresponding to the *user_data*. It should be noted that the $[ID_s]$ corresponds to services offered by p. With the user's information, p generates a customized access token ($\mathcal{T}_{pu}$) for u containing p's identifier (ID_p), list of permitted service identifiers ($[ID_s]$), the user's certificate ($Cert_u$), u's authorization levels for the permitted services ($[L_u]$), and an expiry time T_{exp} (Line 3). To protect the token's integrity and for provenance, p signs the token (Line 4). The provider further obtains M_{pk}, the MABE's master public key (Line 5), which is needed in the MABE encryption process. Upon storing u's information and her token in the *userTable*, p securely sends a tuple, including $\mathcal{T}_{pu}$, its signature ($\sigma_{\mathcal{T}_{pu}}$), and M_{pk} to u (Lines 6-7). However, if p doesn't accept u's registration request, it returns a negative acknowledgement to u (Lines 8-9). The presence

Protocol 2 Edge Server Registration

{**At Edge Server**}

1: send $\{[ID_s], Cert_e\}$ to AIAs (provider hosted on $\mathbb{C}$ and the corresponding base station)

{**At Provider & Base station**}

2: $[A_e] \leftarrow$ ABE.KeyGen($M_{pk}, ask_e, id_e, [ID_s]$)
3: store $\{e, [A_e]\}$ in *serverTable*
4: return $\{e, [A_e]\}$ to e

Protocol 3 User Registration

{**At User**}

1: send $\{user\_data, Cert_u\}$ to p

{**At Provider**}

2: **if** p accepts u's registration request **then**
3: Set $\mathcal{T}_{pu} = (ID_p, [ID_s], Cert_u, [L_u], T_{exp})$
4: $\sigma_{\mathcal{T}_{pu}} \leftarrow \text{Sign}_{SK_P}(\mathcal{T}_{pu})$
5: $M_{pk} \leftarrow$ retrieveABECredentials()
6: store $\{u, \mathcal{T}_{pu}\}$ in *userTable*
7: return $\{\mathcal{T}_{pu}, \sigma_{\mathcal{T}_{pu}}, M_{pk}\}$ to u
8: **else**
9: return $\perp$
10: **end if**

of ID_p and the $\sigma_{\mathcal{T}_{pu}}$ helps to ensure authenticity and integrity of u's token and prevents tampering with the token. Also, $Cert_u$ in the token indicates u's ownership of the token. Additionally, the set of permitted service identifiers, $[ID_s]$, is included to enable easy vetting of requests by PEC servers. Using tokens in *APECS*, the PEC server is not required to contact p to verify u's authorization. Finally, T_{exp} proves token's freshness and enables a lazy revocation process through token expiration.

5.2 Service Request Protocol

Protocol 4 details u's service request procedure. We note that requesting a content is an instance of requesting a static service which does not require user specific input data. In contrast, offloading a computation is an instance of a user requesting a dynamic service, which may require some input data (from the user to perform computation on). These two types of services are different in the sense that the requesting user's data should be protected when the user is requesting a dynamic service, hence the need for data encryption with MABE. Initially, u has to specify if the desired service (ID_s) is dynamic (*e.g.,* image annotation) or static (*e.g.,* video streaming). For a static service, u creates two of the request's components by specifying the *content_name* as C_1 and a null $C2$ (Lines 1-2). We note that the *content_name* is not required to be encrypted. However, MABE encryption can be used for encrypting the *content_name* to preserve u's privacy at the cost of additional latency. For a dynamic service, u first encrypts symmetric key K using the MABE scheme with the master public key M_{pk} and the requisite service attributes represented by ID_s to generate C_1 (Line 4). She then uses a symmetric key cryptosystem such as AES and k to encrypt the required service data (D) for generating C_2 (Line 5). The publicly visible attributes in the MABE policy for C_1 identify the specific service requested by the user to all PEC servers who receive the request. Subsequently, u creates her request (Req) as a four-element

Protocol 4 User's Service Request

1: **if** ID_s is Static Service Request **then**
2: $C_1 \leftarrow content\_name, C_2 \leftarrow \perp$
3: **else if** ID_s is Dynamic Service Request **then**
4: $C_1 \leftarrow$ ABE.Encrypt(M_{pk}, ID_s, K)
5: $C_2 = \text{Enc}_K(D)$
6: **end if**
7: set Req = $\{\mathcal{T}_{pu}, ID_s, C_1, C_2\}$, $\sigma_{Req} \leftarrow \text{Sign}_{SK_u}(Req)$
8: send $\{Req, \sigma_{Req}\}$

tuple, including the token $\mathcal{T}_{pu}$, the requested service with identifier ID_s, C_1, and C_2.

The user then signs the crafted request using SK_u that corresponds to her certificate ($Cert_u$), embedded in $\mathcal{T}_{pu}$ (Line 7), and sends the request and its signature as a payload to the base station to be forwarded to the edge network (Line 8).

5.3 Service Response Protocol

As detailed in Protocol 5, edge server e receives the request from the base station (the base station just serves as a relay) and verifies the signature on the request using VK_u extracted from $\mathcal{T}_{pu}$ (Lines 1-2). If signature verification fails, e returns error and drops the connection (Lines 33-35). Successful verification indicates that the request is generated by $\mathcal{T}_{pu}$'s owner. If successful, e confirms the freshness of $\mathcal{T}_{pu}$ by comparing its expiry time (T_{exp}) with the current time (T_c) (Lines 4-5). Subsequently, e searches for $\mathcal{T}_{pu}$ in its revocation table (*revocTable*) to ensure that u has not been added to access-denied list (Lines 6-8). If any checks fail, e returns an error and drops the connection. In the event that a valid user has encountered any of these failures, the user may request a new token and obtain a valid $\mathcal{T}_{pu}$ using her new certificate and established user credentials (refer to Protocol 1). Upon token validation, e uses ID_p that is contained within $\mathcal{T}_{pu}$ to lookup $Cert_p$ in its local cache. If e does not have the $Cert_p$ corresponding to ID_p, it can obtain (from the user or the service provider) $Cert_p$ using ID_p (Line 9). On receiving $Cert_p$, e extracts the VK_p from $Cert_p$ and verifies the signature on $\mathcal{T}_{pu}$, dropping the connection if invalid (Lines 10-11). Successful verification validates $\mathcal{T}_{pu}$'s integrity and provenance. Finally, e compares u's authorization tier (L_u) with the service tier of the requested data (L_D), contained in the data (Lines 13-15). If authorization fails, e drops the connection; otherwise, it moves to the next step.

Once e has successfully authenticated and authorized u and can process the request. Request fulfillment begins with the PEC server identifying the requested service type based on ID_s (Line-17). For a dynamic service request, e verifies whether it offers the requested service or not (Line 18). If e is capable, it decrypts C_1 using its decryption keys $[A_e]$ and retrieves the symmetric key K, using it to decrypt the data (D) needed for service execution (Lines 19-21). In the event that the server does not possess the capability to execute the service, it forwards the service request to another server (Lines 22-23). For a static service request, e looks up the *content_name* (in C_1) in its cache and returns the data to the user (Line 26-27). If the data is not available in e's cache, it forwards the request to another server or p as defined by the application logic (Lines 28-29). Using MABE in *APECS* helps with asynchronous authentication/authorization of the PEC server–a PEC server without sufficient attributes cannot access user's data. Moreover, MABE enables efficient PEC server revocation without global system re-key; discussed in the following.

5.4 User and PEC server Revocation Protocols

Access right revocation is a commonplace functionality of any access control system. In *APECS*, we consider both user access revocation as well as PEC server revocation. Of particular importance is PEC server revocation, necessary to prevent revoked server's access to users' data. We start with revocation of user u and then

Protocol 5 Edge Server's Service Response

1: receive $\{Req, \sigma_{Req}\}$
2: extract $\{\mathcal{T}_{pu}, ID_s, C_1, C_2\} \leftarrow Req$, $(ID_p, [ID_s], Cert_u, L_u, T_{exp}) \leftarrow \mathcal{T}_{pu}, VK_u \leftarrow Cert_u$
3: **if** $true \leftarrow \mathsf{Verify}_{VK_u}(Req, \sigma_{Req})$ **then**
4: **if** $T_{exp} < T_c$ **then**
5: return error
6: **else if** $\mathcal{T}_{pu} \in revocTable$ **then**
7: return error
8: **end if**
9: $Cert_p \leftarrow \mathsf{lookupProvider}(\mathsf{ID_p}), VK_P \leftarrow Cert_p$
10: **if** $false \leftarrow \mathsf{Verify}_{VK_p}(\sigma_{\mathcal{T}_{pu}}, \mathcal{T}_{pu})$ **then**
11: drop connection
12: **end if**
13: retrieve L_D for *content_name*
14: **if** $L_u \leq L_D$ **then**
15: drop connection and return error
16: **end if**
17: **if** ID_s is Dynamic Service Request **then**
18: **if** $true \leftarrow \mathsf{checkServerCapability}(ID_s)$ **then**
19: $K \leftarrow \mathsf{ABE.Decrypt}(M_{pk}, [A_e], C_1)$
20: $D \leftarrow \mathsf{Dec}_K(C_2)$
21: return $\mathsf{fulfillService}(D)$
22: **else**
23: $\mathsf{requestService}(Req, \sigma_{Req})$
24: **end if**
25: **else if** ID_s is Static Service Request **then**
26: **if** $true \leftarrow \mathsf{contentCacheLookup}(C_1)$ **then**
27: return $\mathsf{contentCacheRetrieve}(C_1)$
28: **else**
29: $\mathsf{requestService}(Req, \sigma_{Req})$
30: **end if**
31: **end if**
32: **else**
33: drop connection and return error
34: **end if**

PEC server e. As shown in Protocol 6, for revoking u, p removes $\mathcal{T}_{pu}$ from its *userTable* and forwards the token to its AIA that is hosted on the Cloud (Line 1). On receiving the revocation notification, $\mathbb{C}$ retrieves the list of all PEC servers who could serve u, and notifies them of the revocation (Lines 2-6). Revocation communication could be done scalably using distributed ledger [15, 29]. On revocation notification, each e will add $\mathcal{T}_{pu}$ to its $revocTable$ (Lines 7-8). If a token is expired, the entry can be removed.

In *APECS*, revocation of e is handled by the AIAs (Provider and Base stations) who execute the MABE algorithms via a local system re-key for the non-revoked PEC servers (updating M_{pk} to M'_{pk}). While ABE revocation is generally costly, due to system re-keying, *APECS* uses the MABE scheme in a way that optimizes PEC server revocation. When p decides to revoke e, it instructs e's base station (the second AIA managing far less number of PEC servers that p) to revoke e (Lines 1-2 in Protocol 7). The base station updates its public/private key pair and shares it with other AIAs, enabling AIAs (including the base stations) to run the MABE system setup for calculating M'_{pk} (Line 3). The base station then generates and

Protocol 6 User Revocation

{**At Provider**}

1: delete $\mathcal{T}_{pu}$ for u from $userTable$ and notify $\mathbb{C}$.

{**At Cloud**}

2: receive $\{\mathcal{T}_{pu}\}$ from p.
3: $E \leftarrow$ edgeServersWith($[ID_s]$), $E \subset \mathbb{E}$
4: **for each** $e \in E$ **do**
5: notify e of revoked $\mathcal{T}_{pu}$
6: **end for**

{**At Edge Server**}

7: receive $\mathcal{T}_{pu}$ from $\mathbb{C}$.
8: add $\mathcal{T}_{pu}$ to $revocTable$.

distributes a new set of secret keys for all of its PEC servers, except the revoked one (Lines 5-11). Finally, the base station broadcasts the new public parameters to the users in its vicinity (Line 12). This revocation localizes the re-keying operation to only the PEC servers associated with the revoking base station.

The PEC server revocation can be optimized if base stations are more involved in service orchestration. In such case, on receiving a service request, the base station acts as a broker and steers requests away from revoked PEC servers. In addition, the service provider's AIA hosted on $\mathbb{C}$ can update the base stations' $revocTable$ with a set of revoked PEC servers reported by other base stations to prevent revoked PEC servers from migrating to other base stations. Thus, minimizing the number of system re-keys and eliminating the need to re-key the local PEC servers per revocation.

5.5 APECS PKC-based Design

We also propose *APECS PKC* as an alternative *APECS* design that utilizes the traditional public key cryptosystem (PKC), which has a less complex system and security configuration (using transport layer security and the PKC infrastructure). *APECS PKC* is suitable for static scenarios where the user is aware of the PEC servers and their services (through a service discovery process, which we do not discuss) and the user(s) and PEC servers can synchronously interact.

Protocol 7 Edge Server Revocation

{**at Provider**}

1: identify $\hat{e} \in E$ that should be revoked.
2: notify base station $b_n \in \mathbb{B}$ that $\hat{e}$ is associated with.

{**at AIAs**}

3: $\left(M'_{pk} = (\text{sysparam}, apk_1, \ldots, apk'_n), ask_1, \ldots, ask'_n\right) \leftarrow$ ABE.Setup($1^\lambda, n$); {updated b_n's public/private key pair}

{**at Base Station**}

4: $E_{b_n} \subset \mathbb{E}$; {all PEC servers associated with b_n}
5: **for each** $e \in E_{b_n}$ **do**
6: **if** $e \neq \hat{e}$ **then**
7: $[A'_e] \leftarrow$ ABE.KeyGen($M'_{pk}, ask_e, id_e, [ID_s]$)
8: store $\{e, [A'_e]\}$ in $serverTable$
9: return $\{e, [A'_e]\}$ to e
10: **end if**
11: **end for**
12: broadcast M'_{pk} to $\mathbb{U}$

Considering *APECS PKC* is an obvious choice for static environments, we will discuss its design and assess its efficacy. *APECS PKC* also provides a very good foil to compare *APECS* more thoroughly. In this approach, a PEC server has to obtain a customized token (similar to users' tokens) from each service provider and the base station, it is associated with, to prove its affiliation with them to the user (affiliation is the token pair from the provider and the base station). This is in contrast with the *APECS* design where PEC servers have to obtain MABE credentials. The tokens issued by the providers and base stations are signed by them for authentication.

To request a service, after selecting a PEC server, the user establishes a Transport Layer Security (TLS) connection with the PEC server to securely share her authentication token obtained from the provider. Upon successful verification of the user's token (Lines 2-12 of Protocol 5), the PEC server shares its tokens (from the service provider and the base station) with the user. This allows the user to verify the shared tokens' integrity and provenance (in a process similar to Lines 2-12 of Protocol 5), and by extension, the PEC server's authenticity for the requested service. On successful mutual authentication/authorization, the user securely shares her data with the PEC server for service execution and the server responds with the computation results. We note that the TLS channel should be established using the certificates that are contained in the tokens (for both the user and the PEC server) to avoid a challenge-response for mutual authentication.

6 APECS SECURITY ANALYSIS

6.1 Formal Security Analysis

We now provide a formal analysis of *APECS* in the Universal Composability (UC) Framework [2]. The notion of UC security is captured by the pair of definitions below:

Definition 6.1. *(UC-emulation [2]) Let π and ϕ be probabilistic polynomial-time (PPT) protocols. We say that π UC-emulates ϕ if for any PPT adversary $\mathcal{A}$ there exists a PPT adversary $\mathcal{S}$ such that for any balanced PPT environment $\mathcal{Z}$ we have*

$$\text{EXEC}_{\phi,\mathcal{S},\mathcal{Z}} \approx \text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$$

Definition 6.2. *(UC-realization [2]) Let $\mathcal{F}$ be an ideal functionality and let π be a protocol. We say that π UC-realizes $\mathcal{F}$ if π UC-emulates the ideal protocol for $\mathcal{F}$.*

We define an ideal functionality, $\mathcal{F}_{\text{APECS}}$, consisting of five independent ideal functionalities, $\mathcal{F}_{\text{register}}$, $\mathcal{F}_{\text{response}}$, $\mathcal{F}_{\text{revoke}}$, $\mathcal{F}_{\text{smt}}$, $\mathcal{F}_{\text{sig}}$. $\mathcal{F}_{\text{register}}$ models the user and edge servers' registration processes, $\mathcal{F}_{\text{response}}$ models the processing of a user's service request, and $\mathcal{F}_{\text{revoke}}$ models the revocation functionality. We use two helper functionalities from [2], $\mathcal{F}_{\text{sig}}$ and $\mathcal{F}_{\text{smt}}$, to model ideal functionalities for digital signatures and secure/authenticated channels, respectively. We assume that $\mathcal{F}_{\text{APECS}}$ maintains internal state that is accessible at any time to $\mathcal{F}_{\text{register}}$, $\mathcal{F}_{\text{response}}$ and $\mathcal{F}_{\text{revoke}}$, specifically three tables, *uTable*, *sTable* and *dTable*. The parties that interact with the ideal functionalities are the members of sets of edge servers, $\mathbb{EC}$, service providers, $\mathbb{SP}$, base stations $\mathbb{BS}$, and a user u. We assume that each member of the three sets has a unique identifier. The *dTable* contains all data provided by different service providers, *uTable* contains details about the services a user is registered for,

Functionality $\mathcal{F}_{\text{register}}$

(1) When a service provider, $SP \in \mathbb{SP}$ sends a request, (register, *spid*, *sname*, *scat*, *sdata*, *stype*), $\mathcal{F}_{\text{register}}$ adds t_d = (*spid*, *sname*, *scat*, *sdata*, *stype*) to the *dTable*; if t_d already exists, $\mathcal{F}_{\text{register}}$ returns t_d. When an *SP* sends a request (update, *spid*, ·, ·, ·, ·), $\mathcal{F}_{\text{register}}$ updates t_d. When an *SP* sends a request (deregister, *spid*, *sname*, *scat*), $\mathcal{F}_{\text{register}}$ deletes the corresponding tuple t_d from *dTable*.

(2)(a) When a user u sends a registration request, (register, *uid*, *scat*, *spid*) to $\mathcal{F}_{\text{register}}$ (where *uid*, *spid* are the user's and service provider's unique identifiers, *scat* is the category of the service u wishes to subscribe to), $\mathcal{F}_{\text{register}}$ checks if there exists a tuple t_u = (*uid*, *scat*, *spid*) in *uTable*. If yes, $\mathcal{F}_{\text{register}}$ returns t_u to u, and forwards (exists, t_u) to $\mathcal{S}$. Else $\mathcal{F}_{\text{register}}$ sends a message, (register, *uid*, *scat*) to $SP \in \mathbb{SP}$ whose identifier is *spid*. If *SP* responds with an "allow", $\mathcal{F}_{\text{register}}$ adds tuple t_u = (*uid*, *scat*, *spid*) to *uTable*, returns "success" to u, and forwards (newreg, t_u) to $\mathcal{S}$. Else $\mathcal{F}_{\text{register}}$ returns $\perp$ to u, and forwards (failReg, *uid*, *scat*, *spid*) to $\mathcal{S}$.

(b) When u sends a request (update, *uid*, *scat*′, *spid*), $\mathcal{F}_{\text{register}}$ retrieves a tuple t_u = (*uid*, ·, *spid*) in *uTable*. $\mathcal{F}_{\text{register}}$ sends a message, (update, *uid*, *scat*′), to *SP*. If *SP* replies with $\perp$, $\mathcal{F}_{\text{register}}$ returns $\perp$ to u, and forwards (failUpdate, t_u) to $\mathcal{S}$. Else $\mathcal{F}_{\text{register}}$ updates or creates (if the retrieval of t_u returned $\perp$) a tuple t_u with (*uid*, *scat*′, *spid*), and returns "success" to u, and forwards (successUpdate, t_u) to $\mathcal{S}$. In case a new t_u was created, $\mathcal{F}_{\text{register}}$ forwards (newreg, t_u) to $\mathcal{S}$.

(c) If a user sends a request (deregister, *uid*, ·, *spid*), $\mathcal{F}_{\text{register}}$ deletes tuple t_u = (*uid*, ·, *spid*) in *uTable* and forwards (deregister, *uid*, ·, *spid*) to *SP*, and $\mathcal{S}$.

(3)(a) When an edge server, $EC \in \mathbb{EC}$, identified by *ecid* sends a request to $\mathcal{F}_{\text{register}}$, (register, *spid*, *ecid*, *bsid*), where *spid* is the identifier of a service provider $SP \in \mathbb{SP}$ whose services *EC* wants to offer via *bsid* which denotes the identifier of a base station $BS \in \mathbb{BS}$, $\mathcal{F}_{\text{register}}$ checks if there exists a tuple in *sTable*, t_s = (*spid*, *ecid*, *bsid*). If yes, $\mathcal{F}_{\text{register}}$ returns t_s to *EC*, and forwards (exists, t_s) to $\mathcal{S}$. Else $\mathcal{F}_{\text{register}}$ sends a message (register, *spid*, *ecid*, *bsid*) to *SP* and *BS*. If *SP* and *BS* both respond with "allow", $\mathcal{F}_{\text{register}}$ adds tuple t_s = (*spid*, *ecid*, *bsid*) to *sTable*, collects all tuples t_d = (*spid*, ·, ·, ·, ·) from *dTable*, sends them to *EC*, and forwards (newreg, t_s, t_d) to $\mathcal{S}$. If either of them respond with $\perp$, it returns $\perp$ to *EC*, and forwards (failReg, *spid*, *ecid*, *bsid*) to $\mathcal{S}$.

(b) If *EC* sends a request (deregister, *spid*, *ecid*, *bsid*), $\mathcal{F}_{\text{register}}$ deletes tuples t_s from *sTable*, and forwards (deregister, *spid*, *ecid*, *bsid*) to *SP*, *BS*, and $\mathcal{S}$.

Figure 2: Ideal functionality for Service Registration

and *sTable* contains information about the service providers an edge server provides services on behalf of. We now briefly describe the design of our ideal functionalities.

$\mathcal{F}_{\text{register}}$: The $\mathcal{F}_{\text{register}}$ functionality shown in Figure 2 handles the system setup and registration/de-registration of a user u and members of $\mathbb{EC}$. This also handles registration of data associated with members of $\mathbb{SP}$, as well as service updates. When a service provider *SP* wishes to register, it initiates contact with $\mathcal{F}_{\text{register}}$ by sending a tuple (register, *spid*, *sname*, *scat*, *sdata*, *stype*), where *spid* denotes the unique identifier of *SP*, *sname* denotes the name of the service *SP* is offering, *scat* denotes the service category of *sname* (e.g., bronze, gold, silver), and *stype* indicates if a given *sname* is associated with static or dynamic requests. For static requests, e.g., movies, *sdata* contains the relevant data files, for dynamic requests, e.g., image annotation, *sdata* contains the algorithms needed to process the user-supplied input. $\mathcal{F}_{\text{register}}$ creates a new tuple in *dTable* containing the data supplied by *SP*, if one does not exist. Since each tuple is uniquely identified by (*spid*, *sname*, *scat*), when *SP* de-registers, it just needs to send (*spid*, *sname*, *scat*) to $\mathcal{F}_{\text{register}}$, who deletes the tuple from *dTable*.

When a user u, identified by *uid*, wants to register for a service, it contacts $\mathcal{F}_{\text{register}}$ with (register, *uid*, *scat*, *spid*). We assume that a user can register for only one category of service with an *spid*. If *SP* permits u to register, $\mathcal{F}_{\text{register}}$ adds u's information to *uTable*, and forwards the registration information to $\mathcal{S}$. Similarly, when an already-registered u wishes to update their service category to *scat*′, $\mathcal{F}_{\text{register}}$ will check with *SP* and act accordingly. $\mathcal{F}_{\text{register}}$ will also notify $\mathcal{S}$ whether the update request was successful. When u terminates its service and de-registers from an *spid*, $\mathcal{F}_{\text{register}}$ deletes the unique tuple (*uid*, ·, *spid*) without needing to ask *SP*'s permission, but informs *SP* and $\mathcal{S}$ about u's de-registration.

An edge server *EC* will register with both a service provider *SP* and a base station *BS* (with *bsid*) to model the fact that in the real world, all entities communicate over networks through their local base stations. We assume each *EC* will register with a unique (*SP*, *BS*) pair, i.e., the tuple (*spid*, *ecid*, *bsid*) is unique. If both *SP* and *BS* approve of *EC*'s request, $\mathcal{F}_{\text{register}}$ will add *EC*'s information to *sTable*, else it will notify *EC* and $\mathcal{S}$ that the registration request was denied. At this point, $\mathcal{F}_{\text{register}}$ will also send to *EC* and $\mathcal{S}$ information about all the services *EC* is registered for with all *SP*s. When an *EC* wishes to stop providing services on behalf of an *SP*, it de-registers itself. $\mathcal{F}_{\text{register}}$ deletes the unique tuple (*spid*, *ecid*, *bsid*) from *sTable*, without needing to ask *SP*'s or *BS*'s permission, but informs them and $\mathcal{S}$ about it.

$\mathcal{F}_{\text{response}}$: The $\mathcal{F}_{\text{response}}$ functionality shown in Figure 3 handles a service request from a user identified by *uid*. When the *uid*, that registered with service provider *spid*, submits a request to $\mathcal{F}_{\text{response}}$ for a service identified by *sname*, it sends a request containing (*spid*, *sname*, *uid*, *bsid*, *udata*). The *bsid* and *spid* in the request help identify the list of *ecid*s connected to the base station *bsid* that the user is connected to, and that can process the user's request. The request also includes user data (*udata*) which would be used by *ecid* if *sname* is a dynamic service that needs to process the user data, *udata* would be $\perp$ if the request is a static request.

Once $\mathcal{F}_{\text{response}}$ receives the request from the user, it forwards (recvReq, *spid*, *sname*, *uid*, *bsid*, *udata*) to $\mathcal{S}$ and retrieves the tuple t_d = (*spid*, *sname*, *scat*, *sdata*, *stype*) from *dTable* containing *spid* and *sname*. If no such tuple exists, then the service requested by *uid* is not offered by *spid* and a $\perp$ is returned to the user along with (failReq, *spid*, *sname*, *uid*, *bsid*, *udata*) to $\mathcal{S}$, otherwise $\mathcal{F}_{\text{response}}$ continues to the next step. Next, $\mathcal{F}_{\text{response}}$ checks whether *uid* is authorized to access service *sname* from *spid*. It retrieves tuple t_u

Functionality $\mathcal{F}_{\text{response}}$

(1) Upon receipt of a request (*spid*, *sname*, *uid*, *bsid*, *udata*) from a user, $\mathcal{F}_{\text{response}}$ retrieves the data tuple, t_d = (*spid*, *sname*, *scat*, *sdata*, *stype*) containing *spid* and *sname* from user's request and sends (recvReq, *spid*, *sname*, *uid*, *bsid*, *udata*) to $\mathcal{S}$. If t_d does not exist return $\perp$ to user and send (failReq, *spid*, *sname*, *uid*, *bsid*, *udata*) to $\mathcal{S}$.
(2) Then $\mathcal{F}_{\text{response}}$ checks the *uTable* and retrieves user tuple t_u = (*uid*, *scat*, *spid*) where *uid* in t_u is same as that in user's request, and *spid* and *scat* are same as those in t_d. If no such t_u exists return $\perp$ to user and send (failReq, *spid*, *sname*, *uid*, *bsid*, *udata*) to $\mathcal{S}$.
(3) $\mathcal{F}_{\text{response}}$ then retrieves all tuples matching t_s = (*spid*, ·, *bsid*) in *sTable* where *spid* and *bsid* in t_s is the same as that in the user's request. If no such tuples exists return $\perp$ to user and send (failReq, *spid*, *sname*, *uid*, *bsid*, *udata*) to $\mathcal{S}$.
(4) If all previous verifications pass, then $\mathcal{F}_{\text{response}}$ forwards request (*uid*, *sname*, *udata*) to all edge server *ecid*s in the t_s tuples that was retrieved in the previous step and forward (ReqEC, *uid*, *sname*, *udata*) to $\mathcal{S}$.
(5) Each *ecid*, on receiving a request from $\mathcal{F}_{\text{response}}$, does: 1) If *stype* associated with *sname* is "static", then *ecid* retrieves the *sdata* associated with *sname* and returns *msg* = *sdata* to $\mathcal{F}_{\text{response}}$. 2) If *stype* associated with *sname* is "dynamic", then *ecid* retrieves the *sdata* function associated with *sname*. It runs sdata(*udata*) $\rightarrow$ *msg* and forwards *msg* to $\mathcal{F}_{\text{response}}$.
(6) $\mathcal{F}_{\text{response}}$ forwards the first *msg* associated with the current request received from any *ecid* to *uid* and $\mathcal{S}$, and discards all other following *msg*s.

Figure 3: Ideal functionality for Responding to Requests

from *uTable* containing *uid*, *spid* from the request, and *scat* from t_d. If no such tuple exists, then this reflects that the user is not signed up with the given *spid* to access services tagged under *scat* category and a $\perp$ is returned to the user and $\mathcal{F}_{\text{response}}$ sends (failReq, *spid*, *sname*, *uid*, *bsid*, *udata*) to $\mathcal{S}$. $\mathcal{F}_{\text{response}}$ then retrieves all tuples t_s = (*spid*, ·, *bsid*) to identify all *ecid*s that can process the user's request. If no such tuple exists, then this indicates that there are no *ecid*s connected to *bsid* that can process the user's request and provide services on behalf of *spid*. $\mathcal{F}_{\text{response}}$ returns $\perp$ to the user along with (failReq, *spid*, *sname*, *uid*, *bsid*, *udata*) to $\mathcal{S}$.

If all the above checks succeed, then $\mathcal{F}_{\text{response}}$ has a list of all *ecid*s available to process the user's request and *uid* is a verified subscriber to the requested service. $\mathcal{F}_{\text{response}}$ sends (*uid*, *sname*, *udata*) to *ecid*s in the tuples t_s, retrieved in the previous step and forwards (ReqEC, *uid*, *sname*, *udata*) to $\mathcal{S}$. When each *ecid* receives the request, if *sname* is a static request, then *ecid* retrieves the data associated with *sname* and responds to $\mathcal{F}_{\text{response}}$ with *msg* containing *sdata*. If *sname* is a dynamic request, then the *ecid* retrieves the algorithms, *sdata* associated with *sname*, processes *udata*, *sdata*(*udata*) $\rightarrow$ *msg*, and responds to $\mathcal{F}_{\text{response}}$ with *msg* which contains the output. $\mathcal{F}_{\text{response}}$ on receiving the first *msg* from any *ecid* forwards it to *uid* and drops all subsequent *msg*s from other *ecid*s. $\mathcal{F}_{\text{response}}$ also forwards *msg* to $\mathcal{S}$.

Functionality $\mathcal{F}_{\text{revoke}}$

(1) Upon receipt of a request (revoke, *spid*, *ecid*) from SP (*spid*), $\mathcal{F}_{\text{revoke}}$ checks the *sTable* for all tuples t_s = (*spid*, *ecid*, ·). If any exist, $\mathcal{F}_{\text{revoke}}$ deletes the tuples from *sTable* and forwards (revoke, *spid*, *ecid*) to all *bsid*s in the deleted tuples. Else returns $\perp$ to SP.
(2) Upon receipt of a request (revoke, *uid*, *spid*) from SP identified by *spid*, $\mathcal{F}_{\text{revoke}}$ checks the *uTable* for all tuples of the form t_u = (*uid*, ·, *spid*). If any exist, it deletes the tuples from *uTable* and returns "success". Else, it returns $\perp$.

Figure 4: Ideal functionality for User/Edge Server Revoke

$\mathcal{F}_{\text{revoke}}$: The $\mathcal{F}_{\text{revoke}}$ functionality shown in Figure 4 handles the revocation of an edge server by service provider SP. The functionality also handles the revocation of a user's access to services provided by SP. When $\mathcal{F}_{\text{revoke}}$ receives a request (revoke, *spid*, *ecid*) from service provider *spid*, it checks the *sTable* for the existence of all tuples (*spid*, *ecid*, ·) and deletes all such tuples if any exist. This effectively revokes an *EC* identified by *ecid* from providing services on behalf of SP. When $\mathcal{F}_{\text{revoke}}$ receives a request (revoke, *uid*, *scat*, *spid*) from service provider *spid*, it checks the *uTable* for existence of a tuple (*uid*, *scat*, *spid*) and deletes it if such a tuple exists. This effectively revokes an user identified by *uid* from services provided by SP under *scat* subscription category.

We further discuss the design of our ideal functionalities and provide the proof of the following theorem in Appendix 11.

Theorem 6.1. *Let $\mathcal{F}_{\text{APECS}}$ be an ideal functionality for* APECS. *Let $\mathcal{A}$ be a probabilistic polynomial-time (PPT) adversary for* APECS, *and let $\mathcal{S}$ be an ideal-world PPT simulator for $\mathcal{F}_{\text{APECS}}$.* APECS *UC-realizes $\mathcal{F}_{\text{APECS}}$ for any PPT distinguishing environment $\mathcal{Z}$.*

6.2 Informal Security Analysis

Before elaborating on malicious PEC servers and service consumers, we briefly mention the impact of misbehaving cloud providers and base stations. In *APECS*, the Cloud is the enabler of the communication between the PEC servers (hosting the service providers instances) and the service providers. As such, it does not play any active operational role and hence, its malicious behavior does not impact the system's security. In this paper, we built a federated authority by considering two AIAs (one at the service provider and one at the base station connected to the user device) so that the malicious intent of one does not compromise the security and privacy of users' data. Only with both AIAs being malicious, the users' data can be decrypted illegally. Thus, malicious base stations alone cannot violate users' privacy. Note that using two AIAs is only for illustration purposes. *APECS* can use multiple AIAs (N), in which case, the system tolerates N-1 AIAs going rogue. In fact, we use three AIAs for illustration in our experimental results (Section 7).

6.2.1 Malicious PEC Server. As per Section 3.3, a malicious PEC server may hijack the communication or impersonate legitimate PEC servers to obtain users' data. Moreover, a malicious PEC server (authorized server that does not follow the protocols) may collude with an unauthorized user to illegitimately provide a service. In *APECS*, the user encrypts the data (if needed) using a symmetric cipher and encrypts the corresponding symmetric key using MABE. This allows only the authorized PEC servers (having requisite secret

keys from all AIAs) to successfully decrypt the symmetric key and decrypt the user's data. This prevents the unauthorized servers from obtaining the user's data (threat *(e)* in the threat model).

A colluding PEC server could provide either/both the static and the dynamic service to an unauthorized user. In the former, an unauthorized user obtains a content either from the malicious PEC server or by intercepting the channel. Encryption of the content by the service provider using a key pre-shared with the users (using techniques such as ABE or broadcast encryption) can ensure that unauthorized users cannot decrypt the content (threat *(f)*) [16, 17]. In the latter case, we argue that there is no incentive for a PEC server to use its resources for executing a service without being compensated assuming an accounting/billing framework exists for tracking legitimate service execution for compensation.

We also note that a malicious service provider may attempt to orchestrate a denial of service (DoS) attack on the PEC servers by assigning expired or short-lived tokens to its users. However, obtaining a fresh token from service providers incurs negligible cost (it only requires one round trip time per user) and does not impose any overhead on the PEC servers processing. Furthermore, by orchestrating such a DoS campaign, the service provider sacrifices its users' quality of experience, which only damages its reputation. Thus, we do not consider such DoS attacks a common threat.

6.2.2 Malicious Service Consumer. Following the threat model, consumers' threats include requesting services without valid tokens (*e.g.,* forged or expired) and unauthorized use of valid tokens (*e.g.,* shared, intercepted, or replayed). In *APECS*, PEC servers assess tokens' validity by verifying the service provider's signatures on tokens and the consumers' signatures on requests (request include the signed tokens). A provider's signature on a token can be verified by its certificate while the service consumer's signature should be verified using the certificate embedded in the signed token. This prevents a malicious consumer from sharing his token with unauthorized users (threats *(a)* and *(d)*). The only possibility for a malicious service consumer to successfully share his token is to craft a signed request and share it with the unauthorized user. For this attack to be successful, the malicious consumer has to further modify the timestamp of the request's signature or forward it instantly. We note that such an attack can be thwarted by updating *APECS* with a challenge-response interaction between the service consumer and the corresponding base station ahead of service request. The base station uses the consumer's certificate embedded in the token to validate the identity using the challenge-response process and subsequently allows the consumer to request the service.

Prior to signature verification, edge servers verify tokens' freshness using the embedded expiry time dropping the requests with stale tokens (threat *(c)*). Moreover, edge servers compare the requested service provider's identity with the one contained in the token to prevent a malicious consumer from using a valid token for other services (*e.g.,* using face detection token for the image annotation service). Thus, by virtue of the signature on the token and its embedded information, edge servers can detect and drop forged or expired tokens (threat *(b)*).

7 EXPERIMENTAL RESULTS AND ANALYSIS

7.1 Implementation Scope

The reference implementation of *APECS* comprises four components: the user engine, the PEC server engine, the service provider engine, and the cloud engine, all implemented in C++. We used Pairing-Based Cryptography (PBC) library (v.0.5.14) and C Programming Language (v.9.3.0) for the MABE implementation, and C++ libssl-dev library (v.1.1.1) for the symmetric key functionality. The MABE framework was evaluated using the default "Type a" curve provided by the PBC library which uses symmetric pairings for all the pairing operations. For communication between these engines, we used the gRPC framework (v.1.20.0). The user engine is in charge of executing functions on the user's behalf, including the user's data related functions, *i.e.,* generation, storage, and encryption, token related functions, *i.e.,* obtaining, storing, and consuming, and data encryption/decryption. For *APECS PKC*, we extended the user engine by verifying the PEC servers' tokens. The user engine is implemented in 1630 source lines of code (SLoC). The PEC server engine performs authentication, authorization, and users' service execution. We implemented the token verification process in C++ using the jwt-cpp library. Features such as $revocTable$ and content cache are maintained by calls to a local NO-SQL mongoDB database (v.4.2.9). All communication uses gRPC framework with TLS 1.2. The PEC server engine is implemented in 2000 SLoC.

The provider engine operates the service provider's functionalities, such as storage of $userTable$ in a local NO-SQL mongoDB instance, user registration, token renewal, revocation, and content delivery using gRPC C++ library. It further cooperates in the setup of the MABE framework through the use of the PBC library. The provider engine is implemented in 1950 SLoC. The cloud engine hosts the service providers' AIAs. As such, it runs a portion of MABE framework setup, which is implemented using the PBC library. The cloud engine maintains a local NO-SQL mongoDB instance to store providers' profiles and revoked tokens. Using gRPC framework, it maintains standardized API routes for the invocation of edge servers, provider registration, and access-denied notification. The cloud engine is implemented in 600 SLoC.

For comparison, we prototyped an access control enforcement mechanism that uses trusted centralized Cloud for enforcement of access policies–a common approach that is currently adopted by many providers. In our prototype, users obtain authentication tokens (Definition 4.1) from the service providers and share them with the Cloud over secure channels (TLS) whenever requesting a service. The Cloud follows the *APECS PKC* token verification process to authenticate and authorize users. However, due to the common assumption of Cloud's trustworthiness, users do not authenticate the Cloud (a one-way authentication of the users).

7.2 Experiment Setup

The assessment of MABE performance in isolation was performed on three device classes. The first device class is that of a Compact Edge device which is represented by a Jetson TX2 with 8 GB of RAM and a CPU cluster composed of a dual-core NVIDIA Denver2 and a quad-core ARM Cortex-A57, both operating at 2.00 GHz. The second device class, a handheld device, is represented by an InstaGENI virtual machine (VM) with 1 GB of RAM and a 2.10 GHz

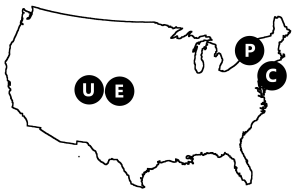


Figure 5: Node placement in GENI. User Ⓤ and Edge Server Ⓔ are hosted at the University of Colorado. Provider Ⓟ and Cloud Ⓒ are hosted at Cornell and New York Universities.

Intel Xeon CPU E5-2450. Finally, the third device class used for MABE performance evaluation is a Desktop with 16 GB of RAM and a 3.60 GHz Intel Xeon W-2123 CPU.

In our experiments, we configure the MABE system with three AIAs to represent a more complicated scenario for studying scaling (instead of the relatively simpler scenario with two AIAs–one service provider and one base station). An access policy has maximum two attributes per AIA. In realistic operating scenarios of *APECS*, we do not expect more than two attributes per AIA, *e.g.*, AIA's identity and service type. For consistency, in *APECS PKC* each PEC server uses three tokens per user request. We perform *APECS*' reference implementation on a network consisting of four virtual machines (VMs) hosted on the distributed GENI testbed [1]. We chose the GENI testbed as it provides a large-scale and geographically distributed network experiment infrastructure–the closest resemblance to real networks. We deployed each VM in different GENI Aggregates across the United States to resemble a true edge-cloud network topology. As shown in Figure 5, the instance at New York University runs the cloud engine, the user and PEC server engines run on dedicated VMs at University of Colorado, and the provider engine runs at Cornell University. In assessing the access control throughput, we deployed our PEC server engine on Desktop-class and Handheld-class platforms.

7.3 Results and Analysis

We benchmarked the performance of MABE using the PBC library, encrypting and decrypting a 512 bit symmetric key on three devices classes mentioned above, namely Compact Edge, Handheld, and Desktop. The PBC library is built on top of the GNU Multiple Precision Arithmetic Library (GMP) library. Figure 6 represents the results averaged over 1000 paired encryption-decryption runs. For the Compact Edge device–the lowest computation capability–encryption took about 20.6 milliseconds, while decryption took

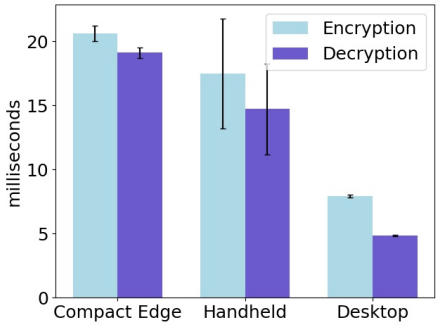


Figure 6: Benchmark timing of multi-authority attribute-based encryption [3] across multiple platforms.

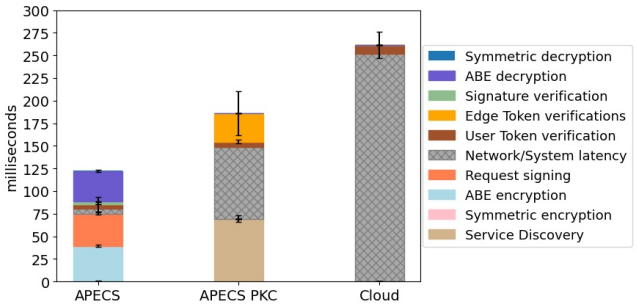


Figure 7: Comparison of average runtime for proposed and contemporary access control approaches.

19.1 milliseconds. Using the handheld device reduced the encryption latency to 17.4 milliseconds and decryption to 14.7 milliseconds. We note that the handheld device is represented by a VM instance which explains the presence of a larger error range. Finally, for the desktop device, encryption was completed in 7.9 milliseconds and decryption was completed in 4.8 milliseconds.

In Figure 7, we present the results comparing the average runtime of APECS (both MABE and the PKC approaches) with contemporary access control approaches involving the Cloud. We note that the combined code, which combines the MABE implemented in C with the networking and PKC in C++ still has room for optimization[3]. We now discuss the results from the partially optimized code. We benchmarked the performance of *APECS*, *APECS PKC*, and the cloud-based access control on the GENI testbed. We measured the end-to-end latency of these schemes for 1000 service requests while timing the individual components that make up the complete interaction. As shown in Figure 7 *APECS* was the fastest in performing mutual access control with around 123 milliseconds, followed by *APECS PKC* with 186 milliseconds, and the Cloud access control with 262 milliseconds. Note that *APECS* drastically reduced the Cloud access control latency, by 50%, despite performing mutual access control between the users and PEC servers (the Cloud prototype performs only user authentication and authorization). We highlight the simplicity-efficiency trade-off in *APECS* and *APECS PKC*: *APECS PKC*'s simpler design comes with a higher mutual authentication latency (roughly 50% increased latency), which is undesirable in many dynamic edge applications.

Table 2: Averaged Latency (msecs) Across Three Approaches

Operations	APECS	APECS PKC	Cloud
Service Discovery	—	69.5	—
Symmetric Encryption	0.7	—	—
ABE Encryption	39.0	—	—
Request Signing	35.6	—	—
Network/System Latency	5.8	79.5	252.0
User Token Verification	4.8	5.8	9.6
Edge Token Verification	—	31.4	—
Request Signature Verification	2.7	—	—
ABE Decryption	33.8	—	—
Symmetric Decryption	0.5	—	—
Total	**122.7**	**186.1**	**261.6**

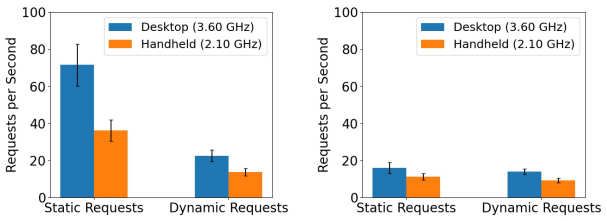
For *APECS PKC*, we implemented a rudimentary service discovery process, in which the user securely obtains the list of eligible

[3]Code is available on https://github.com/nsol-nmsu/APECS.

Table 3: *APECS* Communication Complexity

Provider Reg.	PEC Server Reg.	User Reg.	Service/Data Request/Response	User Revocation	PEC Revocation
$O(1)$	$O(\|\mathbb{P}+\mathbb{B}\|)$	$O(1)$	$O(1)$	$O(\|\mathbb{E}\|)$	$O(\|\mathbb{P}+\mathbb{E}\|)$

PEC servers (for a given service) from the base station ahead of service request. Note, as we concentrate on access control, service discovery is out of the scope for this paper. In *APECS PKC*, the end-to-end latency is composed of service discovery and is dominated by the secure communication between the user and PEC server (both using TLS connection). Finally, in the cloud-based access control, the secure communication between the user and the Cloud is the dominant portion of the authentication latency. Table 2 includes the averaged timing of each individual operation in *APECS*, *APECS PKC*, and Cloud. The missing values correspond to operations that are not needed in the corresponding approaches. We note that the network latency in APECS PKC and Cloud encompass the setup and encryption and decryption in the TLS; no ABE operation has been used in these two approaches.



(a) APECS (b) APECS PKC

Figure 8: Access control enforcement throughput for static and dynamic service requests on two platforms.

Finally, we benchmark the throughput of *APECS* and *APECS PKC* on a single PEC server. We define the throughput as the number of authentication and authorization operations that a PEC server can perform in unit time (Figure 8). To eliminate network and communication latency based variances we eliminate them by running all four components of *APECS* on the same machine (the one that was being tested). In this experiment, the user sent 1000 service requests to the PEC server. Note that the service request processing does not include the service execution (*e.g.,* image annotation) to clearly identify the throughput of the access control process.

For *APECS*, when performing static service requests, the PEC server engine running on the Desktop was able to process an average of 71 requests per second while the PEC server engine on the Handheld processed an average of 36 requests per second (Figure 8(a)). In performing dynamic service requests, the Desktop engine averaged 22 requests processed per second while the Handheld engine averaged 14 requests processed per second. This was expected as dynamic service request processing includes MABE encryption and decryption operations while static request processing involves less compute-intensive cryptographic operations.

APECS PKC processed an average of 16 and 11 static service requests per second when running on the Desktop and Handheld devices, respectively (Figure 8(b)). As for dynamic service requests, *APECS PKC* processed an average of 14 and 9 requests per second for the Desktop and Handheld devices, respectively. Overall, *APECS* outperformed *APECS PKC* both for static and dynamic service requests. For static service requests, this result was expected since *APECS* does not use MABE encryption and decryption while *APECS PKC* uses TLS channel for communication. For dynamic service requests, despite *APECS* using costly MABE operations, it outperformed *APECS PKC*–indicating MABE operations in *APECS* are more efficient than establishing TLS sessions in *APECS PKC*.

We also assessed *APECS* communication complexity (Table 3). The service provider registration process incurs constant communication complexity as it requires a round trip communication between the provider and the Cloud. Registering a PEC server requires a round trip communication between the PEC server and each of the AIAs, leading to $O(|\mathbb{P}+\mathbb{B}|)$ communication complexity per PEC server. As per the construction in [3], each PEC server has to obtain attributes from all AIAs corresponding to the providers. We note that, in *APECS*, the number of providers and base stations is constant. A user registration requires a round trip communication between the user and the service provider, resulting in constant communication complexity. Similarly, service request and response incurs constant communication complexity (we discount the potential of multiple packets being needed as determined by payload size). A user revocation process involves the delivery of the revoked token from the Cloud to the PEC servers that offer relevant services, resulting in $O(|\mathbb{E}|)$ communication complexity. Revoking a PEC server comprises a round trip communication from the service provider to the base station, interaction among the AIAs for the distribution of base station's new key, and the distribution of new attributes to the PEC servers that are connected to the base station. Thus, resulting in $O(|\mathbb{P}+\mathbb{B}|)$ communication complexity.

8 CONCLUSIONS

In this paper, we proposed, *APECS*, a distributed access control mechanism for the dynamic PEC ecosystem. In *APECS*, the authentication/authorization tasks are delegated to the PEC servers. *APECS* utilizes capability-based tokens and multi-authority ABE with an efficient revocation mechanism that does away with system-wide re-keying–the major drawback of ABE schemes. We also proposed *APECS PKC*, an alternative design suitable when the consumer and the PEC server can interact synchronously. Evaluation of our implementations demonstrated the practicality of our mechanisms.

9 ACKNOWLEDGEMENTS

Research supported in part by Intel Labs, US NSF awards #1800088, #2028797, #1914635, EPSCoR Cooperative agreement #OIA-1757207, US DoE SETO grant #DE-EE0008774, and the US Federal Aviation Administration (FAA). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF, DoE, FAA or Intel Corp. We thank Austin Bos for his contribution to the system implementation. We also thank Dr. Mattijs Jonker for his helpful shepherding and the anonymous reviewers for their insightful feedback and suggestions.

REFERENCES

[1] M. Berman, J. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar. 2014. GENI: A federated testbed for innovative network experiments. *Computer Networks* 61 (2014), 5–23.

[2] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS*. IEEE, 136–145.

[3] Melissa Chase and Sherman S. M. Chow. 2009. Improving privacy and security in multi-authority attribute-based encryption. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*. ACM, 121–130.

[4] Cisco. 2017. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2016 to 2021 White Paper. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.

[5] Cisco. 2017. The Zettabyte Era: Trends and Analysis. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html.

[6] Michael W Condry and Catherine Blackadar Nelson. 2016. Using smart edge IoT devices for safer, rapid response with industry IoT control operations. *Proc. IEEE* 104, 5 (2016), 938–946.

[7] Hui Cui, Xun Yi, and Surya Nepal. 2018. Achieving scalable access control over encrypted data for edge computing networks. *IEEE Access* 6 (2018), 30049–30059.

[8] Kai Fan, Qiang Pan, Junxiong Wang, Tingting Liu, Hui Li, and Yintang Yang. 2018. Cross-domain based data sharing scheme in cooperative edge computing. In *2018 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 87–92.

[9] C. Freitag, J. Katz, and N. Klein. 2017. Symmetric-key broadcast encryption: The multi-sender case. In *International Conference on Cyber Security Cryptography and Machine Learning*. Springer, 200–214.

[10] Giulio Grassi, Kyle Jamieson, Paramvir Bahl, and Giovanni Pau. 2017. Parkmaster: An in-vehicle, edge-based video analytics service for detecting open parking spaces in urban environments. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. IEEE/ACM, 1–14.

[11] Dick Hardt et al. 2012. *The OAuth 2.0 authorization framework*. Technical Report. RFC 6749, October.

[12] Ruei-Hau Hsu, Jemin Lee, Tony QS Quek, and Jyh-Cheng Chen. 2018. Reconfigurable security: Edge-computing-based framework for IoT. *IEEE Network* 32, 5 (2018), 92–99.

[13] Kaiqing Huang. 2019. Multi-Authority Attribute-Based Encryption for Resource-Constrained Users in Edge Computing. In *2019 International Conference on Information Technology and Computer Application (ITCA)*. IEEE, 323–326.

[14] Mingxin Ma, Guozhen Shi, and Fenghua Li. 2019. Privacy-oriented blockchain-based distributed key management architecture for hierarchical access control in the IoT scenario. *IEEE Access* 7 (2019), 34045–34059.

[15] Zhuo Ma, Junwei Zhang, Yongzhen Guo, Yang Liu, Ximeng Liu, and Wei He. 2020. An efficient decentralized key management mechanism for VANET with blockchain. *IEEE Transactions on Vehicular Technology* 69, 6 (2020), 5836–5849.

[16] Satyajayant Misra, Reza Tourani, and Nahid Ebrahimi Majd. 2013. Secure content delivery in information-centric networks: Design, implementation, and analyses. In *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*. ACM, 73–78.

[17] Satyajayant Misra, Reza Tourani, Frank Natividad, Travis Mick, Nahid Ebrahimi Majd, and Hong Huang. 2017. AccConF: An access control framework for leveraging in-network cached data in the ICN-enabled wireless edge. *IEEE transactions on dependable and secure computing* 16, 1 (2017), 5–17.

[18] Muhammad Baqer Mollah, Md Abul Kalam Azad, and Athanasios Vasilakos. 2017. Secure data sharing and searching at the edge of cloud-assisted internet of things. *IEEE Cloud Computing* 4, 1 (2017), 34–42.

[19] Aafaf Ouaddah, Anas Abou Elkalam, and Abdellah Ait Ouahman. 2017. Towards a novel privacy-preserving access control model based on blockchain technology in IoT. In *Europe and MENA Cooperation Advances in Information and Communication Technologies*. Springer, 523–533.

[20] Yuwen Pu, Chunqiang Hu, Shaojiang Deng, and Arwa Alrawais. 2020. RPEDS: A Recoverable and Revocable Privacy-Preserving Edge Data Sharing Scheme. *IEEE Internet of Things Journal* 7, 9 (2020), 8077–8089.

[21] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. 2017. On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration. *IEEE Communications Surveys & Tutorials* 19, 3 (2017), 1657–1681.

[22] Reza Tourani, Satyajayant Misra, Travis Mick, and Gaurav Panwar. 2017. Security, Privacy, and Access Control in Information-Centric Networking: A Survey. *IEEE Communications Surveys & Tutorials* (2017).

[23] Reza Tourani, Srikathyayani Srikanteswara, Satyajayant Misra, Richard Chow, Lily Yang, Xiruo Liu, and Yi Zhang. 2020. Democratizing the Edge: A Pervasive Edge Computing Framework. *arXiv preprint arXiv:2007.00641* 1, 1 (2020), 1–7.

[24] Reza Tourani, Ray Stubbs, and Satyajayant Misra. 2018. TACTIC: Tag-based access control framework for the information-centric wireless edge networks. In *International Conference on Distributed Computing Systems*. IEEE, 456–466.

[25] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyengar, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2019. Towards scalable edge-native applications. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. IEEE/ACM, 152–165.

[26] Kaiping Xue, Weikeng Chen, Wei Li, Jianan Hong, and Peilin Hong. 2018. Combining data owner-side and cloud-side access control for encrypted cloud storage. *IEEE Transactions on Information Forensics and Security* 13, 8 (2018), 2062–2074.

[27] Kaiping Xue, Peixuan He, Xiang Zhang, Qiudong Xia, David SL Wei, Hao Yue, and Feng Wu. 2019. A Secure, Efficient, and Accountable Edge-Based Access Control Framework for Information Centric Networks. *IEEE/ACM Transactions on Networking* 27, 3 (2019), 1220–1233.

[28] K. Xue, X. Zhang, Q. Xia, D. Wei, H. Yue, and F. Wu. 2018. SEAF: A secure, efficient and accountable access control framework for information centric networking. In *Conference on Computer Communications*. IEEE, 2213–2221.

[29] Kan Yang, Jobin J Sunny, and Lan Wang. 2018. Blockchain-based decentralized public key management for named data networking. In *The international conference on computer communications and networks (ICCCN 2018)*. IEEE.

[30] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li. 2017. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. IEEE/ACM, 1–13.

[31] Qingyang Zhang, Quan Zhang, Weisong Shi, and Hong Zhong. 2018. Distributed collaborative execution on the edges and its application to amber alerts. *IEEE Internet of Things Journal* 5, 5 (2018), 3580–3593.

[32] M. Zhao, C. Hu, X. Song, and C. Zhao. 2019. Towards dependable and trustworthy outsourced computing: A comprehensive survey and tutorial. *Journal of Network and Computer Applications* 131 (2019), 55–65.

[33] Qian Zhou, Mohammed Elbadry, Fan Ye, and Yuanyuan Yang. 2018. Heracles: Scalable, Fine-Grained Access Control for Internet-of-Things in Enterprise Environments. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 1772–1780.

10 COMPUTATIONAL ASSUMPTIONS

Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two cyclic multiplicative groups of prime order q generated by g_1 and g_2 respectively, $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ be a bilinear map such that $\forall x \in \mathbb{G}_1, y \in \mathbb{G}_2$, and $a, b \in \mathbb{Z}_q$, $\hat{e}(x^a, y^b) = \hat{e}(x, y)^{ab}$, and $\hat{e}(g_1, g_2) \neq 1$.

Definition 10.1. *The Decisional Diffie-Hellman (DDH) problem in prime order group $\mathbb{G}$ =<g> is defined as follows: on input g, g^a, g^b, $g^c \in \mathbb{G}$, decide if c = ab or c is a random element of $\mathbb{Z}_q$.*

Definition 10.2. *Let algorithm BDH_Gen(1^λ) output the parameters $(\hat{e}(\cdot,\cdot), q, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ where there is an efficiently computable isomorphism ψ from $\mathbb{G}_2$ to $\mathbb{G}_1$. The Decisional Bilinear Diffie-Hellman (DBDH) problem is defined as follows: given $g_1 \in \mathbb{G}_1$, $g_2, g_2^a, g_2^b, g_2^c \in \mathbb{G}_2$ and $Z \in \mathbb{G}_T$ as input, decide if $Z = \hat{e}(g_1, g_2)^{abc}$ or $\hat{e}(g_1, g_2)^R$ for $R \in \mathbb{Z}_q$.*

Definition 10.3. *The k-Decisional Diffie-Hellman Inversion (k-DDHI) problem in prime order group $\mathbb{G}$ =<g> is defined as follows: on input a (k+2)-tuple $g, g^s, g^{s^2}, \ldots, g^{s^k}, g^u \in \mathbb{G}^{k+2}$, decide if $u = 1/s$ or u is a random element of $\mathbb{Z}_q$.*

Definition 10.4. *Let BDH_Gen(1^λ) output the parameters for a bilinear mapping $(\hat{e}) : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$. The eXternal Diffie-Hellman (XDH) assumption states that, for all probabilistic polynomial time adversaries $\mathcal{A}$, the DDH problem is hard in $\mathbb{G}_1$. This implies that there does not exist an efficiently computable isomorphism $\psi' : \mathbb{G}_1 \rightarrow \mathbb{G}_2$.*

11 UC SECURITY ANALYSIS

11.1 Discussion

The security properties *APECS* aims to provide are preventing unauthorized users from availing services, and preventing malicious edge servers from offering services they are not authorized to provide,

and accessing users' input and personal data. The design of our ideal functionalities must reflect these properties.

$\mathcal{F}_{\text{register}}$ enforces that users can register for availing an *SP*'s services only if the *SP* allows them to. When a user wants to upgrade/downgrade their service category, $\mathcal{F}_{\text{register}}$ forwards the request to *SP*, and allows the change only if *SP* permits it. An *SP* could, of course refuse a user's registration or service category update request, but we do not consider this as malicious behavior on the part of the *SP*, since an *SP* can decide whom it wants to provide services to. All details are stored in an internal table, *uTable* of $\mathcal{F}_{\text{APECS}}$, and cannot be modified by users and/or service providers. When a user is revoked by an *SP*, $\mathcal{F}_{\text{revoke}}$ promptly deletes the corresponding entry from *uTable*. When a user tries to request services of *SP* in a category it has not registered for, *e.g.,* a bronze member requesting gold member services, $\mathcal{F}_{\text{response}}$ will check the *uTable* and return $\perp$ to the user. This is also true when the user tries to access services from non-existent edge servers. Hence a user will never be able to improperly request services it has not signed up for.

When an edge server, *EC*, wants to provide services on behalf of an *SP*, it needs to register with *SP* and a base station, *BS*. We assume that when an *EC* registers with an *SP* and *BS*, it can provide all services offered by *SP* on *SP*'s behalf in all categories. This can be easily modified to account for various combinations of *sname*/*scat* offered by different *EC*s, but we do not depict them here for presentational clarity. An *EC*, based on *stype*, can distinguish whether an incoming user's service request is static or dynamic. In the static case, it returns the data (*e.g.,* movie), and in the dynamic case, it runs the algorithms stored in the *sdata* field of the corresponding *sname* on the user's input, *udata*, and returns the result to $\mathcal{F}_{\text{response}}$, which forwards it to the user. Thus, there is no way a malicious *EC* can provide unauthorized services to a user on behalf of an *SP*, either in collusion with the user or otherwise. Also, $\mathcal{F}_{\text{response}}$ will not forward any request containing *udata* to *EC*, unless it verifies that *EC* is indeed authorized to service that user's request, so *EC* cannot get unauthorized access to user data/inputs.

11.2 Proof

We now prove Theorem 6.1.

Proof: We give a series of games, each of which is indistinguishable from its predecessor by a PPT $\mathcal{Z}$.

Game 0: This is the same as the real-world *APECS*. $\mathcal{Z}$ interacts directly with *APECS* and $\mathcal{A}$.

Game 1: $\mathcal{S}$ internally runs $\mathcal{A}$ and simulates the secure and authenticated channels functionality $\mathcal{F}_{\text{smt}}$.

Lemma 11.1. *For all PPT adversaries $\mathcal{A}$ and PPT environments $\mathcal{Z}$, there exists a simulator $\mathcal{S}$ such that*

$$\text{Exec}_{\text{Game0},\mathcal{Z}} \approx \text{Exec}_{\text{Game1},\mathcal{Z}}$$

The two games are trivially indistinguishable since $\mathcal{S}$ just executes the simulator for $\mathcal{F}_{\text{smt}}$.

Game 2: $\mathcal{S}$ communicates with the honest parties and $\mathcal{A}$, and simulates the protocols of *APECS* with the help of $\mathcal{F}_{\text{APECS}}$. $\mathcal{A}$ can corrupt any user or *EC* at any point in time by sending a message "corrupt" to them. Once an entity is corrupted, all their information is sent to $\mathcal{A}$ and all further communication to and from the corrupted party is routed through $\mathcal{A}$. We now state and prove the following lemma:

Lemma 11.2. *For all PPT adversaries $\mathcal{A}$ and PPT environments $\mathcal{Z}$, there exists a simulator $\mathcal{S}$ such that*

$$\text{Exec}_{\text{Game1},\mathcal{Z}} \approx \text{Exec}_{\text{Game2},\mathcal{Z}}$$

SP, *EC*, *BS* create their respective key-pairs, *SP* sets up the services it offers and service categories. All public keys are published as part of M_{pk}. $\mathcal{S}$ gets *SP*'s public key, certificate, $Cert_p$, creates $spid, ID_p$, constructs the tuple t_d = (register, *spid*, *sname*, *scat*, $\cdot$, $\cdot$) and passes it on to $\mathcal{F}_{\text{register}}$ in the ideal-world who adds t_d to its *dTable*. In the real-world, $\mathcal{S}$ returns ID_p to *SP*. $\mathcal{S}$ receives a registration request from *EC*, $([ID_s], Cert_e)$, upon which it creates an *ecid* associated with *EC*, constructs tuple t_s = (register, *spid*, *ecid*, *bsid*) and forwards t_s to $\mathcal{F}_{\text{register}}$ in the ideal-world who adds t_s to its *sTable*. $\mathcal{S}$ then forwards $([ID_s], Cert_e)$ to *SP*, *BS* who will complete the registration in the real-world and return $(e, [A_e])$ to $\mathcal{S}$ who forwards it to *EC*. $\mathcal{S}$ also queries the key generation function of $\mathcal{F}_{\text{sig}}$, and simulates the key generation procedure for $\mathcal{S}_{\text{sig}}$ where $\mathcal{S}_{\text{sig}}$ is the simulation of the specific digital signature scheme being used. When an *EC* needs to get revoked, *SP* will forward to $\mathcal{S}$ the *ecid* ($\hat{e}$ in the real world). $\mathcal{S}$ will pass long $\hat{e}$ to the appropriate *BS* in the real world, and in the ideal-world, $\mathcal{S}$ will create and forward to $\mathcal{F}_{\text{revoke}}$ a tuple (revoke, *spid*, *ecid*). $\mathcal{F}_{\text{revoke}}$ will delete the corresponding tuple from *dTable*, and forward the successfully-processed revocation request to *bsid* via $\mathcal{S}$. In the real-world, *SP* and *BS* will update their respective parts of M_{pk} to M'_{pk} and make M'_{pk} public. *SP*, *BS* will also re-key the non-revoked *EC*s, and pass on their new keys to them via $\mathcal{S}$.

User u sends a registration request, $(user\_data, Cert_u)$ to $\mathcal{S}$. $user\_data$ contains information about the services $[ID_s]$ u is requesting, service provider ID_p, the service categories $[L_u]$, and expiry time of the user's subscription, T_{exp}. $\mathcal{S}$ forwards $(user\_data, Cert_u)$ to *SP*. *SP* creates a token $\mathcal{T}_{pu} = (ID_p, [ID_s], Cert_u, [L_u], T_{exp})$, and simulates a signature on $\mathcal{T}_{pu}$ via $\mathcal{S}_{\text{sig}}$. $\mathcal{S}$ then returns $(\mathcal{T}_{pu}, \sigma_{\mathcal{T}_{pu}}, M_{pk})$ to u. In the ideal world, $\mathcal{S}$ constructs tuple t_u = (register, $uid = H(\mathcal{T}_{pu})$, *scat*, *spid*) and sends to $\mathcal{F}_{\text{register}}$, where H is a collision-resistant hash function. $\mathcal{F}_{\text{register}}$ will add (*uid*, *scat*, *spid*) to its *uTable*. If $\mathcal{F}_{\text{register}}$ returns a $\perp$, $\mathcal{S}$ returns $\perp$ to u. When a user u needs to get revoked, *SP* will notify all *EC*s through $\mathcal{S}$ and $\mathcal{S}$ will forward $\mathcal{T}_{pu}$ to the *EC*s in the real-world. In the ideal-world, $\mathcal{S}$ will create and forward (revoke, $uid = H(\mathcal{T}_{pu})$, *scat*, *spid*) to $\mathcal{F}_{\text{revoke}}$. $\mathcal{F}_{\text{revoke}}$ will delete the corresponding entry from its *uTable*.

Any user u can send a service request to $\mathcal{S}$. There are three cases to consider: a revoked user sending a service request, an un-revoked user sending a service request and a revoked *EC* trying to process the request (and thus gain access to that user's private input data supplied with the request), and an un-revoked user sending a service request which is processed by an un-revoked *EC*. We discuss them below:

(1) **Case 1**: Revoked u sending a service request: u creates and sends a service request $((Req = (\mathcal{T}_{pu}, [ID]_s, C_1, C_2)), \sigma_{\mathcal{T}_{pu}})$. $\mathcal{S}$ needs to forward C_1, C_2 to the appropriate *EC*(*s*), since it cannot decrypt them itself. $\mathcal{S}$ first does $uid \leftarrow H(\mathcal{T}_{pu})$ and sends $(Req, \sigma_{\mathcal{T}_{pu}})$ to the *bsid* and *ecid*s associated with *spid*. $\mathcal{S}$ finds the appropriate *spid* by calling the Verify interface

of $\mathcal{S}_{\text{sig}}$ to verify the signature on $\sigma_{\mathcal{T}_{pu}}$ with the appropriate VK_{spid}. Honest *EC*s will return "error", if the user is revoked, or if the timestamp, T_{exp} is past its expiry date, while malicious *EC*s may still process the revoked user's request. In the ideal-world, $\mathcal{S}$ creates a tuple (*spid*, *sname*, *uid*, *bsid*, *udata*) and forwards to $\mathcal{F}_{\text{response}}$. Since u was revoked before sending the request, $\mathcal{F}_{\text{response}}$ will return $\perp$ (the check in Step 2 of $\mathcal{F}_{\text{response}}$ will fail). $\mathcal{S}$ then returns "error" to u. $\mathcal{S}$ will disregard any responses it receives from malicious *EC*s possibly colluding with the revoked user.

(2) **Case 2**: Revoked *EC* trying to process u's request: User u creates and sends a service request to $\mathcal{S}$ similar to Case 1, and $\mathcal{S}$ forwards the request to *bsid* and *ecid*s. In the real-world, each *BS* will revoke *EC*s on the request of the appropriate *SP* with whom *EC* is registered. When an *EC* gets revoked, *BS* will run Protocol 7, Steps 3-12, to re-issue new keys to the un-revoked *EC*s who possessed the same attributes as the *EC* getting revoked. This ensures that *BS* will not forward $(Req, \sigma_{\mathcal{T}_{pu}})$ to revoked *EC*s, nor will $\mathcal{S}$ accept any responses from them. In the ideal-world, when an *ecid* needs to get revoked, the corresponding *spid* with whom *ecid* is registered will send a (revoke, *spid*, *ecid*) message to $\mathcal{F}_{\text{revoke}}$. Upon receipt of this, $\mathcal{F}_{\text{revoke}}$ will promptly delete that *ecid*'s tuple from *sTable*, and send the tuple to $\mathcal{S}$ who will not forward $(Req, \sigma_{\mathcal{T}_{pu}})$ to the revoked *EC*s. Nevertheless, if it still receives responses from revoked *EC*, $\mathcal{S}$ will ignore them. The rest of the simulation proceeds similar to Case 1.

(3) **Case 3**: Un-revoked user sending a service request processed by an un-revoked *EC*: User u creates and sends a service request $((Req = (\mathcal{T}_{pu}, [ID]_s, C_1, C_2)), \sigma_{\mathcal{T}_{pu}})$ as in the previous two cases. $\mathcal{S}$ sends $(Req, \sigma_{\mathcal{T}_{pu}})$ to the *bsid* and *ecid*s associated with *spid*. The *ecid*s response is forwarded back to $\mathcal{S}$. If the request is for a service (dynamic request), i.e., $C_2 \neq \perp$, $\mathcal{S}$ will forward the request to *bsid*, and all *ecid*s. $\mathcal{S}$ will accept the first response it receives from an *ecid*. Since $\mathcal{S}$ forwards the request to all *EC*s, some might respond saying they cannot provide the requested service; $\mathcal{S}$ ignores such responses. Eventually, at least one *EC* will send a response of the form fulfillService$(\cdot) \rightarrow msg$, which $\mathcal{S}$ forwards to u. If the request is for data (static request), i.e., $C_2 = \perp$, $\mathcal{S}$ will forward the request to *bsid* and all *ecid*s and accept the first response it receives. It will receive a response of the form contentCacheLookup $\rightarrow msg$, which $\mathcal{S}$ forwards to u. In either case, if all *ecid*s respond with a $\perp$, $\mathcal{S}$ returns $\perp$ to u.

Game 3: In this game, $\mathcal{S}$ needs to simulate the honest parties' outputs to $\mathcal{A}$; $\mathcal{S}$ does not have access to honest parties' outputs as it did in Game 2. $\mathcal{S}$ needs to reflect the outputs and protocol outcomes of the ideal-world in the simulation of the real-world protocol and any attempt by $\mathcal{A}$ to cheat in the real-world has to result in the protocol aborting in the ideal-world. We now state and prove the following lemma:

Lemma 11.3. *For all PPT adversaries $\mathcal{A}$ and PPT environments $\mathcal{Z}$, there exists a simulator $\mathcal{S}$ such that*

$$\text{Exec}_{\text{Game2},\mathcal{Z}} \approx \text{Exec}_{\text{Game3},\mathcal{Z}}$$

$\mathcal{S}$ sets up the public parameters, $\left(M_{pk} = (\text{sysparam}, apk_1, \ldots, apk_n), ask_1, \ldots, ask_n\right) \leftarrow \text{ABE.Setup}(1^\lambda, n)$ and simulates *SP* by creating an ID_p in the real-world. Although this is done for every *SP*, for simplicity, we have represented only one *SP*. In the ideal-world, $\mathcal{S}$ creates *spid*, which is passed on to $\mathcal{F}_{\text{register}}$ to register *SP* as (register, *spid*, $\cdot$, $\cdot$, $\cdot$, $\cdot$).

For every *ecid* in the real-world that $\mathcal{A}$ wants to control, $\mathcal{A}$ sends to $\mathcal{S}$ a tuple $([ID_s], Cert_e)$. In response $\mathcal{S}$ simulates and sends $(e, [A_e])$ to $\mathcal{A}$, where each $[A_e] \leftarrow \text{ABE.KeyGen}(M_{pk}, ask_e, id_e, [ID_s])$. $\mathcal{S}$ also creates $(e, [A_e])$ for simulating honest *EC*s with service attributes not signed up for by the $\mathcal{A}$ in the previous step. $\mathcal{S}$ stores $(e, [A_e])$ for honest *EC*s locally. In the ideal-world, $\mathcal{S}$ sends (*spid*, *ecid*, $\cdot$) to $\mathcal{F}_{\text{register}}$. If $\mathcal{A}$ signals an *EC* be revoked, $\mathcal{S}$ generates the new M'_{pk}. For the un-revoked *EC*s, $\mathcal{S}$ sends their new $[A_e]'$ to $\mathcal{A}$. In the ideal-world, $\mathcal{S}$ sends (revoke, *spid*, *ecid*) to $\mathcal{F}_{\text{revoke}}$ who will delete all tuples of the form (*spid*, *ecid*, $\cdot$) from its *sTable*. If $\mathcal{A}$ tries to revoke a non-existent *EC*, $\mathcal{S}$ will forward (revoke, *spid*, $\cdot$), who will return $\perp$, which $\mathcal{S}$ returns to $\mathcal{A}$.

$\mathcal{A}$ sends registration requests on behalf of corrupted users to $\mathcal{S}$. For each user u's registration, $\mathcal{S}$ creates a token $\mathcal{T}_{pu} = (spid, sname, \cdot, scat, \cdot)$, and creates $\sigma_{\mathcal{T}_{pu}}$ by simulating $\mathcal{S}_{\text{sig}}$. It returns $(\mathcal{T}_{pu}, \sigma_{\mathcal{T}_{pu}}, M_{pk})$ to $\mathcal{A}$. If $\mathcal{A}$ sends a message for a user to get revoked (along with the corresponding *bsid*), $\mathcal{S}$ sends (revoke, *uid*, *spid*) to $\mathcal{F}_{\text{revoke}}$. If $\mathcal{F}_{\text{revoke}}$ returns $\perp$, i.e., $\mathcal{A}$ has tried to revoke a non-existent or an already-revoked user, $\mathcal{S}$ returns $\perp$ to $\mathcal{A}$. Else, $\mathcal{S}$ notifies $\mathcal{A}$ of the successful revocation.

$\mathcal{S}$ receives a service request from $\mathcal{A}$ of the form $(Req = (\mathcal{T}_{pu}, [ID]_s, C_1, C_2), \sigma_{\mathcal{T}_{pu}})$. $\mathcal{S}$ checks if the service request can be satisfied by one of the adversary controlled *EC*s; the request is handled locally by $\mathcal{A}$ and need not be simulated. However, if the request cannot be satisfied by an adversary controlled *EC*, $\mathcal{S}$ will utilize $\mathcal{F}_{\text{response}}$ functionality to respond to the user request.

In static requests, C_1 is plaintext, so that tells $\mathcal{S}$ what the *sname* is. $\mathcal{S}$ then calls $\mathcal{S}_{\text{sig}}$ to verify the signature on $\sigma_{\mathcal{T}_{pu}}$ with the appropriate VK_{spid}. This tells $\mathcal{S}$ what the *spid* is. Also, $H(\mathcal{T}_{pu}) \rightarrow uid$ which tells $\mathcal{S}$ what the *uid* is. When $\mathcal{A}$ sent $\mathcal{T}_{pu}$, it will tell $\mathcal{S}$ which *bsid* the request is intended for. So, $\mathcal{S}$ has all the information it needs to construct a tuple $\mathcal{F}_{\text{response}}$ (*spid*, *sname*, *uid*, *bsid*, C_2) and sends to $\mathcal{F}_{\text{response}}$. If the *EC* is not corrupted, i.e., $\mathcal{S}$ simulates the output of a honest *EC* by forwarding the output, $\{msg, \perp\}$, of $\mathcal{F}_{\text{response}}$ to $\mathcal{A}$. Note that if $\mathcal{F}_{\text{response}}$ returned a $\perp$ then that means $\mathcal{A}$ queried on behalf of a revoked user and/or a revoked *EC*. When $C_2 \neq \perp$ (dynamic requests), $\mathcal{S}$ looks at the set of attributes ID_s for the key-policy ABE that were used in the generation of C_1. This will tell it the *sname*s that $\mathcal{A}$ is requesting. $\mathcal{S}$ uses the locally stored keys ($[A_e]$) to decrypt C_1 to get symmetric key K, and uses K to decrypt C_2 to retrieve *udata*. $\mathcal{S}$ can then construct a tuple (*spid*, *sname*, *uid*, *bsid*, *udata*) to send to $\mathcal{F}_{\text{response}}$. The rest of the simulation proceeds as in the static case.

□