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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:


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, supports several practical, useful applications such as traffic control, autonomous driving, providing cognitive assistance to users, 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
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 hootstrapped 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 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.

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 \((k\text{-DDH})\) 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 \((i.e., \text{content})\) 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 \( u \in U \) interested in using a service provider’s \( p \in P \) services \((e.g., \text{Instagram})\) 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 \( e \in E \) 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.

**Definition 4.1. Authentication Token**

Token \( T_{pu} \) 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. \( T_{pu} \) is a unique token that includes the service provider’s unique identifier, \( ID_p \), the service identifier, \( ID_s \), \((\text{or a list of service identifiers})\), the user’s certificate, \( Cert_u \), the user’s authorization level, \( L_u \), \((\text{or a list of authorization levels})\), and its expiry time, \( T_{exp} \): \( T_{pu} := \{ID_p, [ID_s], Cert_u, [L_u], T_{exp}\} \).

The service provider’s identifier, \( ID_p \), in \( T_{pu} \) enables the access control enforcers, \( e.g., \text{PEC servers} \), 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 \( T_{pu} \), the PEC servers prevent \( u \)’s unauthorized access to other services \( p \) provides that requires independent membership per service. For each service \((\text{static or dynamic})\), \( L_u \) indicates \( u \)’s authorization level, \( i.e., \text{Bronze, Silver, or Gold} \), to be matched against the required authorization level of the requested service. Token \( T_{pu} \) 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, \( T_{pu} \) 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
discovery 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’s 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) \((\text{sysparam}, (\text{apk}_{1}, \text{ask}_{1}), \ldots, (\text{apk}_{n}, \text{ask}_{n})) \leftarrow \text{ABE.Setup}(1^{\lambda}, n)\): This algorithm runs once in the beginning to setup the system parameters and the AIAs. It takes in a security parameter, \( \lambda \), 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, \( M_{pk} = (\text{sysparam}, \text{apk}_{1} \ldots \text{apk}_{n}) \).

2) \(\mathbb{G}_{k}^{\mathbb{Z}_{q}} \leftarrow \text{ABE.KeyGen}(M_{pk}, \text{ask}_{k}, \text{id}_{k}, A_{k})\): This algorithm is run by AIA \( k \), and takes as input \( M_{pk} \), \( k \)’s secret key, \( \text{ask}_{k} \), a user id, \( \text{id}_{k} \), and a set of attributes, \( A_{k} \), s.t., \( |A_{k}| \geq d_{k} \). It outputs the user’s secret keys \( \mathbb{G}_{k}^{\mathbb{Z}_{q}} \).

3) \(C \leftarrow \text{ABE.Encrypt}(M_{pk}, (A_{1}, \ldots, A_{n}), m)\): 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) \(\{m, \bot\} \leftarrow \text{ABE.Decrypt}(M_{pk}, (\mathbb{G}_{1}, \ldots, \mathbb{G}_{n}), C)\): 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 \( \bot \). 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, \( C \), service providers, \( P \), PEC servers, \( E \), and users, \( U \). 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 (\( p \in P \)) must register with the Cloud (\( C \)) 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 (\( R_{AC} \)) to \( C \), 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 \( R_{AC} \) from \( p \), \( C \) 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 (\( e \in E \)) initiates its registration process by securely
serverTable

AIA stores the generated keys in a table (Line 1). Each AIA executes the MABE key generation algorithm (Line 2). The publicly known master public key $\text{pk}_{\text{MABE}}$ is used to securely send user credentials to the userTable (Line 3). The AIA stores the generated keys in a serverTable and securely sends them to the user (Lines 4-5), allowing it to decrypt any requests that it may receive from users, so long as the user’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 ($\text{Cert}_u$) and general user data ($\text{user\_data}$) that are often used for creating user accounts, such as credentials, e-mail, and birth-date (Line 1). The $\text{user\_data}$ 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 (LID1) corresponding to the $\text{user\_data}$. It should be noted that the LID1 corresponds to services offered by $p$. With the user’s information, $p$ generates a customized access token ($\text{TP}_u$) for $u$ containing $p$’s identifier (LID2), list of permitted service identifiers (LID3), the user’s certificate ($\text{Cert}_u$), $u$’s authorization levels for the permitted services (LID4), and an expiry time $\text{exp}$ (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 $\text{TP}_u$, its signature ($\sigma_{\text{TP}_u}$), 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 of $\sigma_{\text{TP}_u}$ and the $\sigma_{\text{TP}_u}$ helps to ensure authenticity and integrity of $u$’s token and prevents tampering with the token. Also, $\text{Cert}_u$ in the token indicates $u$’s ownership of the token. Additionally, the set of permitted service identifiers, LID1, 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, $\text{exp}$ 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 (LID2) 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 $C_2$ (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 tuple.

Protocol 4 User’s Service Request

\begin{verbatim}
1. if $ID_s$ is Static Service Request then
   1. $C_1 \leftarrow \text{content\_name}, C_2 \leftarrow \bot$
2. else if $ID_s$ is Dynamic Service Request then
   1. $C_1 \leftarrow \text{ABE. Encrypt}(M_{pk}, ID_s, K)$
   2. $C_2 \leftarrow \text{Enc}_K(D)$
3. end if
4. set Req = $\{\text{TP}_u, ID_s, C_1, C_2\}, \sigma_{\text{Req}} \leftarrow \text{Sign}_{SK_u}(\text{Req})$
5. send $\{\text{Req}, \sigma_{\text{Req}}\}$
\end{verbatim}
tuple, including the token $T_{pu}$, 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 $T_{pu}$ (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 $T_{pu}$ (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 $T_{pu}$'s owner. If successful, $e$ confirms the freshness of $T_{pu}$ by comparing its expiry time ($T_{exp}$) with the current time ($T_e$) (Lines 4-5). Subsequently, $e$ searches for $T_{pu}$ 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 $T_{pu}$ using her new certificate and established user credentials (refer to Protocol 1). Upon token validation, $e$ uses $ID_p$ that is contained within $T_{pu}$ 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 $T_{pu}$, dropping the connection if invalid (Lines 10-11). Successful verification validates $T_{pu}$'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 decrpyts $C_1$ using its decryption keys $[A_{\nu}]$ 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

```plaintext
1: receive $\{Req, \sigma_{Req}\}$
2: extract $\{T_{pu}, ID_s, C_1, \sigma_{C1}\} \leftarrow Req, \{ID_p, [ID_s], Cert_u, L_u, T_{exp}\} \leftarrow T_{pu}, VK_u \leftarrow Cert_u$
3: if $true \leftarrow Verify_{VK_u}(\{Req, \sigma_{Req}\})$ then
4: if $T_{exp} < T_e$ then
5: return error
6: else if $T_{pu} \in revocTable$ then
7: return error
8: end if
9: Cert$_p \leftarrow lookupProvider,ID_p, VK_p \leftarrow Cert_p$
10: if $false \leftarrow Verify_{VK_p}(\sigma_{T_{pu}}, T_{pu})$ 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 $true \leftarrow checkServerCapability(ID_s)$ then
19: $K \leftarrow ABE.Decrypt(M_{pk}, [A_{\nu}], C_1)$
20: $D \leftarrow Decrypt(D_{\nu})$
21: return fulfillService($D$)
22: else
23: requestService($Req, \sigma_{Req}$)
24: end if
25: else if $ID_s$ is Static Service Request then
26: if $true \leftarrow contentCacheLookup(C_1)$ then
27: return contentCacheRetrieve($C_1$)
28: else
29: 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 $T_{pu}$ 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 $T_{pu}$ 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 AIASs (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 AIA, enabling AIA (including the base stations) to run the MABE system setup for calculating $M'_{pk}$ (Line 3). The base station then generates and
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 (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 \( \pi \) and \( \phi \) be probabilistic polynomial-time (PPT) protocols. We say that \( \pi \) UC-emulates \( \phi \) if for any PPT adversary \( A \) there exists a PPT adversary \( S \) such that for any balanced PPT environment \( Z \) we have

\[
\text{EXEC}_\phi.S,Z \equiv \text{EXEC}_\pi.A,Z
\]

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

We define an ideal functionality, \( F_{\text{APECS}} \), consisting of five independent ideal functionalities, \( F_{\text{register}}, F_{\text{response}}, F_{\text{revoke}}, F_{\text{smrt}}, F_{\text{sig}} \). \( F_{\text{register}} \) models the user and edge servers’ registration processes, \( F_{\text{response}} \) models the processing of a user’s service request, \( F_{\text{revoke}} \) models the revocation functionality. We use two helper functionalities from [2], \( F_{\text{sig}} \) and \( F_{\text{smrt}} \), to model ideal functionalities for digital signatures and secure/authenticated channels, respectively. We assume that \( F_{\text{APECS}} \) maintains internal state that is accessible at any time to \( F_{\text{register}}, F_{\text{response}} \) and \( F_{\text{revoke}} \), specifically three tables, \( uTable, sTable \) and \( dTable \). The parties that interact with the ideal functionalities are the members of sets of edge servers, \( \mathbb{S} \), service providers, \( \mathbb{P} \), base stations \( \mathbb{B} \), 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 \( F_{\text{register}} \)

(1) When a service provider, \( SP \in SP \) sends a request, (register, spid, sname, scat, sdata, stype), \( F_{\text{register}} \) adds \( t_d = (spid, sname, scat, sdata, stype) \) to the \( dTable \); if \( t_d \) already exists, \( F_{\text{register}} \) returns \( t_d \). When an \( SP \) requests a service update (reject, spid, \( \ldots \)), \( F_{\text{register}} \) updates \( t_d \). When an \( SP \) sends a request (deregister, spid, sname, scat), \( F_{\text{register}} \) deletes the corresponding tuple \( t_d \) from \( dTable \).

(2) When a user \( u \) sends a registration request, (register, uid, scat, spid) to \( F_{\text{register}} \) (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), \( F_{\text{register}} \) checks if there exists a tuple \( t_u = (uid, scat, spid) \) in \( uTable \). If yes, \( F_{\text{register}} \) returns \( t_u \) to \( u \), and forwards (exists, \( t_u \)) to \( S \). Else \( F_{\text{register}} \) sends a message, (register, \( uid, scat \)) to \( SP \in SP \) whose identifier is \( spid \). If \( SP \) responds with an “allow”, \( F_{\text{register}} \) adds tuple \( t_u = (uid, scat, spid) \) to \( uTable \); returns “success” to \( u \), and forwards (success, \( t_u \)) to \( S \). Else \( F_{\text{register}} \) returns \( \bot \) to \( u \), and forwards (failReg, \( uid, scat, spid \)) to \( S \).

(b) When \( u \) sends a request update (register, \( uid, scat', spid \)), \( F_{\text{register}} \) retrieves a tuple \( t_u = (uid, scat, spid) \) in \( uTable \). \( F_{\text{register}} \) sends a message, (update, \( uid, scat' \)), to \( SP \). If \( SP \) replies with \( \bot \), \( F_{\text{register}} \) returns \( \bot \) to \( u \), and forwards (failUpdate, \( t_u \)) to \( S \). Else \( F_{\text{register}} \) updates or creates (if the retrieval of \( t_u \) returned \( \bot \)) a tuple \( t_u \) with \( (uid, scat', spid) \), and returns “success” to \( u \), and forwards (successUpdate, \( t_u \)) to \( S \). In case a new \( t_u \) was created, \( F_{\text{register}} \) forwards (newReg, \( t_u \)) to \( S \).

(c) If a user sends a request (deregister, \( uid, \ldots, spid \)), \( F_{\text{register}} \) deletes tuple \( t_u = (uid, \ldots, spid) \) in \( uTable \) and forwards (deregister, \( uid, \ldots, spid \)) to \( SP \), and \( S \).

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

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

Figure 2: Ideal functionality for Service Registration

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

\( F_{\text{register}} \): The \( F_{\text{register}} \) functionality shown in Figure 2 handles the system setup and registration/de-registration of a user \( u \) and members of \( BC \). This also handles registration of data associated with members of \( SP \), as well as service updates. When a service provider \( SP \) wishes to register, it initiates contact with \( F_{\text{register}} \) 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. \( F_{\text{register}} \) 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 \( F_{\text{register}} \) who deletes the tuple from \( dTable \).

When a user \( u \), identified by \( uid \), wants to register for a service, it contacts \( F_{\text{register}} \) 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, \( F_{\text{register}} \) adds \( u \)’s information to \( uTable \), and forwards the registration information to \( S \). Similarly, when an already-registered \( u \) wishes to update their service category to \( scat' \), \( F_{\text{register}} \) will check with \( SP \) and act accordingly. \( F_{\text{register}} \) will also notify \( S \) whether the update request was successful. When \( u \) terminates its service and de-registers from an \( spid \), \( F_{\text{register}} \) deletes the unique tuple (\( uid, \ldots, spid \)) without needing to ask \( SP \)’s permission, but informs \( SP \) and \( S \) 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, \( F_{\text{register}} \) will add \( EC \)’s information to \( sTable \), else it will notify \( EC \) and \( S \) that the registration request was denied. At this point, \( F_{\text{register}} \) will also send to \( EC \) and \( S \) 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. \( F_{\text{register}} \) deletes the unique tuple (\( spid, ecid, bsid \)) from \( sTable \), without needing to ask \( SP \)’s or \( BS \)’s permission, but informs them and \( S \) about it.

\( F_{\text{response}} \): The \( F_{\text{response}} \) 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 \( F_{\text{response}} \) 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 ecids 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 \( \bot \) if the request is a static request.

Once \( F_{\text{response}} \) receives the request from the user, it forwards (recvReq, \( spid, sname, uid, bsid, udata \)) to \( S \) 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 \( \bot \) is returned to the user along with (failReq, \( spid, sname, uid, bsid, udata \)) to \( S \), otherwise \( F_{\text{response}} \) continues to the next step. Next, \( F_{\text{response}} \) 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 $\langle \text{spid}, \text{sname}, \text{uid}, \text{bsid}, \text{udata} \rangle$ from a user, $\mathcal{F}_{\text{response}}$ retrieves the data tuple, $t_d = \langle \text{spid}, \text{sname}, \text{scat}, \text{sdata}, \text{stype} \rangle$ containing spid and sname from user’s request and sends (reqvReq, spid, sname, uid, bsid, udata) to S. If $t_d$ does not exist return ⊥ to user and send (failReq, spid, sname, uid, bsid, udata) to S.

(2) Then $\mathcal{F}_{\text{response}}$ checks the uTable and retrieves user tuple $t_u = (\text{uid}, \text{scat}, \text{spid})$ 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 ⊥ to user and send (failReq, spid, sname, uid, bsid, udata) to S.

(3) $\mathcal{F}_{\text{response}}$ then retrieves all tuples matching $t_s = \langle \text{spid}, \cdot, \text{bsid} \rangle$ in sTable where spid and bsid in $t_s$ is the same as that in the user’s request. If no such tuples exist return ⊥ to user and send (failReq, spid, sname, uid, bsid, udata) to S.

(4) If all previous verifications pass, then $\mathcal{F}_{\text{response}}$ forwards request $\langle \text{uid}, \text{sname}, \text{udata} \rangle$ to all edge server ecids in the $t_s$ tuples that was retrieved in the previous step and forward (ReqEC, uid, sname, udata) to 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) → 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 S, and discards all other following msgs.

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 ⊥ is returned to the user and $\mathcal{F}_{\text{response}}$ sends (failReq, spid, sname, uid, bsid, udata) to S. $\mathcal{F}_{\text{response}}$ then retrieves all tuples $t_s = \langle \text{spid}, \cdot, \text{bsid} \rangle$ to identify all ecids that can process the user’s request. If no such tuple exists, then this indicates that there are no ecids connected to bsid that can process the user’s request and provide services on behalf of spid. $\mathcal{F}_{\text{response}}$ returns ⊥ to the user along with (failReq, spid, sname, uid, bsid, udata) to S.

If all the above checks succeed, then $\mathcal{F}_{\text{response}}$ has a list of all ecids 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 ecids in the tuples $t_s$, retrieved in the previous step and forwards (ReqEC, uid, sname, udata) to 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 ecid retrieves the algorithms, sdata associated with sname, processes udata, sdata(udata) → 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 msgs from other ecids. $\mathcal{F}_{\text{response}}$ also forwards msg to 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 $t_s = \langle \text{spid}, \text{ecid}, \cdot \rangle$. If any exist, $\mathcal{F}_{\text{revoke}}$ deletes the tuples from sTable and forwards (revoke, spid, ecid) to all bsids in the deleted tuples. Else returns ⊥ 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 $t_u = \langle \text{uid}, \cdot, \text{spid} \rangle$. If any exist, it deletes the tuples from uTable and returns "success". Else, it returns ⊥.

Figure 4: Ideal functionality for User/Edge Server Revocation

$\mathcal{F}_{\text{revoke}}$: 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 $A$ be a probabilistic polynomial-time (PPT) adversary for APECS, and let $S$ 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
Thus, we do not consider such DoS attacks a common threat. 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 (e) in the threat model). 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.

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
We benchmarked the performance of MABE using the PBC library, AIA’s identity and service type. For consistency, in AIAs to represent a more complicated scenario for studying services classes mentioned above, namely Compact Edge, Handheld, Cloud. 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 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. 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 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. 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

<table>
<thead>
<tr>
<th>Operations</th>
<th>APECS</th>
<th>APECS PKC</th>
<th>Cloud</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service Discovery</td>
<td>69.5</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Symmetric Encryption</td>
<td>0.7</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ABE Encryption</td>
<td>39.0</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Request Signing</td>
<td>39.6</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Network/System Latency</td>
<td>5.8</td>
<td>252.0</td>
<td>79.5</td>
</tr>
<tr>
<td>User Token Verification</td>
<td>4.8</td>
<td>5.8</td>
<td>9.6</td>
</tr>
<tr>
<td>Edge Token Verification</td>
<td>31.4</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Request Signature Verification</td>
<td>2.7</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ABE Decryption</td>
<td>33.8</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Symmetric Encryption</td>
<td>0.5</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Total</td>
<td>122.7</td>
<td>186.1</td>
<td>261.6</td>
</tr>
</tbody>
</table>

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

![Figure 8: Access control enforcement throughput for static and dynamic service requests on two platforms.](image)

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 $O(|P + B|)$ 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 $O(|B|)$ 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 $O(|P + B|)$ 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-E0008774, 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


10 COMPUTATIONAL ASSUMPTIONS

Let $G_1$ and $G_2$ be two cyclic multiplicative groups of prime order $q$ generated by $g_1$ and $g_2$ respectively, $a \in G_1 \times G_2 \rightarrow \mathbb{G}_T$ be a bilinear map such that $V_x \in G_1$, $y \in G_2$, 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 $\mathbb{G} < \mathbb{G}_T$ 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 $BDH_{Gen}^{(1)}$ output the parameters $(\hat{e}, \hat{e}^{-1}, q, g_1, g_2, G_1, G_2, \mathbb{G}_T)$ where there is an efficiently computable isomorphism $\psi$ from $G_2$ to $G_1$. The Decisional Bilinear Diffie-Hellman (DBDH) problem is defined as follows: given $g_1 \in G_1, g_2, g_2^a, g_2^b, g_2^c \in G_2$ and $Z \in \mathbb{G}_T$ as input, decide if $Z = \hat{e}(g_1, g_2)^{abc}$ or $\hat{e}(g_1, g_2)^{ab}$ for $R \in \mathbb{Z}_q$.

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

Definition 10.4. Let $BDH_{Gen}^{(1)}$ output the parameters for a bilinear mapping $(\hat{e}) : G_1 \times G_2 \rightarrow \mathbb{G}_T$. The external Diffie-Hellman (XDH) assumption states that, for all probabilistic polynomial time adversaries $A$, the DDH problem is hard in $G_1$. This implies that there does not exist an efficiently computable isomorphism $\psi : G_1 \rightarrow G_2$.

11 UC SECURITY ANALYSIS

11.1 Discussion

The security properties $AP_nC_n$ 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.

\( F_{\text{register}} \) 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, \( F_{\text{register}} \) 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, \( uTable \), and cannot be modified by users and/or service providers. When a user is revoked by an SP, \( F_{\text{revolve}} \) promptly deletes the corresponding entry from \( uTable \). When a user tries to request services of an SP in a category it has not registered for, e.g., a bronze member requesting gold member services, \( F_{\text{response}} \) will check the \( uTable \) and return \( \bot \) 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 \( \text{snname} / \text{scat} \) offered by different ECs, but we do not depict them here for presen-
tational clarity. An EC, based on \( \text{type} \), 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 \( \text{sdata} \) field of the corresponding \( \text{snname} \) on the user’s input, \( udata \), and returns the result to \( F_{\text{response}} \), 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, \( F_{\text{response}} \) 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 \( Z \).

**Game 0:** This is the same as the real-world APECS. \( Z \) interacts directly with APECS and \( A \).

**Game 1:** \( S \) internally runs \( A \) and simulates the secure and authenticated channels functionality \( F_{\text{smi}} \).

**Lemma 11.1.** For all PPT adversaries \( A \) and PPT environments \( Z \), there exists a simulator \( S \) such that

\[
\text{Exec}_{\text{Game}1}, Z = \text{Exec}_{\text{Game}2}, Z
\]

The two games are trivially indistinguishable since \( S \) just executes the simulator for \( F_{\text{smi}} \).

**Game 2:** \( S \) communicates with the honest parties and \( A \), and simulates the protocols of APECS with the help of \( F_{\text{APECS}} \). \( A \) 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 \( A \) and all further communication to and from the corrupted party is routed through \( A \). We now state and prove the following lemma:

**Lemma 11.2.** For all PPT adversaries \( A \) and PPT environments \( Z \), there exists a simulator \( S \) such that

\[
\text{Exec}_{\text{Game}1}, Z \approx \text{Exec}_{\text{Game}2}, Z
\]

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

User u sends a registration request, \( (\text{user}_data, Cert_{tu}) \) to 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, \( T_{\text{exp}} \). \( S \) forwards \( (\text{user}_data, Cert_{tu}) \) to \( SP \). \( SP \) creates a token \( \text{pu} = (ID_p, \{ID_s\}, Cert_{tu}, \{L_u\}, T_{\text{exp}}) \), and simulates a signature on \( \text{pu} \) via \( S_{\text{sig}} \). Then returns \( (\text{pu}, \sigma_{\text{pu}}, M_{pk}) \) to u. In the ideal world, \( S \) constructs tuple \( t_u = (\text{register}, uid = H(\text{pu}), \text{scat}, \text{spid}) \) and sends to \( F_{\text{register}} \), where \( H \) is a collision-resistant hash function. \( F_{\text{register}} \) will add \( (uid, \text{scat}, \text{spid}) \) to its \( uTable \). If \( F_{\text{register}} \) returns \( \bot \), then \( S \) returns \( \bot \) to u. When a user u needs to get revoked, \( SP \) will notify all ECs through \( S \) and \( S \) will forward \( \text{pu} \) to the ECs in the real-world. In the ideal-world, \( S \) will create and forward (revoke, \( uid = H(\text{pu}), \text{scat}, \text{spid} \)) to \( F_{\text{revolve}} \). \( F_{\text{revolve}} \) will delete the corresponding entry from its \( uTable \).

Any user u can send a service request to 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 \( (\text{Req} = (\text{pu}, \{ID_s, C_1, C_2\}), \sigma_{\text{pu}}) \). S needs to forward \( C_1, C_2 \) to the appropriate ECs(s), since it cannot decrypt them itself. S first does \( uid \leftarrow H(\text{pu}) \) and sends \( (\text{Req}, \sigma_{\text{pu}}) \) to the \( bsid \) and \( ecids \) associated with \( spid \). S finds the appropriate \( spid \) by calling the Verify interface
of $S_{\text{sig}}$ to verify the signature on $\sigma_{\text{pu}}$ with the appropriate $VK_{\text{spid}}$. Honest ECs will return "error", if the user is revoked, or if the timestamp, $T_{\text{exp}}$ is past its expiry date, while malicious ECs may still process the revoked user's request. In the ideal-world, $S$ creates a tuple $(\text{spid}, \text{name}, \text{uid}, \text{bsid}, \text{udata})$ and forwards to $F_{\text{Response}}$. Since $u$ was revoked before sending the request, $F_{\text{Response}}$ will return ⊥ (the check in Step 2 of $F_{\text{Response}}$ will fail). $S$ then returns "error" to $u$. $S$ will disregard any responses it receives from malicious ECs possibly colluding with the revoked user.

(2) **Case 2:** Revoked EC trying to process user $u$'s request. User $u$ creates and sends a service request to $S$ similar to Case 1, and $S$ forwards the request to $\text{bsid}$ and $\text{ecids}$. In the real-world, each $BS$ will revoke ECs 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 ECs who possessed the same attributes as the $EC$ getting revoked. This ensures that $BS$ will not forward $(\text{Req}, \sigma_{\text{pu}})$ to revoked ECs, nor will $S$ 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, $\text{spid}$, $\text{ecid}$) message to $F_{\text{Revoke}}$. Upon receipt of this, $F_{\text{Revoke}}$ will prompt $S$ to delete $ecid$'s tuple from $\text{stable}$, and send the tuple to $S$ who will not forward $(\text{Req}, \sigma_{\text{pu}})$ to the revoked ECs. Nevertheless, if it still receives responses from revoked EC, $S$ 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 $(\langle \text{Req} = (\text{spid}, [\text{ID}_s], C_1, C_2) \rangle, \sigma_{\text{pu}})$ as in the previous two cases. $S$ sends $(\text{Req}, \sigma_{\text{pu}})$ to the $\text{bsid}$ and $\text{ecids}$ associated with $\text{spid}$. The $\text{ecids}$ response is forwarded back to $S$. If the request is for a service (dynamic request), i.e., $C_2 \neq \bot$, $S$ will forward the request to $\text{bsid}$, and all $\text{ecids}$. $S$ will accept the first response it receives from an $ecid$. Since $S$ forwards the request to all ECs, some might respond saying they cannot provide the requested service; $S$ ignores such responses. Eventually, at least one $EC$ will send a response of the form $\text{fulfillService}(\cdot) \rightarrow \text{msg}$, which $S$ forwards to $u$. If the request is for data (static request), i.e., $C_2 = \bot$, $S$ will forward the request to $\text{bsid}$ and all $\text{ecids}$ and accept the first response it receives. It will receive a response of the form $\text{contentCacheLookup} \rightarrow \text{msg}$, which $S$ forwards to $u$.

In either case, if all $\text{ecids}$ respond with $\bot$, $S$ returns $\bot$ to $u$.

**Game 3:** In this game, $S$ needs to simulate the honest parties' outputs to $A$, $S$ does not have access to honest parties' outputs as it did in Game 2. $S$ needs to reflect the outputs and protocol outcomes of the ideal-world in the simulation of the real-world protocol and any attempt by $A$ 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 $A$ and PPT environments $Z$, there exists a simulator $S$ such that

$$\text{Exec}_{\text{Game2}, Z} \equiv \text{Exec}_{\text{Game3}, Z}$$