When Query Authentication Meets Fine-Grained Access Control: A Zero-Knowledge Approach

Cheng Xu
Hong Kong Baptist University
chengxu@comp.hkbu.edu.hk

Jianliang Xu
Hong Kong Baptist University
xujl@comp.hkbu.edu.hk

Haibo Hu
Hong Kong Polytechnic University
haibo.hu@polyu.edu.hk

Man Ho Au
Hong Kong Polytechnic University
csallen@comp.polyu.edu.hk

ABSTRACT

Query authentication has been extensively studied to ensure the integrity of query results for outsourced databases, which are often not fully trusted. However, access control, another important security concern, is largely ignored by existing works. Notably, recent breakthroughs in cryptography have enabled fine-grained access control over outsourced data. In this paper, we take the first step toward studying the problem of authenticating relational queries with fine-grained access control. The key challenge is how to protect information confidentiality during query authentication, which is essential to many critical applications. To address this challenge, we propose a novel access-policy-preserving (APP) signature as the primitive authenticated data structure. A useful property of the APP signature is that it can be used to derive customized signatures for unauthorized users to prove the inaccessibility while achieving the zero-knowledge confidentiality. We also propose a grid-index-based tree structure that can aggregate APP signatures for efficient range and join query authentication. In addition to this, a number of optimization techniques are proposed to further improve the authentication performance. Security analysis and performance evaluation show that the proposed solutions and techniques are robust and efficient under various system settings.

KEYWORDS

Query processing, data integrity, fine-grained access control

ACM Reference Format:
Cheng Xu, Jianliang Xu, Haibo Hu, and Man Ho Au. 2018. When Query Authentication Meets Fine-Grained Access Control: A Zero-Knowledge Approach. In *Proceedings of 2018 International Conference on Management of Data (SIGMOD'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3183741

1 INTRODUCTION

With the prosperity of data-as-a-service (DaaS) and cloud computing, more and more enterprises are outsourcing their databases to off-the-shelf cloud data engines, such as Amazon SimpleDB, Microsoft Azure Cloud SQL, Salesforce Cloud Database, and Google BigQuery. While these cloud engines address the needs of enterprise databases for high performance, high availability, and low operation costs, enterprises are at the risk of losing control of query integrity. Due to service outages, program glitches, security vulnerabilities, and other reasons, enterprises cannot guarantee the correctness of query results returned by an outsourced database. To address this issue, authenticated query processing has been proposed and studied by a large body of literature [14, 15, 24, 34, 36, 38]. It relies on the data owner (i.e., the enterprise) to sign a well-designed authenticated data structure (ADS), based on which the outsourced database can construct a cryptographic proof for the results of each online query.

However, existing works do not fully address another important security concern that usually coexists with integrity, i.e., access control. Access control ensures each user can access only the data he/she is authorized to use. With the increasing popularity of moving enterprise databases into the cloud, the need for access control in data sharing is becoming more indispensable than ever. For example, to support cloud-based ERP and OLAP systems, a Salesforce cloud database can be configured to support up to 10,000 access control polices [27]. To this end, attribute-based encryption (ABE) [2, 9] has been a prevailing technique used by cloud databases to support *fine-grained* access control [10, 29–31]. In essence, each data record is specified with an access policy based on attributes (e.g., post title and/or specialty), rather than identities, so that only the users who possess authorized attributes can access the record. For example, a patient may authorize the access of his/her medical record only to senior researchers or doctors specializing in cancer.

Unfortunately, only very few studies have explored query authentication for databases where access control is enforced [3, 12, 22] and they suffer from three major limitations. First, they only support simple access control rules at a coarse-grained level. For example, [22] simply does not allow disclosure of data that are outside the query range; [3, 12] divide the database space into sub-spaces and enforce access control for each sub-space, based on identities. Second, unlike fine-grained access control, the access control in these existing studies is not cryptographically enforced, which renders the system susceptible to bypasses and SQL injection attacks [29]. Third, the existing authentication techniques distinguish between inaccessible data and non-existent data and, hence, could leak sensitive information beyond what can be deduced from accessible data. For example, suppose a doctor is authorized to access medical records associated with some specific disease only; by knowing the existence of the inaccessible records, he/she may infer the distribution of other diseases in the database.

To overcome these limitations, in this paper, we take the first step toward studying the problem of authenticating relational queries with fine-grained access control. To answer a query, the outsourced

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183741

database returns those records that not only satisfy the query condition but are also accessible to the user. As with existing query authentication techniques, a cryptographic proof is returned, along with the query results. The key challenge lies in how to protect information confidentiality during query authentication. That is, the proof must be *zero-knowledge*, so that it reveals nothing beyond the accessible records. For example, if no record is returned for a given search key, the user cannot deduce from the proof whether there does not exist a matching record or the matching record is inaccessible to him/her. This is essential in preventing enumeration attacks that exhaustively and iteratively issue queries of overlapping ranges to learn the distribution of search keys in the database. It is also useful to prevent linkage attacks that use such auxiliary information to compromise privacy in associated databases [7].

As a building block, we first propose a novel access-policy-preserving (APP) signature based on a variant of the attribute-based signature (ABS) scheme [19]. The main novelty of the APP signature is its dual roles: (i) for authorized users, it provides a proof of integrity of the data record; (ii) for unauthorized users, it can be tailored to derive customized signatures to prove the inaccessibility while achieving the zero-knowledge confidentiality. We then design the authenticated data structure (ADS) that consists of APP signatures of data records, based on which authenticated query processing algorithms are developed. To address performance issues for range queries and join queries, we further propose AP^{2}G-tree, a grid-index-based ADS that can aggregate APP signatures. Several optimization techniques that are compatible with the original security model or a relaxed one are also developed. To summarize, our contributions made in this paper are as follows:

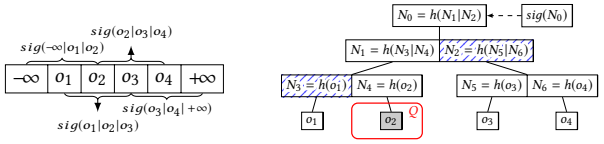
- To the best of our knowledge, this is the first work on query authentication for databases with fine-grained access control. We believe that this is a timely study, as enterprise cloud database systems dictate integrity (query authentication) and authorization (access control) at the same time.
- We propose a novel ABS-based APP signature as the primitive for ADS, together with a grid-index-based tree structure that can aggregate APP signatures for efficient range and join query authentication.
- We develop several optimization techniques that are either compatible with the original security model or a relaxed one.
- We conduct a security analysis and empirical study on the authentication performance with respect to various factors such as query range size and database cardinality.

The rest of the paper is organized as follows. Section 2 reviews existing studies on access control and query authentication. Section 3 introduces the formal problem definition, followed by cryptographic primitives in Section 4. Section 5 presents our solution for equality queries, which is then generalized to range queries and join queries in Section 6 with the design of AP^{2}G-tree. A security analysis is presented in Section 7, and Sections 8 and 9 present optimization techniques for the zero-knowledge model and a relaxed model, respectively. Section 10 presents the experimental results, followed by a conclusion in Section 11.

2 RELATED WORK

To the best of our knowledge, no existing solution supports zero-knowledge query authentication with fine-grained access control. In this section, we briefly review the relevant techniques.

Access Control. Enforcing access control in file systems or database systems has been widely studied in the literature. Traditionally, this is done by employing an *access control list* (ACL), in which a permission manifest is attached to each individual file or data record. However, this method suffers from poor scalability when dealing with massive data or complex access control requirements. To remedy such issues, *role-based access control* (RBAC) is proposed by Sandhu *et al.* [28]. RBAC implements an access control mechanism over role permissions, user-role, and role-role relationships. This makes it a flexible technology in supporting both *discretionary access control* (DAC) and *mandatory access control* (MAC). However, only static and pre-defined access policies can be supported with RBAC. To overcome this limitation and support dynamic, context-aware, and fine-grained access control, *fuzzy identity-based encryption* (Fuzzy IBE) [26] and, later, *attribute-based encryption* (ABE) [9] are developed. These approaches define an access policy with a complex boolean function over many different attributes to support *attribute-based access control* (ABAC) [25]. There are two categories of ABEs. In *key-policy ABE* (KP-ABE) [9], each data object is associated with a set of attributes, while users' decryption keys define the access policies with a boolean function over those attributes. In *ciphertext-policy ABE* (CP-ABE) [2], the access policy is embedded in each data object's ciphertext, while users are given a set of attributes as private keys to present their roles. ABE offers an effective way to enforce fine-grained access control over encrypted data, but it cannot be used to authenticate one's identity. To address this, *attribute-based signature* (ABS) [16, 19] is proposed as a signature scheme to prove one's identity that satisfies certain fine-grained constraints. While ABAC was originally designed for file systems, it has been recently adopted by various cloud databases to support fine-grained access control [10, 29–31].



(a) Signature Chaining **(b) Merkle Hash Tree**

Figure 1: Basic Authentication Techniques

Authenticated Query Processing. A large body of research on authenticated query processing has been carried out to verify the integrity of query results against an untrusted service provider [14, 15, 17, 24, 33, 34, 36, 38]. There are two basic techniques that can be used to achieve this: signature chaining (Figure 1a) and Merkle hash tree (Figure 1b). The former technique uses a public-key cipher with which the data owner can generate digital signatures for each data value using a private key. Consequently, the user can verify the authenticity of the data value using the public key. To establish the completeness of the query results, chaining signatures are produced to capture the correlation of each value and its adjacent values [24]. Merkle hash trees (MHTs), on the other hand, are built on index trees [20]. Each entry in a leaf node is assigned a digest based on its hashed data value, and each entry in an internal node is assigned a digest derived from its child nodes. The data owner signs the root digest of the MHT, which can be used to verify any subset of data values. For example, in Figure 1b, a range query Q will return $\{o_2, N_2, N_3\}$, based on which the user can reconstruct the root digest $N_0 = h(h(N_3|h(o_2))|N_2)$ for verification. MHT has been widely adapted to various index structures. Typical examples include the Merkle B-tree for relational data [15], the Merkle R-tree for spatial data [36, 37], the authenticated inverted index for text

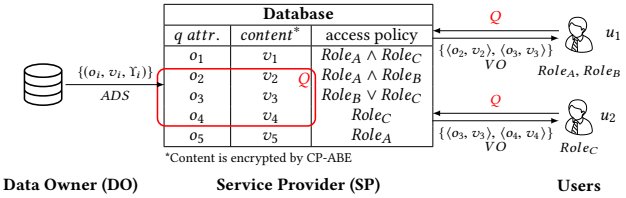


Figure 2: Query Authentication with Access Control

data [23], the authenticated prefix tree for multi-source data [5], and the MMB^{cloud}-tree for cloud service selection [17]. It has also been adopted to support authenticated join queries [35].

More recently, some studies have been conducted to unify query authentication with a privacy-preserving requirement. That is, the proof of the query integrity should not disclose any confidential information from the dataset. Notably, privacy-preserving query authentication for location-based range queries and top-k queries has been studied in [4, 11], in which new techniques are proposed to verify the comparison results in regard to private values. To protect all information of the dataset other than the query itself, Ghosh *et al.* propose a novel approach to achieving zero-knowledge authentication for range and closest point queries on one-dimensional data [8]. Unfortunately, it considers a case in which the user only submits a single query; continuous queries from the adversary would largely undermine its zero-knowledge privacy assurance. On the other hand, there are only a few query authentication studies that support access control. For keyword searches, Sun *et al.* develop a scheme to support verifiable searches with fine-grained access control [32]. However, this scheme only targets file-level searches. For relational databases, Chen *et al.* propose a verifiable range query algorithm with basic access control, which is built upon the concept of accessible space [3]. Assuming a similar coarse-grained access control model, Jain and Prabhakar further consider data privacy in query authentication and access control [12]. To date, how to effectively support query authentication with fine-grained access control in relational databases remains an open research question.

3 PROBLEM DEFINITION

In this paper, we study the problem of authenticated query processing on relational data with fine-grained access control. As shown in Figure 2, there are three parties in the system: (i) *data owner* (DO); (ii) *service provider* (SP); and (iii) *users*. The relational database $\mathcal{D}$, owned by the DO, is enforced with access control. Each record in $\mathcal{D}$ is a tuple $\langle o_i, v_i, \Upsilon_i \rangle$, where o_i is the query attribute, v_i is the content attribute, and Υ_i is the access policy. We assume that o_i is discrete and distinct among all records.[1] We further assume that v_i is encrypted by CP-ABE [2] (details in Section 4) for access control.[2] All the records are signed by the DO using an *authenticated data structure* (ADS) and are outsourced to the SP, who answers queries from the users on the behalf of the DO.

System Setup. During the system setup, the DO generates a public key pk_{user} (CP-ABE encryption key) and a secret key sk_{DO} (ABS signing key), which can be used to encrypt the data and generate the ADS, respectively. The DO also distributes a secret key sk_{user} (CP-ABE decryption key) and a public key pk_{DO} (ABS verification key) to the users, who can use them to decrypt and verify the query results, respectively.

Access Policy. Access policies control the results of a query with respect to the roles of the user.[3] For example, given the same range query Q in Figure 2, the result set to user u_1 with $Role_A$ and $Role_B$ is $\{\langle o_2, v_2 \rangle, \langle o_3, v_3 \rangle\}$ and that to user u_2 with $Role_C$ is $\{\langle o_3, v_3 \rangle, \langle o_4, v_4 \rangle\}$. In general, an access policy can be viewed as a boolean function in the form of $\Upsilon : \{0,1\}^n \rightarrow \{0,1\}$. It accepts multiple boolean inputs, each corresponding to a role, and outputs 1 or 0. For example, if Υ is defined as $(Role_A \wedge Role_C) \vee Role_B$, then $\Upsilon(Role_A) = 0$ and $\Upsilon(Role_B, Role_C) = 1$. Here, for clarity, we omit those inputs with zero values (e.g., a singleton $Role_A$ implies that $Role_B$ and $Role_C$ are missing). While, technically, an access policy can take any boolean function, in the literature, most works focus on a specific type, namely, monotone boolean functions [2, 9, 16, 19]. Formally, the monotonic property means that, for all a_i and b_i in $\{0,1\}$, if $a_1 \leq b_1, \ldots, a_n \leq b_n$, then $\Upsilon(a_1, \ldots, a_n) \leq \Upsilon(b_1, \ldots, b_n)$ holds. This property suggests that a user with more roles has a higher clearance level and so can access more data. In this paper, we assume that all access control policies are monotone boolean functions that are normalized in a *disjunctive normal form* (DNF).

Threat Model. Two security threats are considered in this paper: (i) the SP is not fully trusted and it might return tampered or incomplete query results; and (ii) the users are curious and may want to learn the information about the data to which access is unauthorized.

To address the first threat, we advocate authenticated query processing that enables the SP to prove the correctness of the query results [15, 24, 36]. Specifically, during query processing, the SP traverses the ADS and constructs a *verification object* (VO) that includes the verification information of the results. The VO is returned to the user along with the results. Using the VO, the user should establish the *soundness* and *completeness* of a result set RS:

- **Soundness**. No records in RS are tampered with and are truly the results with respect to the roles of the user.
- **Completeness**. All records not in RS are either non-results or are inaccessible to the user.

Take the range query Q in Figure 2 as an example. $RS = \{\langle o_2, v_2 \rangle, \langle o_3, v_3 \rangle\}$ is the correct result set for user u_1. $RS_1 = \{\langle o_2, v_2' \rangle, \langle o_3, v_3 \rangle\}$ and $RS_2 = \{\langle o_2, v_2 \rangle, \langle o_3, v_3 \rangle, \langle o_4, v_4 \rangle\}$ are not sound because they either contain fake (v_2' in RS_1) or inaccessible (o_4 in RS_2) records. $RS_3 = \{\langle o_2, v_2 \rangle\}$ is not complete because o_3 is missing.

To further address the second threat, we need to protect data access against unauthorized users during query authentication. We define three levels of confidentiality requirements:

- **Data Content Confidentiality**. The contents of inaccessible records are protected. This is a basic requirement for any query service that supports access control.
- **Access Policy Confidentiality**. In addition to data content confidentiality, the access policies of inaccessible records are protected. That is, the users must not know that their denied access is due to the lack of some specific role(s).
- **Zero-Knowledge Confidentiality**. Any information beyond the accessible records $\mathcal{D}^+$ in the database $\mathcal{D}$ is protected. That is, the users can gain nothing about $\mathcal{D} \backslash \mathcal{D}^+$ (e.g., not even its size or the associated access policies).

The above security notions will be formalized when we perform

[1] The proposed approach is also applicable to continuous data which can be converted into discrete data by discretization techniques [13]. How to handle duplicate records will be discussed in Appendix E.

[2] For simplicity, we assume that the query attribute is in plaintext. In cases where the query attribute is also encrypted, privacy-preserving query techniques [30] can be applied to retrieve query results, which is an orthogonal issue to query authentication.

[3] We use "roles" and "attributes" interchangeably in the context of ABE and ABS.

security analysis in Section 7.2. We assume that there is no collusion between the SP and users.

In what follows, we propose a zero-knowledge query authentication solution that addresses both of the security threats under fine-grained access control. We will develop new query authentication techniques for ADS generation, VO construction, and result verification that achieve the zero-knowledge confidentiality. To cater for performance-centric applications, we will also discuss how to further improve the query authentication performance by relaxing the confidentiality requirement.

4 PRELIMINARIES

This section gives some preliminaries on cryptographic constructs and integrity assurance.

Cryptographic Hash Function. A cryptographic hash function $hash(\cdot)$ accepts an arbitrary-length string as its input and returns a fixed-length bit string. It is collision resistant; it is difficult to find two different messages, m_1 and m_2, such that $hash(m_1) = hash(m_2)$. Classic cryptographic hash functions include the SHA-1, SHA-2, and SHA-3 family.

Bilinear Pairing. Bilinear pairing maps a pair of elements in two groups to a single element in a third group, which serves as a basic operation in *attribute-based encryption* (ABE) and *attribute-based signature* (ABS), as shown later in this paper.

Let $\mathbb{G}$, $\mathbb{H}$, and $\mathbb{G}_T$ be three cyclic multiplicative groups with the same order p, where p is a prime. Let g, h be the generator of $\mathbb{G}$ and $\mathbb{H}$, respectively. We can find a bilinear mapping $e : \mathbb{G} \times \mathbb{H} \rightarrow \mathbb{G}_T$, which has the following properties:

- Bilinearity: If $u \in \mathbb{G}$, $v \in \mathbb{H}$, and $e(u, v) \in \mathbb{G}_T$, then $e(u^a, v^b) = e(u, v)^{ab}$ for any u, v.
- Non-degeneracy: $e(g, h) \neq 1$.

Ciphertext-Policy Attribute-Based Encryption (CP-ABE). CP-ABE is proposed to realize complex access control over encrypted data [2]. By embedding the access policy into the ciphertext, it enables fine-grained access control represented by a boolean function of attributes. It consists of a set of algorithms:

- CP-ABE.Setup$(1^\lambda) \rightarrow (mk, pp)$: On input a security parameter 1^λ, it generates a master private key mk and a public key pp.
- CP-ABE.KeyGen$(mk, \mathcal{A}) \rightarrow sk_{\mathcal{A}}$: On input a master private key mk and an attribute set $\mathcal{A}$, it outputs a decryption key $sk_{\mathcal{A}}$.
- CP-ABE.Encrypt$(pp, x, \Upsilon) \rightarrow c_\Upsilon$: On input a public key pp and an access policy Υ, it encrypts plaintext x into ciphertext c_Υ.
- CP-ABE.Decrypt$(sk_{\mathcal{A}}, c_\Upsilon) \rightarrow x$: On input a decryption key $sk_{\mathcal{A}}$, ciphertext c_Υ, it outputs plaintext x if $\Upsilon(\mathcal{A}) = 1$; otherwise $\perp$ is returned.

Attribute-Based Signature (ABS). Introduced in [19], ABS is a signature scheme that enables a party to sign a message with fine-grained access control over the identifying information. Unlike a traditional signature scheme, messages can be signed with a monotone boolean function predicate that is satisfied by the attributes obtained from the authority. It consists of the following algorithms:

- ABS.Setup$(1^\lambda) \rightarrow (msk, mvk)$: On input a security parameter 1^λ, it generates a master signing key msk and a master verification key mvk.
- ABS.KeyGen$(msk, \mathcal{A}) \rightarrow sk_{\mathcal{A}}$: On input a master signing key msk and an attribute set $\mathcal{A}$, it outputs a signing key $sk_{\mathcal{A}}$.
- ABS.Sign$(sk_{\mathcal{A}}, m, \Upsilon) \rightarrow \sigma_{m,\Upsilon}$: On input a signing key $sk_{\mathcal{A}}$, a message m, and a monotone boolean function predicate Υ, where $\Upsilon(\mathcal{A}) = 1$, it outputs a signature $\sigma_{m,\Upsilon}$.
- ABS.Verify$(mvk, m, \Upsilon, \sigma_{m,\Upsilon}) \rightarrow \{0, 1\}$: On input a master verification key mvk, an unverified message m, an unverified monotone boolean function predicate Υ, and a signature $\sigma_{m,\Upsilon}$, it outputs 1 if the signature is valid.

More elaborate procedures, along with extended algorithms, will be given in Section 5.2.

5 EQUALITY QUERY AUTHENTICATION

A naive solution to query authentication with fine-grained access control is to use the Merkle hash tree [20] to construct a verification object (VO) for authenticated query processing and to use CP-ABE to encrypt data records for access control. More specifically, for any query, all data records (including those inaccessible ones) that lie in the query range, along with the VO, are returned to the user. After verifying the records, the user can decrypt and access the authorized records, using the attribute secret key obtained from the DO. However, this solution has two problems. First, it will return a large number of inaccessible records to the user, which incurs high computation and communication overheads. Second, for the inaccessible records, thanks to CP-ABE, although they cannot be decrypted by the user, returning them still reveals some information about their existence and their access policies. This violates our zero-knowledge confidentiality requirement.

In the following, we propose new query authentication techniques that support both fine-grained access control and zero-knowledge confidentiality. We start with equality queries by developing a novel signature scheme in this section and then extend it to range queries and join queries in Section 6.

In an equality query, the user specifies a query key o_q as well as his/her access role set $\mathcal{A}$. Since the query attribute is distinct, there could be three possible outcomes:

- There is one record matching o_q and it is accessible to the query user.
- There is one record matching o_q but it is not accessible to the query user.
- There is no record matching o_q.

Recalling our zero-knowledge confidentiality does not allow the user to distinguish between the last two outcomes. To prevent the information leakage caused by non-existent records, we introduce a global pseudo access role, $Role_\emptyset$, which is not possessed by any user. We treat each non-existent record as a *pseudo* record that is associated with the access policy $Role_\emptyset$. As such, for any equality query, there is always one matching record with one of the two possible outcomes: accessible or inaccessible.

5.1 ADS Generation and Query Processing

ADS Generation. Our construction for zero-knowledge query authentication is built upon a novel signature scheme. During the system setup, the DO generates a signature for each data record. These signatures, serving as authenticated data structures (ADS), are sent to the SP, who will then use them to support verifiable queries. As mentioned earlier, the signature has two design requirements. First, given a record i, the signature should capture all of its three components (query attribute o_i, data content v_i, and access policy Υ_i) so that it can be used as a proof of integrity. Second, in case the record is inaccessible to some users, the signature can be tailored to prove the inaccessibility with the zero-knowledge confidentiality. To this end, we propose a new access-policy-preserving (APP) signature based on a variant of the attribute-based signature

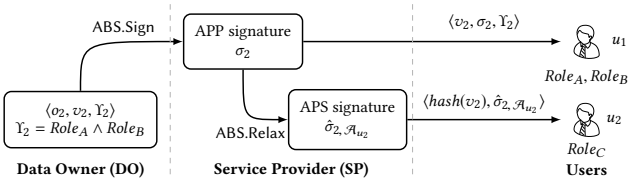


Figure 3: Equality Query Authentication

(ABS) (to be detailed in Section 5.2).

Definition 5.1 (APP Signature). Consider a record $\langle o_i, v_i, \Upsilon_i \rangle$. Let sk_{DO} be the signing key of the DO, $hash(\cdot)$ a cryptographic hash function, and '|' denote string concatenation. The access-policy-preserving (APP) signature for this record is generated as:

$$\sigma_i = \text{ABS.Sign}(sk_{\text{DO}}, hash(o_i)|hash(v_i), \Upsilon_i)$$

In the case of pseudo (non-existent) records, v_i will be assigned a random value and Υ_i will be $Role_\emptyset$.

Authenticated Query Processing. After the SP receives the APP signatures for all records from the DO, it is able to support authenticated query processing by constructing a VO for the query result. Figure 3 shows an example of two different users issuing an equality query with key o_2. User u_1 is allowed to access the record o_2, so it is straightforward to return the APP signature generated by the DO as the VO. That is, the SP will return $\langle v_2, \sigma_2 = \text{ABS.Sign}(sk_{\text{DO}}, hash(o_2)|hash(v_2), \Upsilon_2), \Upsilon_2 \rangle$. On the user side, $hash(o_2)|hash(v_2)$ will be computed based on the query attribute o_2 and the result v_2 returned by the SP. If it can be verified by the signature σ_2 under the policy Υ_2, $\langle o_2, v_2 \rangle$ is a genuine result.

Next, we consider the case in which the data record is inaccessible to the user (e.g., u_2 in Figure 3). Since the query attribute is distinct, we only need to prove the inaccessibility of this data record. Simply returning the APP signature does not work, because this would disclose the access policy and hence the specific reason why the user cannot access the record. To achieve the zero-knowledge confidentiality, the SP can only leverage the information which is already known to the user to prove the inaccessibility. Specifically, the global access role set $\mathbb{A}$ and the user's own role set $\mathcal{A}$ are such information the SP can use.

In a monotone boolean function, given an access role set as input, the reason for it to output 0 is only because it lacks one or more roles in $\mathbb{A}\backslash\mathcal{A}$ (i.e., those roles that the user does not own). Thus, our idea is to make use of a *super* access policy, denoted by $\hat{\Upsilon}_{\mathcal{A}}$, that does not honor the role set $\mathcal{A}$. More specifically, it is a boolean function that fuses each role in $\mathbb{A}\backslash\mathcal{A}$ using the OR operator. For example, in Figure 3, if the access role universe $\mathbb{A}$ is $\{Role_\emptyset, Role_A, Role_B, Role_C\}$, then $\hat{\Upsilon}_{\{Role_C\}} = Role_\emptyset \vee Role_A \vee Role_B$ for user u_2. In essence, it is the weakest access policy under which the user still cannot access the record. Based on this concept, we define the following access-policy-stripped (APS) signature that embeds the super access policy.

Definition 5.2 (APS Signature). Consider a record $\langle o_i, v_i, \Upsilon_i \rangle$. Denoted by $\mathbb{A}$ and $\mathcal{A}$ the global access role set and the query user's role set, respectively. Let sk_{DO} be the signing key of the DO, $hash(\cdot)$ a cryptographic hash function, and '|' denote string concatenation. The access-policy-stripped (APS) signature for this record and the user with access role set $\mathcal{A}$ is defined as:

$$\hat{\sigma}_{i,\mathcal{A}} = \text{ABS.Sign}(sk_{\text{DO}}, hash(o_i)|hash(v_i), \hat{\Upsilon}_{\mathcal{A}}),$$

where $\hat{\Upsilon}_{\mathcal{A}} = a_1 \vee a_2 \vee \cdots \vee a_n, \quad a_i \in \mathbb{A}\backslash\mathcal{A}$.

Algorithm 1: Authentication of Equality Queries

ADS Generation (by the DO)
- **for** *each data record* $\langle o_i, v_i, \Upsilon_i \rangle$ **do**
  - $\sigma_i \leftarrow \text{ABS.Sign}(sk_{\text{DO}}, hash(o_i)|hash(v_i), \Upsilon_i)$;
- Outsource all records $\langle o_i, v_i, \Upsilon_i, \sigma_i \rangle$ to the SP;

VO Construction (by the SP)
- **Input:** Query attribute o_q, access role set $\mathcal{A}$
- **if** $\Upsilon_{o_q}(\mathcal{A}) = 1$ **then** $VO \leftarrow \langle v_{o_q}, \sigma_{o_q}, \Upsilon_{o_q} \rangle$;
- **else**
  - $\hat{\sigma}_{o_q,\mathcal{A}} \leftarrow \text{ABS.Relax}(\sigma_{o_q}, \mathbb{A}\backslash\mathcal{A})$; `// Section 5.2`
  - $VO \leftarrow \langle hash(v_{o_q}), \hat{\sigma}_{o_q,\mathcal{A}} \rangle$;
- send CP-ABE.Encrypt$(pp, VO, \wedge_{a\in\mathcal{A}} a)$ to the user;

Result Verification (by the user)
- run CP-ABE.Decrypt to decrypt the VO;
- **if** $\Upsilon_{o_q}(\mathcal{A}) = 1$ **then**
  - run ABS.Verify$(mvk, hash(o_q)|hash(v_{o_q}), \Upsilon_{o_q}, \sigma_{o_q})$;
- **else**
  - $\hat{\Upsilon}_{\mathcal{A}} \leftarrow \vee_{a\in\mathbb{A}\backslash\mathcal{A}} a$;
  - run ABS.Verify$(mvk, hash(o_q)|hash(v_{o_q}), \hat{\Upsilon}_{\mathcal{A}}, \hat{\sigma}_{o_q,\mathcal{A}})$;

The APS signature, which can be derived from the APP signature (see Section 5.2 for details), serves as the VO for an inaccessible record. By the embedded super access policy, the user is able to verify the inaccessibility, but cannot infer the specific roles he/she lacks. In the running example of Figure 3, for user u_2, the SP will return $\langle hash(v_2), \hat{\sigma}_{2,\mathcal{A}_{u_2}} = \text{ABS.Sign}(sk_{\text{DO}}, hash(o_2)|hash(v_2), \hat{\Upsilon}_{\{Role_C\}}) \rangle$. During result verification, $hash(o_2)|hash(v_2)$ will be computed by the user based on the query attribute o_2 and the value $hash(v_2)$ returned by the SP, and $\hat{\Upsilon}_{\{Role_C\}}$ will be computed from the user's access role set. With these components, the user can verify the super policy $\hat{\Upsilon}_{\{Role_C\}}$ via the APS signature $\hat{\sigma}_{2,\mathcal{A}_{u_2}}$ to prove that the record is indeed inaccessible to the user.

Finally, to prevent impersonation attacks, the SP will encrypt the query result as well as the VO using a traditional one-key cipher, such as AES, with the one-key cipher key encrypted using CP-ABE under the access policy $a_1 \wedge a_2 \wedge \cdots \wedge a_n$, $a_i \in \mathcal{A}$. Therefore, only the query user who indeed has the access role set $\mathcal{A}$ as claimed can decrypt the query result.

Algorithm 1 summarizes the procedures discussed above for authenticating equality queries with the zero-knowledge confidentiality.

5.2 ABS with Predicate Relaxation

In this section, we develop a variant of the ABS scheme, which supports predicate relaxation on super access policies and hence can be used to generate the APP signature.

5.2.1 Monotone Span Program. To build the ABS scheme, we first introduce the monotone span program, which is a special form of matrix that can be used to present its equivalent monotone boolean function. It is defined as follows.

Definition 5.3 (Monotone Span Program). Let $\Upsilon : \{0,1\}^n \to \{0,1\}$ be a monotone boolean function. A monotone span program for Υ over a field $\mathbb{F}$ is an $\ell \times t$ matrix $\mathbf{M}$ with entries $\mathbb{F}$, with a labeling function $a : [\ell] \to [n]$, which associates each row of $\mathbf{M}$ with an input variable of Υ. For every $(x_1, \cdots, x_n) \in \{0,1\}^n$, it satisfies the following:

$$\Upsilon(x_1, \cdots, x_n) = 1 \iff \exists \mathbf{v} \in \mathbb{F}^{1\times\ell} : \mathbf{vM} = [1, 0, 0, \cdots, 0] \text{ and } (\forall i : x_{a(i)} = 0 \Rightarrow \mathbf{v}_i = 0)$$

In other words, $\Upsilon(x_1, \cdots, x_n) = 1$ if and only if the rows of $\mathbf{M}$ indexed by $\{i \mid x_{a(i)} = 1\}$ span the vector $[1, 0, 0, \cdots, 0]$.

There are many different approaches to constructing a monotone span program from a monotone boolean function [18, 21]. In this paper, we choose a recursive algorithm as shown in Algorithm 5 in Appendix A.1, which accepts a boolean function expressed in AND and OR operators as inputs [21].

5.2.2 ABS Construction. Derived from the Practical Instantiation 4 in [19], our ABS construction consists of the following algorithms.

ABS.Setup$(1^\lambda) \rightarrow (msk, mvk)$: Let $(\mathbb{G}, \mathbb{H}, \mathbb{G}_T, e)$ be a bilinear pairing. Choose random generators:

$$g, C \leftarrow \mathbb{G}; \qquad h_0, h \leftarrow \mathbb{H}.$$

Choose random values $a_0, a, b \leftarrow \mathbb{Z}_p$ and compute:

$$A_0 = h_0^{a_0}; \qquad A = h^a \text{ and } B = h^b.$$

The master signing key is $msk = (a_0, a, b)$, and the master verifying key is $mvk = (g, h_0, h, A_0, A, B, C)$.

ABS.KeyGen$(msk, \mathcal{A}) \rightarrow sk_{\mathcal{A}}$: Choose random value $K_{\text{base}} \leftarrow \mathbb{G}$. The signing key is computed as:

$$sk_{\mathcal{A}} = (K_{\text{base}}, K_0 = K_{\text{base}}^{1/a_0}, \{K_u = K_{\text{base}}^{1/(a+bu)} | u \in \mathcal{A}\}).$$

ABS.Sign$(sk_{\mathcal{A}}, m, \Upsilon) \rightarrow \sigma_{m,\Upsilon}$: Convert Υ to its corresponding monotone span program $\mathbf{M} \in \mathbb{Z}_p^{\ell \times t}$, with row labeling $u : [\ell] \rightarrow \mathbb{A}$, where $\mathbb{A}$ denotes the universe of attributes. Also compute the vector $\mathbf{v} = [v_0, \ldots, v_\ell]$ that corresponds to the satisfying attributes $\mathcal{A}$. Pick random values $\tau, r_0, r_1, \ldots, r_\ell \leftarrow \mathbb{Z}_p$. The signature is composed as:

$$\sigma_{m,\Upsilon} = (\tau, Y, W, S_1, \ldots, S_\ell, P_1, \ldots, P_t);$$
$$Y = K_{\text{base}}^{r_0}; \quad S_i = K_{u(i)}^{v_i \cdot r_0} \cdot (Cg^{\text{hash}})^{r_i} \quad (\forall i \in [\ell]);$$
$$W = K_0^{r_0}; \quad P_j = \prod_{i=1}^{\ell} (AB^{u(i)})^{\mathbf{M}_{ij} \cdot r_i} \quad (\forall j \in [t]).$$

Here hash $= hash(\tau, m)$ for some collision-resistant hash function $hash(\cdot)$. Note that the signer may not have $K_{u(i)}$ for every attribute mentioned in the claim-predicate. However, in such cases, $v_i = 0$.

ABS.Verify$(mvk, m, \Upsilon, \sigma_{m,\Upsilon}) \rightarrow \{0, 1\}$: To verify the signature, the verifier converts Υ to its corresponding monotone span program $\mathbf{M} \in \mathbb{Z}_p^{\ell \times t}$, with row labeling $u : [\ell] \rightarrow \mathbb{A}$. The signature is valid if and only if all of the following constraints hold:

$$Y \stackrel{?}{\neq} 1; \qquad e(W, A_0) \stackrel{?}{=} e(Y, h_0);$$

$$\prod_{i=1}^{\ell} e\left(S_i, (AB^{u(i)})^{\mathbf{M}_{ij}}\right) \stackrel{?}{=} \begin{cases} e(Y, h) e(Cg^{\text{hash}}, P_1) & \text{if } j = 1, \\ e(Cg^{\text{hash}}, P_j) & \text{if } j > 1. \end{cases}$$

5.2.3 Predicate Relaxation. Our ABS scheme supports predicate relaxation. This enables the SP to derive new signatures on super access policies. It works as follows.

ABS.Relax$(\sigma_{m,\Upsilon}, \mathcal{A}') \rightarrow \sigma_{m,\Upsilon'}$: Given a signature $\sigma_{m,\Upsilon}$ signed under predicate Υ and an attribute set $\mathcal{A}'$, this operation outputs a new signature $\sigma_{m,\Upsilon'}$ signed under the new predicate $\Upsilon' = \vee_{a \in \mathcal{A}'} a$. It can succeed if and only if $\Upsilon'(\mathcal{A}) = 1, \forall \mathcal{A} \in \{A \mid \Upsilon(A) = 1, A \subseteq \mathbb{A}\}$. In other words, $\mathcal{A}'$ should satisfy the condition $\Upsilon(\mathbb{A} \backslash \mathcal{A}') = 0$. For example, in Figure 2, the APP signature of o_2 has the predicate $\Upsilon_2 = Role_A \wedge Role_B$. By invoking ABS.Relax, it can be relaxed to a new signature, the predicate of which is $\Upsilon_2' = Role_\emptyset \vee Role_A \vee Role_B$, for user u_2 (i.e., u_2's super access policy), because $\Upsilon_2(\mathbb{A} \backslash \{Role_\emptyset, Role_A, Role_B\}) = \Upsilon_2(\{Role_C\}) = 0$. However, relaxing it to a signature, say with predicate $\Upsilon_2'' = Role_\emptyset \vee Role_C$, will fail, as $\Upsilon_2(\mathbb{A} \backslash \{Role_\emptyset, Role_C\}) = \Upsilon_2(\{Role_A, Role_B\}) = 1$.

Algorithm 2: ABS.Relax

Function ABS.Relax$(\sigma_{m,\Upsilon}, \mathcal{A}')$
- **Input:** ABS signature $\sigma_{m,\Upsilon}$, attribute set $\mathcal{A}'$
- **Output:** ABS signature $\sigma_{m,\Upsilon'}$, where $\Upsilon' = \vee_{a \in \mathcal{A}'} a$
- $\langle \tau, Y, W, \{S_i\}, \{P_j\} \rangle \leftarrow \sigma_{m,\Upsilon}$;
- // step 1: purging unwanted attributes
- $\langle rows, cols, L, kept\_rows, kept\_cols, flag \rangle \leftarrow$ Purge$(\Upsilon, \mathcal{A}')$;
- **if** $flag$ **is** *false* **then abort**;
- $\tilde{P}_1 \leftarrow \prod_{j \in kept\_cols} P_j$;
- **for** $i = 1$ **to** $len(\mathcal{A}')$ **do**
  - $u \leftarrow \mathcal{A}'[i]$;
  - **if** $u \in \{L[l] | l \in kept\_rows\}$ **then**
    - // step 2: merging duplicate attributes
    - $\tilde{S}_i \leftarrow \prod_{k \in \{l | L[l] = u\}} S_k$;
  - **else**
    - // step 3: appending missing attributes
    - $r \leftarrow$ random value, $\tilde{S}_i \leftarrow (Cg^{\text{hash}})^r, \tilde{P}_1 \leftarrow \tilde{P}_1 \cdot (AB^u)^r$;
- // step 4: re-randomizing signature
- $r \leftarrow$ random value, $\sigma_{m,\Upsilon'} \leftarrow \langle \tau, Y^r, W^r, \{\tilde{S}_i^r\}, \tilde{P}_1^r \rangle$;
- **return** $\sigma_{m,\Upsilon'}$;

The overall procedure is summarized in Algorithm 2. It consists of the following steps:

(1) Purging the attributes existing in Υ but not existing in $\mathcal{A}'$. The algorithm is illustrated in Algorithm 6 in Appendix A.2, which is essentially a modified version of Algorithm 5. The idea is to perform a bottom-up search on the monotone boolean function tree of Υ. However, instead of building the monotone span program, we compute which rows (S_i) and columns (P_j) in the original monotone span program (signature $\sigma_{m,\Upsilon}$) should be kept. When traversing the tree, both AND and OR operators will be scanned. When the tree node is an AND operator, we choose one of the qualified child nodes to keep. In contrast, when it is an OR operator, we will have to keep all the child nodes and require them to be qualified. Here, we say a node is qualified if and only if the sub-predicate from that node yields 1 when given $\mathcal{A}'$ as input. The result can then be used to construct a new signature whose predicate is $\vee_{i \in kept\_rows} u(i)$, as shown below, where u is the labeling function.

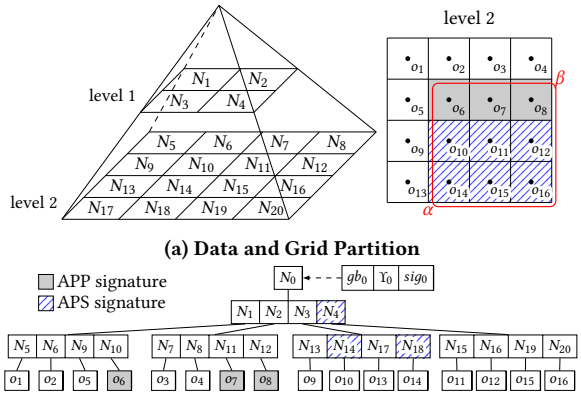
$$\tilde{\sigma} = (\tau, \tilde{Y}, \tilde{W}, \{\tilde{S}_i\}, \tilde{P}_1);$$
$$\tilde{Y} = Y; \quad \tilde{S}_i = S_i \quad (\forall i \in kept\_rows);$$
$$\tilde{W} = W; \quad \tilde{P}_1 = \prod_{k \in kept\_cols} P_k.$$

It is worth noting that if the relationship between Υ and $\mathcal{A}'$ cannot be satisfied, at the end of tree traversal, we will find none of tree nodes is qualified. This is because we cannot purge the attributes from an OR operator, which is required if $\Upsilon(\mathbb{A} \backslash \mathcal{A}') \neq 0$.

(2) Merging the duplicate attributes. In $\tilde{\sigma}$ obtained from the last step, there may be duplicate attributes in $u(i), i \in kept\_rows$. If so, we can merge them by computing a new $\tilde{S'}_i$ as the product of all the $\tilde{S}_k$'s that share the same label $u(i)$. The rest of $\tilde{Y}, \tilde{W}, \tilde{P}_1$ in the signature will remain unchanged.

$$\tilde{\sigma'} = (\tau, \tilde{Y'}, \tilde{W'}, \{\tilde{S'}_i\}, \tilde{P'}_1);$$
$$\tilde{Y'} = \tilde{Y}; \quad \tilde{S'}_i = \prod_{k \in \{l | u(l) = u(i)\}} \tilde{S}_k;$$
$$\tilde{W'} = \tilde{W}; \quad \tilde{P'}_1 = \tilde{P}_1.$$

(3) Appending the missing attributes. There might be attributes existing in $\mathcal{A}'$ but not in the signature $\tilde{\sigma}'$ obtained from the last step. We add them back in the following manner, where r_i is a random



(a) Data and Grid Partition

(b) Index

Figure 4: Access-Policy-Preserving Grid-Tree (AP^{2}G-Tree)

number. The result $\hat{\sigma}$ will be the new signature, the predicate of which is $\vee_{a\in\mathcal{A}'} a$.

$$\hat{\sigma} = (\tau, \hat{Y}, \hat{W}, \{\hat{S}\}, \hat{P}_1);$$

$$\hat{Y} = \tilde{Y}'; \quad \hat{S}_i = \begin{cases} \tilde{S}'_i & \text{if } i \in kept\_rows, \\ (Cg^{\text{hash}})^{r_i} & \text{otherwise;} \end{cases}$$

$$\hat{W} = \tilde{W}'; \quad \hat{P}_1 = \tilde{P}'_1 \cdot \prod_{i\in\{l|l\notin kept\_rows, u(l)\in\mathcal{A}'\}} (AB^{u(i)})^{r_i}.$$

(4) Re-randomizing the signature. To further enhance security, in the last step, we re-randomize the signature, as below, to obtain the final signature, where r is a random number:

$$\sigma_{m,\Upsilon'} = (\tau, \hat{Y}^r, \hat{W}^r, \{\hat{S}_i^r\}, \hat{P}_1^r)$$

The security of this ABS construction will be proven in Section 7.1.

6 RANGE & JOIN QUERY AUTHENTICATION

6.1 Range Query Authentication

In the range query scenario, the user submits a query range $[\alpha, \beta]$[4] as well as his/her access role set $\mathcal{A}$. In turn, the SP returns all records in the range $[\alpha, \beta]$ that are accessible to the query user. A naive solution would be to execute the equality query algorithm developed in Section 5 repeatedly for every discrete value in the range $[\alpha, \beta]$. However, this is intrinsically costly. To boost the performance, we propose *access-policy-preserving grid-tree* (AP^{2}G-tree), an authenticated index structure for the DO to construct and sign. It is worth noting that we choose a grid tree to prevent the user from learning any knowledge regarding the data record distribution through the structure of the index tree.

Consider a 2D data space for illustration. Figure 4a shows a multi-layer grid system, which partitions the query attribute space recursively into multiple levels of grid cells until each cell contains only one record. The bounding box of each cell is called a grid box. Figure 4b shows the corresponding AP^{2}G-tree for the records in Figure 4a. Each grid cell has a corresponding tree node N_i in the AP^{2}G-tree, which consists of three components: grid box[5] (denoted by gb_i), access policy (p_i), and APP signature (denoted by sig_i). The latter two components are computed from its C child entries, $c_1, \ldots, c_C$, in the following fashion.

Definition 6.1 (AP^{2}G-tree Non-Leaf Node). Let sk_{DO} be the signing key of the DO and $hash(\cdot)$ a cryptographic hash function. The access policy and APP signature of a non-leaf node are defined as:

$$p_i = p_{c_1} \vee p_{c_2} \vee \cdots \vee p_{c_C}$$
$$sig_i = \text{ABS.Sign}(sk_{\text{DO}}, gb_i, p_i)$$

For example, in Figure 4b, the access policy for N_1 is computed as $p_{N_1} = p_{N_5} \vee p_{N_6} \vee p_{N_9} \vee p_{N_{10}}$, and its APP signature is $sig_{N_1} = \text{ABS.Sign}(sk_{\text{DO}}, gb_{N_1}, p_{N_1})$.

Definition 6.2 (AP^{2}G-tree Leaf Node). The access policy and APP signature of a leaf node are identical to those of the single record lying in the corresponding cell.

For example, for leaf node N_5, its policy and signature are Υ_{o_1} and σ_{o_1}, respectively.

The AP^{2}G-tree is built by the DO in a bottom-up fashion and is then outsourced to the SP. It is worth noting that the AP^{2}G-tree is always a full tree, since we treat data records not existing in the original database as pseudo records inaccessible to any user, as explained in Section 5. This construction yields a space cost of $O((n + \log(n))m)$, where n is the size of the index space and m is the size of the monotone span program of the access policies. With the definitions above, the access policy of a non-leaf node essentially represents whether or not a user can access any record inside the grid box of this node. This property will allow us to perform effective pruning in the construction of VO during query processing.

On the SP side, the processing of a range query $[\alpha, \beta]$ can be executed as a breadth-first search. Starting from the root node, if a non-leaf node partially intersects the query range, it will be branched, i.e., its subtree is further explored. On the other hand, if a non-leaf node is fully covered by the query range, the SP will check whether or not the query user is allowed to access this node. If access is prohibited, the SP will run ABS.Relax to compute the APS signature for this node and add this signature to the VO. In contrast, if the access is permitted, the SP will further explore the subtree until a leaf node is reached. The process of handling a leaf node is the same as that for an equality query. Finally, similar to an equality query, all the results, as well as the VO, will be encrypted using AES and CP-ABE before they are transmitted to the user.

To establish the soundness and completeness of the query results, the user checks the VO in two aspects:

- **Soundness check**. All of the signatures in the VO are valid; for an inaccessible node, the predicate of the corresponding signature is indeed $\vee_{a\in\mathbb{A}\setminus\mathcal{A}} a$; for an accessible record, it is indeed inside the query range $[\alpha, \beta]$.
- **Completeness check**. The union of the indexing spaces for each entry of the VO covers the whole query range $[\alpha, \beta]$. Note that this check is sufficient because one and only one entry is expected for each indexing space.

We summarize the algorithm for authenticating range queries with the zero-knowledge confidentiality in Algorithm 3. Figure 4 illustrates an example in which the user can only access o_6, o_7, o_8 (highlighted with a gray color) in the range query $[\alpha, \beta]$ (highlighted as the red rectangle in Figure 4a). As a result, the SP will return the records o_6, o_7 and o_8, as well as their APP signatures, to the user. Moreover, the SP will run the ABS.Relax algorithm with the APP signatures of N_4, N_{14} and N_{18} as inputs. The derived APS signatures will be sent to the user as part of the VO, to prove their inaccessibility. During result verification, the user will check

[4] α and β are two points that represent the lower and upper bounds of the query range.
[5] A grid box is represented by the coordinates of its lower-left and upper-right points.

Algorithm 3: Authentication of Range Queries

ADS Generation (by the DO)
- **for** *each data record* $\langle o_i, v_i, \Upsilon_i \rangle$ **do**
  - $\sigma_i \leftarrow \text{ABS.Sign}(sk_{\text{DO}}, hash(o_i)|hash(v_i), \Upsilon_i)$;
- **for** gb_i **in** *each* AP^{2}G-tree *non-leaf node* **do**
  - $p_i \leftarrow \vee_{k=1}^{C} p_{c_k}$;
  - $sig_i \leftarrow \text{ABS.Sign}(sk_{\text{DO}}, hash(gb_i), p_i)$;
- Outsource all $\langle o_i, v_i, \Upsilon_i, \sigma_i \rangle$ and $\langle gb_i, p_i, sig_i \rangle$ to SP;

VO Construction (by the SP)
- **Input:** Query range $[\alpha, \beta]$, access role set $\mathcal{A}$
- create an empty queue q;
- q.enqueue(AP^{2}G-tree root);
- **while** q is not empty **do**
  - $n \leftarrow q$.dequeue();
  - **if** n *partially intersects* $[\alpha, \beta]$ **then** q.enqueue(n.children);
  - **else if** n *lies inside* $[\alpha, \beta]$ **then**
    - **if** n *is accessible to the user* **then**
      - **if** n **is** *a leaf node* **then** add$\langle o_n, v_n, \sigma_n, \Upsilon_n \rangle$to VO;
      - **else** q.enqueue(n.children);
    - **else**
      - **if** n **is** *a leaf node* **then**
        - $\hat{\sigma}_{n,\mathcal{A}} \leftarrow \text{ABS.Relax}(\sigma_n, \mathbb{A} \backslash \mathcal{A})$;
        - add $\langle hash(v_n), \hat{\sigma}_{n,\mathcal{A}} \rangle$ to VO;
      - **else**
        - $\hat{sig}_{n,\mathcal{A}} \leftarrow \text{ABS.Relax}(sig_n, \mathbb{A} \backslash \mathcal{A})$;
        - add $\langle gb_n, \hat{sig}_{n,\mathcal{A}} \rangle$ to VO;
- send CP-ABE.Encrypt(pp, VO, $\wedge_{a \in \mathcal{A}} a$) to the user;

Result Verification (by the user)
- run CP-ABE.Decrypt to decrypt the VO;
- check if the union of the region for each entry in the VO covers $[\alpha, \beta]$;
- $\hat{\Upsilon}_{\mathcal{A}} \leftarrow \vee_{a \in \mathbb{A} \backslash \mathcal{A}} a$;
- **for** *each entry* e **in** *VO* **do**
  - **if** e *is accessible to the user* **then**
    - check $o_e \in [\alpha, \beta]$ and $\Upsilon_e(\mathcal{A}) = 1$;
    - run ABS.Verify(mvk, $hash(o_e)|hash(v_e)$, Υ_e, σ_e);
  - **else if** e **is** *a data record* **then**
    - run ABS.Verify(mvk, $hash(o_e)|hash(v_e)$, $\hat{\Upsilon}_{\mathcal{A}}$, $\hat{\sigma}_{e,\mathcal{A}}$);
  - **else** run ABS.Verify(mvk, gb_e, $\hat{\Upsilon}_{\mathcal{A}}$, $\hat{sig}_{e,\mathcal{A}}$);

whether or not all of the signatures in the VO are valid (to verify the soundness) and whether or not the query range $[\alpha, \beta]$ is indeed covered by the union of the indexing spaces for N_4, N_{14}, N_{18}, o_6, o_7, and o_8 (to verify the completeness).

6.2 Join Query Authentication

Next, we discuss how to extend the range query authentication algorithm to support join queries. Consider an equi-join query over two tables R and S, $R \bowtie_{R.o=S.o} S \wedge R.o \in [\alpha, \beta]$, with user's access role set $\mathcal{A}$. The SP should return all record pairs in the query range that are accessible to the query user and satisfy the join condition. For example, in Figure 5, the user can access r_1, r_2 from R and s_1, s_3 from S (highlighted with a gray color); the join result will be the pair of records $\langle r_1, s_1 \rangle$.

To process the join query, the SP starts by executing a breadth-first search from the root node of the AP^{2}G-tree for the table R. The tree search algorithm is almost identical to the one in the range query processing, except for two aspects. First, if a node for R, denoted as N_R, is accessible to the user, the SP will first check whether or not there exists an inaccessible node in the AP^{2}G-tree for

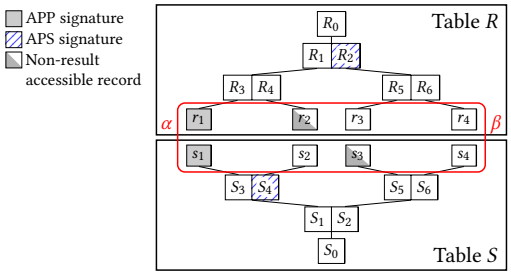


Figure 5: Join Query Authentication

Algorithm 4: Authentication of Joins (VO Construction)

- **Input:** AP^{2}G-tree $\mathcal{T}_R$, $\mathcal{T}_S$, Query range $[\alpha, \beta]$, access role set $\mathcal{A}$
- create an empty queue q;
- q.enqueue($\langle \mathcal{T}_R$ root, $\mathcal{T}_S$ root$\rangle$);
- **while** q is not empty **do**
  - $\langle n_R, n_S \rangle \leftarrow q$.dequeue();
  - **if** n_R *partially intersects* $[\alpha, \beta]$ **then**
    - **for** n **in** n_R.*children* **do** q.enqueue($\langle n, n_S \rangle$);
  - **else if** n_R *lies inside* $[\alpha, \beta]$ **then**
    - **if** n_R *is accessible to the user* **then**
      - $n_S \leftarrow$ the smallest node under n_S, which also covers the region of n_R;
      - **if** n_S *is accessible to the user* **then**
        - **if** n_R **is** *a leaf node* **then**
          - `// In this case, n_S is also a leaf node.`
          - add APP signatures of n_R and n_S to VO;
        - **else**
          - **for** n **in** n_R.*children* **do** q.enqueue($\langle n, n_S \rangle$);
      - **else** compute and add APS signature of n_S to VO;
    - **else** compute and add APS signature of n_R to VO;
- send CP-ABE.Encrypt(pp, VO, $\wedge_{a \in \mathcal{A}} a$) to the user;

the table S that covers N_R. If such a node, denoted as N_S, is found, N_R cannot be part of the join result. Hence, the corresponding APS signature for N_S is computed and appended to the VO. Otherwise, the subtree of N_R is further explored. Second, an accessible record in R is part of the result only if the matching record in S is also accessible. In this case, the APP signatures for this pair of records will be appended to the VO. The rest of the algorithm is the same as the one in the range query processing.

We summarize the VO construction procedure in Algorithm 4, while the ADS generation and result verification procedures are the same as the ones in Algorithm 3. In the example of Figure 5, the SP will return the records r_1 and s_1, as well as their APP signatures, to the user. Moreover, for the inaccessible nodes R_2 and S_4, their derived APS signatures will be returned as part of the VO. On the user side, to verify the soundness, the user will authenticate all the signatures in the VO and check whether or not r_1 and s_1 share the same value on the join attribute; to verify the completeness, the user will check whether or not the union of the indexing spaces for R_2, S_4, and r_1/r_2 covers the whole query range $[\alpha, \beta]$.

This solution can be easily extended to support more general join queries, such as multi-way join and inequality join. The general idea is similar, i.e., using the APS signature of an inaccessible record/node to prove that a certain space does not contribute to the join result. The user verifies the soundness by the given results and their associated APP signatures, and verifies the completeness by checking whether or not the result set and the space represented by the APS signatures together cover the whole query range.

7 SECURITY ANALYSIS

In this section, we perform a security analysis on our proposed ABS scheme and query authentication algorithms.

7.1 Security Analysis on ABS

In order to facilitate our security analysis, we first present the formal definitions of our security notions.

Definition 7.1 (Perfect Privacy). We say an ABS scheme achieves perfect privacy if for all $(msk, mvk) \leftarrow \mathsf{ABS.Setup}(1^\lambda)$, all messages m, such that:

- For all attribute sets $\mathcal{A}_1$, $\mathcal{A}_2$, all signing keys $sk_{\mathcal{A}_1} \leftarrow \mathsf{ABS.KeyGen}(msk, \mathcal{A}_1)$, $sk_{\mathcal{A}_2} \leftarrow \mathsf{ABS.KeyGen}(msk, \mathcal{A}_2)$, all claim-predicates Υ such that $\Upsilon(\mathcal{A}_1) = \Upsilon(\mathcal{A}_2) = 1$, the distributions of $\mathsf{ABS.Sign}(sk_{\mathcal{A}_1}, m, \Upsilon)$ and $\mathsf{ABS.Sign}(sk_{\mathcal{A}_2}, m, \Upsilon)$ are identical.
- For all attribute sets $\mathcal{A}, \mathcal{A}'$, all signing keys $sk_{\mathcal{A}} \leftarrow \mathsf{ABS.KeyGen}(msk, \mathcal{A})$, $sk_{\mathcal{A}'} \leftarrow \mathsf{ABS.KeyGen}(msk, \mathcal{A}')$, all claim-predicates Υ such that $\Upsilon(\mathbb{A} \backslash \mathcal{A}') = 0$ and all signatures $\sigma_{m,\Upsilon} \leftarrow \mathsf{ABS.Sign}(sk_{\mathcal{A}}, m, \Upsilon)$, the distributions of $\mathsf{ABS.Sign}(sk_{\mathcal{A}'}, m, \vee_{a \in \mathcal{A}'} a)$ and $\mathsf{ABS.Relax}(\sigma_{m,\Upsilon}, \mathcal{A}')$ are identical.

This property ensures that an ABS signature generated by ABS.Sign or ABS.Relax will not leak the information about which set of attributes or signing key was used in signing. This is a necessary condition to support zero-knowledge.

Definition 7.2 (Unforgeability). We say an ABS scheme is unforgeable if the success probability of any polynomial-time adversary is negligible in the following experiment:

- Run $(msk, mvk) \leftarrow \mathsf{ABS.Setup}(1^\lambda)$ and give mvk to the adversary.
- The adversary is given access to oracle $\mathsf{ABS.Sign}(\cdot)$, which, on input (m, Υ), outputs the corresponding signature σ. Let $\{m^{(\iota)}, \Upsilon^{(\iota)}, \sigma^{(\iota)}\}_{\iota=1}^{q}$ be the message-policy-signature tuples obtained by the adversary with its q queries.
- At the end the adversary outputs $(m^*, \Upsilon^*, \sigma^*)$.

We say the adversary succeeds if $\mathsf{ABS.Verify}(mvk, m^*, \Upsilon^*, \sigma^*)$ outputs 1 and one of the following results is true:

- $m^* \neq m^{(\iota)}$ for $\iota = 1$ to q.
- For each ι such that $m^* = m^{(\iota)}$, Υ^* is a more restricted policy compared with $\Upsilon^{(\iota)}$. Furthermore, Υ^* is of the form $(\vee_{a \in \mathcal{A}'} a)$ for some $\mathcal{A}' \subset \mathbb{A}$, where $\mathbb{A}$ represents the attribute universe.

This property ensures that a malicious SP could convince the user of an incorrect answer with at most a negligible probability. It is worth noting that this security model of ABS is defined with respect to the threat model of our problem. And it is weaker than the typical security model for ABS in two ways. First, the attacker will not have access to signing keys for different attributes, since the DO is the only party who processes them. Second, a valid forgery must be either a signature on a new message or on a message that has been signed on a more restricted policy. By contrast, in a typical security model of ABS, any new message-policy pair is considered a valid forgery.

Now we reach the theorem on the security of our ABS scheme.

Theorem 7.3. *The construction of Section 5.2 satisfies the security properties of Perfect Privacy (Definition 7.1) and Unforgeability (Definition 7.2) in the generic group model.*

Proof. Please see Appendix B for detailed proofs. □

7.2 Security Analysis on Query Authentication

The formal definition for desired security on our query authentication algorithms consists of two properties, namely unforgeability and zero-knowledge.

Definition 7.4 (Unforgeability). We say our proposed query authentication algorithms are unforgeable if the success probability of any polynomial-time adversary is negligible in the following experiment:

- Run the ADS generation and give all $\langle o_i, v_i, \Upsilon_i, \sigma_i \rangle$ and $\langle gb_i, p_i, sig_i \rangle$ to the adversary.
- The adversary outputs a query q (either a range or join query), an access role set $\mathcal{A}$, a result set RS, and a VO.

We say the adversary succeeds if the VO passes the result verification and one of the following results is true:

- The result set RS contains a record $\langle o^*, v^*, \Upsilon^* \rangle$, such that $\langle o^*, v^*, \Upsilon^* \rangle \neq \langle o_i, v_i, \Upsilon_i \rangle, \forall i$.
- The result set RS contains a record $\langle o^*, v^*, \Upsilon^* \rangle$, such that o^* does not satisfy the query q or $\Upsilon^*(\mathcal{A}) = 0$.
- There exists a record $\langle o_j, v_j, \Upsilon_j \rangle$ not in RS, such that $j \in \{i\}$, o_j satisfies the query q, and $\Upsilon_j(\mathcal{A}) = 1$.

This property ensures that a malicious SP could convince the user of an incorrect or incomplete answer with at most a negligible probability.

Definition 7.5 (Zero-Knowledge). We say our proposed query authentication algorithms are zero-knowledge if the success probability of any polynomial-time adversary is negligible in the following experiment:

- **Game Real**:
  Setup: the adversary picks a database $\mathcal{D}$ and an access role set $\mathcal{A}$, the real challenger runs ADS generation over this database.
  Query: the adversary runs the interactive query protocol with the challenger under the access role set $\mathcal{A}$.
- **Game Ideal**:
  Setup: the adversary picks a database $\mathcal{D} = \{\langle o_i, v_i, \Upsilon_i \rangle\}$ and an access role set $\mathcal{A}$, the simulator runs ADS generation over a database $\mathcal{D}' = \{\langle o_i', v_i', \Upsilon_i' \rangle\}$, such that:

$$\langle o_i', v_i', \Upsilon_i' \rangle = \begin{cases} \langle o_i, v_i, \Upsilon_i \rangle & \text{if } \Upsilon_i(\mathcal{A}) = 1, \\ \langle o_i, \mathsf{random}, Role_\emptyset \rangle & \text{otherwise.} \end{cases}$$

  Query: the adversary runs the interactive query protocol with the simulator under the access role set $\mathcal{A}$.

We say the adversary succeeds if he/she can distinguish the above two games.

This property ensures that a malicious user could extract any information regarding the database beyond accessible records with at most a negligible probability. It is worth noting that, by achieving the zero-knowledge confidentiality, the data content confidentiality and access policy confidentiality are automatically guaranteed.

With the above security definitions, we can show that our proposed query authentication algorithms indeed satisfy the desired security requirements.

Theorem 7.6. *Our proposed query authentication algorithms satisfy the security properties of Unforgeability (Definition 7.4) and Zero-Knowledge (Definition 7.5).*

Proof. Please see Appendix C for detailed proofs. □

8 OPTIMIZATIONS

This section presents two optimization techniques. They are orthogonal to each other, and therefore can be combined to maximize the performance.

8.1 Hierarchical Role Assignment

Since the performance of ABS depends on the number of input roles, one way to optimize the performance is to reduce the size of inaccessible predicates for query users. Thus, we propose hierarchical role assignment. In a hierarchical structure, not possessing one access role implies the lack of some other access roles. For example, if the access role universe is $\{Role_A, Role_{A,S}, Role_{A,P}, Role_B, Role_{B,S}, Role_{B,P}\}$, which corresponds to a member of university A or B, a student of university A or B, and a professor of university A or B, respectively. It is easy to see that a user who is not a member of university A (access role $Role_A$) cannot have any access role associated with university A, i.e., $Role_{A,S}$ and $Role_{A,P}$. As such, the inaccessible predicate for a user with access role $Role_{B,S}$ can be simplified to $Role_A \vee Role_{B,P}$.

Regarding the performance, with hierarchical role assignment, the overhead of the DO setup time is expected to increase slightly, owing to a larger access policy in each record. For example, a record which can only be accessed by the professors of university A will have the access policy $Role_A \wedge Role_{A,P}$, instead of just $Role_{A,P}$. However, with a much smaller inaccessible predicate, the costs of the SP query time and user verification time can be reduced dramatically.

8.2 Acceleration by Parallelism

In our solution to range and join query authentication, the majority of computational overheads comes from the ABS.Relax operations. Fortunately, since these operations are independent to each other, they are highly parallelizable. In light of this, we can accelerate the authentication performance by embracing parallel computing architectures, such as multi-threading, GPU computing, and MapReduce. For example, when the SP processes a query, we can map all ABS.Relax jobs for the inaccessible nodes to the available computing units, whether they are CPU cores or worker nodes in the MapReduce model.

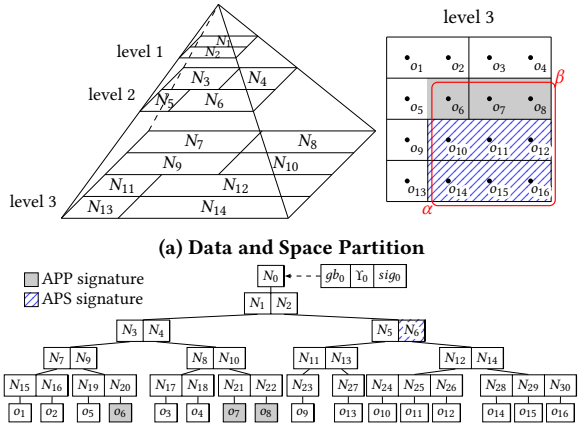
9 RELAXING ZERO-KNOWLEDGE CONFIDENTIALITY REQUIREMENT

Zero-knowledge confidentiality is a strong requirement that may not be demanded in some less critical applications. In this section, we discuss how to boost the performance for range queries and join queries (Section 9.1) and support more general continuous query attributes (Section 9.2), when such a requirement is relaxed to the less restrictive access policy confidentiality.

9.1 Using k-d Tree Index Structure

Recall in Section 6 that the grid tree is chosen to prevent information leakage through the tree structure. However, it is well known that the grid tree is not the most efficient index structure, especially for sparse multi-dimensional databases. Here, in the system that only requires the access policy confidentiality, we propose *access-policy-preserving k-d-tree* (AP^2kd-tree) as an alternative ADS that uses k-d tree [1] to optimize the performance.

Figure 6a shows a k-d tree structure, which splits the query attribute space into two half-spaces at each level. To achieve the maximum efficiency, we aim to increase the chance of pruning inaccessible nodes during query processing. For each split operation, we should find the hyperplane such that given a user's role set, the chance that he/she can access both half-spaces is minimum. Recall that we express the access policies in a *disjunctive normal form* (DNF). Thus, we strive to find the hyperplane such that the intersection of the corresponding OR operator sets of the DNF access policies in the two half-spaces is minimum.



(a) Data and Space Partition

(b) Index

Figure 6: Access-Policy-Preserving k-d-Tree (AP^2kd-Tree)

Let Υ_l and Υ_r be the access policies of two half-spaces. Our goal is to minimize the following objective function:

$$f(\Upsilon_l, \Upsilon_r) = |X_l \cap X_r|,$$

where $\Upsilon_l = \vee_{A_j \in X_l} \wedge_{a_{ij} \in A_j} a_{ij}$ and $\Upsilon_r = \vee_{A_j \in X_r} \wedge_{a_{ij} \in A_j} a_{ij}$.

For example, if $\Upsilon_l = Role_A \vee (Role_B \wedge Role_C)$ and $\Upsilon_r = Role_A \vee Role_D$, then $X_l = \{Role_A, Role_B \wedge Role_C\}$, $X_r = \{Role_A, Role_D\}$, and $f(\Upsilon_l, \Upsilon_r) = 1$. To find the hyperplane, instead of enumerating all the possible choices, we develop a dynamic programming algorithm (see Algorithm 7 in Appendix D). It has a time complexity of $O(n)$, where n is the interval length of the query attribute in the splitting dimension. Further, to prevent the tree imbalance, we propose to switch back to the AP^{2}G-tree split strategy when the tree depth goes beyond $\log(S)$, where S is the area of the index space. Figure 6b shows the corresponding AP^2kd-tree for the data records in Figure 6a. The signature definitions for the tree nodes are identical to those in the AP^{2}G-tree.

The rest of the algorithm for authenticating range queries and join queries with the access policy confidentiality is the same as that in Section 6. Similar to the example shown in Figure 4, if the records o_{10}, o_{11}, o_{12}, o_{14}, o_{15} and o_{16} share the same access policy, the AP^2kd-tree will be like what is presented in Figure 6. Under the same query range $[\alpha, \beta]$, the SP will return the records o_6, o_7, and o_8, as well as their APP signatures. Moreover, the SP will return the APS signature for N_6 as part of the VO. During result verification, the user will check whether or not all of the signatures in the VO are valid and whether or not the union of the indexing spaces for N_6, o_6, o_7, and o_8 covers the whole query range $[\alpha, \beta]$.

9.2 Supporting Continuous Query Attributes

When dealing with discrete query attribute values, we treat non-existent records as pseudo records with access policy $Role_\emptyset$. For continuous query attribute values, this method is also applicable since they can be converted into discrete values by discretization

techniques [13]. Nevertheless, when the zero-knowledge confidentiality is not required, it is acceptable to disclose to the user about the distribution of data records. As such, the DO can view non-existent records as pseudo regions with access policy $Role_{\emptyset}$. For example in Figure 2, if the query attribute is continuous, the DO will generate the APP signatures for the regions $(-\infty, o_1)$, (o_1, o_2), (o_2, o_3), (o_3, o_4), (o_4, o_5), and $(o_5, +\infty)$ with policy $Role_{\emptyset}$, which will then be outsourced to the SP. When processing an equality query, if the query attribute is located in one of the pseudo regions, the SP will apply the ABS.Relax algorithm to compute the APS signature for the corresponding region and use it as the VO. Otherwise, the procedure is the same as the original algorithm introduced in Section 5. Similarly, for a range or join query, the APS signatures for these pseudo regions that intersect the query range will be constructed as part of the VO. The rest of the algorithm is the same as those described in Section 6.

10 PERFORMANCE EVALUATION

In this section, we report our empirical study. Since no prior methods can support zero-knowledge query authentication under fine-grained access control, we mainly evaluate the impact of various system settings on our proposed solutions. We use *TPC Benchmark H* (TPC-H) [6] to generate the databases for testing. Specifically, we choose the `Lineitem` table as the data source and $Q6$ from TPC-H[6] as a basic query. The records in the database contain 12 attributes, among them the first three are set as query attributes. They are in the form of $\langle shipdate, discount, quantity \rangle$. Our experiments test four different scales: 0.1 (600,000 records), 0.3 (1,800,000 records), 1 (6,000,000 records), and 3 (18,000,000 records). The default scale is 0.3. For access policies, we randomly generate them as DNF boolean functions with three parameters: (i) total number of distinct policies, (ii) total number of distinct roles, and (iii) maximum policy length. By default, the total number of roles is set at 10. We generate 10 distinct policies whose root gate is an OR gate with at most three inputs, while each input is an AND gate with at most two roles. This yields a maximum policy length of 6. We assign these policies such that the records under the same query key share the same access policy and are accessed together.

In the experiments, both the DO and SP are set up on an x64 blade server with dual Intel Xeon 2.67GHz X5650 CPU and 32GB RAM running on CentOS 6. The user, on the other hand, is set up on a commodity laptop computer with Intel Core i5 CPU and 4GB RAM, running on macOS Sierra. This enables both the DO and SP to run experiments with 24 hyper-threads and the user to run with 4 hyper-threads. The experiments are written in C++ and use the following libraries: Pairing-based cryptographic for bilinear pairing computation, Crypto++ for secure hash operations, and OpenMP for parallel computation.

Table 1 shows the DO setup overhead for generating the AP^{2}G-tree under the different database scales. For the DO CPU time, we break down the cost into two parts: (i) the time to sign all the APP signatures, and (ii) the time to build the index structure. It can be observed that both the DO CPU time and space cost increase only sublinearly with respect to the growth of database scale.

To evaluate the query performance, we measure two kinds of costs: (i) the computational cost in terms of the SP CPU time and user CPU time, and (ii) the communication cost in terms of the VO size. The results are reported based on an average of 10 randomly generated queries. Note that when measuring the computational cost, we ignore the cost of CP-ABE/AES encryption and decryption, since it is not a critical part of our proposed algorithms and has the performance largely subject to the size of a record.

[6] `SELECT * FROM lineitem WHERE shipdate between '?' AND '?' AND discount between '?' AND '?' AND quantity between '?' AND '?'.`

Table 1: DO Setup Overhead

Database Scale	DO CPU Time		Index Size (Tree Structure + Signatures) (GB)
	Sign APPs (h)	Build Index (h)	
0.1	0.63	0.74	2.47 (0.49 + 1.98)
0.3	0.77	0.95	2.93 (0.56 + 2.37)
1	0.86	1.00	3.14 (0.58 + 2.56)
3	0.87	1.01	3.16 (0.59 + 2.57)

Table 2: Equality Query Performance

Max Policy Length	Accessible Record		Inaccessible Predicate Length	Inaccessible Record		
	User CPU Time (ms)	VO Size (KB)		SP CPU Time (ms)	User CPU Time (ms)	VO Size (KB)
6	33.6	0.9	10	95.8	46.5	1.8
24	143	1.8	20	163	72.7	3.1
96	664	6.6	40	301	128	5.6
384	2,384	23.5	80	575	238	10.8

10.1 Equality Query Performance

In the first set of experiments, we test equality queries, whose performance is essentially the same as the underlying single ABS operation. We vary the max policy length (which affects the costs when the record is accessible) and the inaccessible predicate length (which affects the costs when the record is inaccessible). The results are summarized in Table 2. Note that we omit the SP CPU time for accessible records since in this case the SP simply returns the APP signature signed by the DO and has little computational cost. For all other reported measures, as expected, the costs in CPU time and VO size are proportional to the policy/predicate length.

10.2 Range and Join Query Performance

To evaluate the range query authentication performance, we compare two methods: (i) the basic approach in which equality query authentication is executed repeatedly for every discrete value lying in the query range, and (ii) the AP^{2}G-tree approach developed in Section 6. In each query, the user is assigned with the roles that can access 20% of the data records.

We first vary the query range from 0.03% to 1% of the data space under the default settings. As shown in Figure 7, the AP^{2}G-tree outperforms the basic approach in all metrics. This indicates that the APS signatures generated for AP^{2}G-tree nodes can effectively summarize the inaccessible records in their subtrees. This leads to the reduction in both computation and communication overheads.

To investigate the impact of the database scale and access policies, we fix the query range at 0.1%. Figure 8 shows the results when the database scale is varied from 0.1 to 3. All metrics increase monotonically under AP^{2}G-tree, but those of the basic approach fluctuate a bit. This can be explained as follows. With more records, the access policies become more complex. This affects the costs in two ways: (i) it takes the SP more time to process the ABS.Relax operations, and (ii) it takes the user more time to verify the APP signatures. On the other hand, with the increase of database scale, more records can be accessed by the user and, hence, the inaccessible records become fewer. This decreases the costs on both the SP and the user. For AP^{2}G-tree, owing to its high pruning power, the decreased number of inaccessible records has less impact on the costs. Therefore, its costs increase steadily.

Figures 9 and 10 show the performance trends with the respect

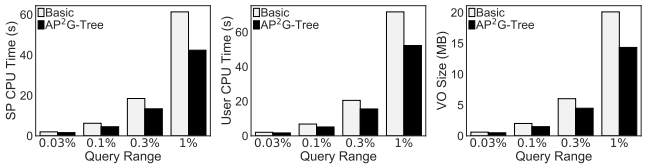


Figure 7: Range Query Performance vs. Range

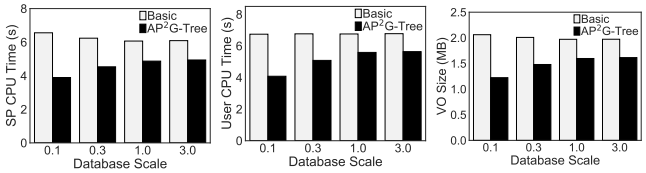


Figure 8: Range Query Performance vs. Database Scale

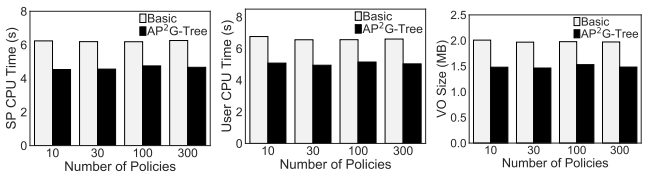


Figure 9: Range Query Performance vs. Policy Diversity

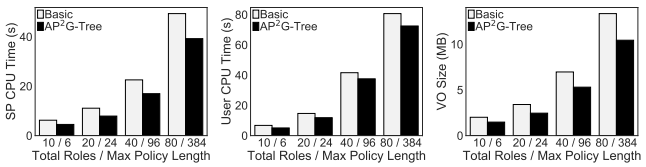


Figure 10: Range Query Performance vs. Total Roles

to the number of distinct policies and the total number of roles/max policy length, respectively. It can be seen that the performance remains almost the same under different policy diversities. However, the larger the role space and the longer access policy length, the higher the overhead incurred for both the computation and communication costs.

Finally, to study the join query performance, we evaluate the join operator of $Q12$ in TPC-H,[7] which joins the tables `Lineitem` and `Orders` on the attribute *orderkey*. Figure 11 reports the results while varying the query range. It is shown that the costs in all metrics under AP^{2}G-tree are substantially lower than those in the basic approach.

10.3 Impact of the Optimizations

We now investigate the impact of different optimization techniques proposed in Section 8. The results of using hierarchical role assignment are shown in Figure 12. In our experiment, we simulate a two-level role hierarchy. Two global hierarchical roles are created and attached randomly to each AND gate in all the access policies. As a result, the average size of a user's inaccessible predicate is decreased from 9 to 6. It can be observed that the hierarchical role assignment reduces the cost in all metrics, due to much less overhead in processing inaccessible records.

To study the acceleration acquired by parallelism, we vary the number of threads for both the SP processing and user verification running on the blade server. As shown in Figure 13, more acceleration can be observed with the first 16 threads, but it becomes

[7] `SELECT * FROM orders, lineitem WHERE o.orderkey = l.orderkey AND l.orderkey between '?' AND '?'.`

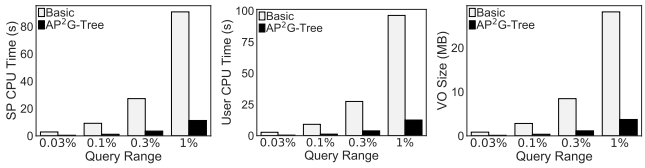


Figure 11: Join Query Performance vs. Range

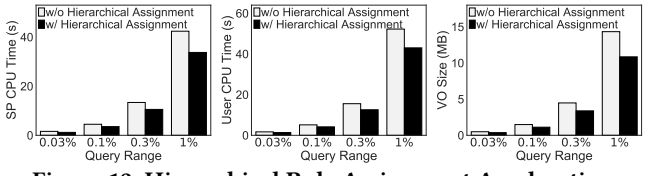


Figure 12: Hierarchical Role Assignment Acceleration

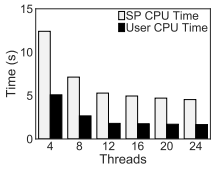


Figure 13: Multi-Threaded Acceleration

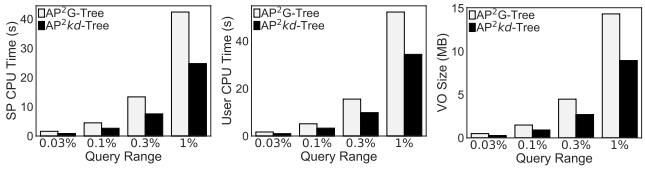


Figure 14: Range Query Performance vs. Index

less effective with more threads. The reason is that the non-parallel part of the algorithm and I/O operations becomes more pronounced after sufficient multi-thread acceleration.

Finally, we study the performance gain if we relax the zero-knowledge confidentiality requirement. As shown in Figure 14, owing to a careful splitting strategy introduced in Section 9.1, AP^2*kd*-tree substantially outperforms AP^{2}G-tree in all evaluation metrics.

11 CONCLUSION

In this paper, we have studied the problem of authenticating relational queries with fine-grained access control. We developed a new variant of the attribute-based signature scheme, which supports predicate relaxation on super access policies. Based on that, we proposed a novel access-policy-preserving (APP) signature as the primitive authenticated data signature to authenticate various types of queries under fine-grained access control. Our approach is zero-knowledge and reveals nothing beyond the accessible records. We also designed a grid-index-based AP^{2}G-tree to improve the performance of processing range queries and join queries. Optimization techniques for both the zero-knowledge model and the access policy confidentiality model were developed. Analytical models and empirical results substantiated the robustness and efficiency of our proposed solutions.

As for future work, we plan to extend the proposed techniques to support more complex queries, such as aggregation. We also plan to study more challenging fine-grained query authentication problems for multi-source data in a distributed environment.

Acknowledgments. This work was supported by Research Grants Council of Hong Kong under GRF Projects 12244916, 15238116, 12202414, 12200914, CRF Project C1008-16G, and National Natural Science Foundation of China under 61572413 and U1636205.

REFERENCES

[1] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* (1975).
[2] John Bethencourt, Amit Sahai, and Brent Waters. 2007. Ciphertext-Policy Attribute-Based Encryption. In *Proc. of IEEE Symposium on Security and Privacy*.
[3] Hong Chen, Xiaonan Ma, Windsor W. Hsu, Ninghui Li, and Qihua Wang. 2008. Access Control Friendly Query Verification for Outsourced Data Publishing. In *ESORICS*.
[4] Qian Chen, Haibo Hu, and Jianliang Xu. 2014. Authenticating Top-k Queries in Location-based Services with Confidentiality. In *Proc. VLDB*.
[5] Qian Chen, Haibo Hu, and Jianliang Xu. 2015. Authenticated Online Data Integration Services. In *Proc. SIGMOD*.
[6] Transaction Processing Performance Council. 2017. TPC Benchmark H. http://www.tpc.org/tpch/. (2017).
[7] Benjamin C. M. Fung, Ke Wang, Rui Chen, and Philip S. Yu. 2010. Privacy-Preserving Data Publishing: A Survey of Recent Developments. *Comput. Surveys* (2010).
[8] Esha Ghosh, Olga Ohrimenko, and Roberto Tamassia. 2016. Efficient Verifiable Range and Closest Point Queries in Zero-Knowledge. *PETS* (2016).
[9] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. 2006. Attribute-based encryption for fine-grained access control of encrypted data.. In *CCS*.
[10] C. Guo, R. Zhuang, Y. Jie, R. Ren, T. Wu, and K. R. Choo. 2016. Fine-grained Database Field Search Using Attribute-Based Encryption for E-Healthcare Clouds. *J Med Syst.* (2016).
[11] Haibo Hu, Jianliang Xu, Qian Chen, and Ziwei Yang. 2012. Authenticating location-based services without compromising location privacy. In *Proc. SIGMOD*.
[12] Rohit Jain and Sunil Prabhakar. 2013. Access Control and Query Verification for Untrusted Databases. In *Proc. DBSec*.
[13] S. Kotsiantis and D. Kanellopoulos. 2006. Discretization Techniques: A recent survey. *GESTS International Transactions on Computer Science and Engineering* (2006).
[14] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. 2010. Authenticated Index Structures for Aggregation Queries. *ACM TISSEC* (2010).
[15] F. Li, G. Kollios, and L. Reyzin. 2006. Dynamic Authenticated Index Structures for Outsourced Databases. In *Proc. SIGMOD*.
[16] Jin Li, Man Ho Au, Willy Susilo, Dongqing Xie, and Kui Ren. 2010. Attribute-based signature and its applications. In *ASIACCS*.
[17] Jingwei Li, Anna Cinzia Squicciarini, Dan Lin, Smitha Sundareswaran, and Chunfu Jia. 2017. MMB^{cloud}-Tree: Authenticated Index for Verifiable Cloud Service Selection. *TDSC* (2017).
[18] Zhen Liu, Zhenfu Cao, and Duncan S Wong. 2010. *Efficient generation of linear secret sharing scheme matrices from threshold access trees.* Technical Report. IACR Cryptology ePrint Archive.
[19] Hemanta K Maji, Manoj Prabhakaran, and Mike Rosulek. 2011. Attribute-Based Signatures. In *Topics in Cryptology*.
[20] Ralph C Merkle. 1989. A Certified Digital Signature. In *CRYPTO*.
[21] V Nikov and S Nikova. 2004. *New Monotone Span Programs from Old.* Technical Report.
[22] H. Pang, A. Jain, K. Ramamritham, and K. L. Tan. 2005. Verifying Completeness of Relational Query Results in Data Publishing. In *Proc. SIGMOD*.
[23] H. Pang and K. Mouratidis. 2008. Authenticating the query results of text search engines. In *Proc. VLDB*.
[24] H. Pang and K.-L. Tan. 2004. Authenticating query results in edge computing. In *Proc. ICDE*.
[25] Sushmita Ruj, Milos Stojmenovic, and Amiya Nayak. 2012. Privacy Preserving Access Control with Authentication for Securing Data in Clouds.. In *CCGRID*.
[26] Amit Sahai and Brent Waters. 2004. Fuzzy Identity-Based Encryption. In *EUROCRYPT*.
[27] Salesforce. 2017. Increasing the Maximum number of Roles or Territories. https://goo.gl/KDtMx5. (2017).
[28] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. 1996. Role-Based Access Control Models. *IEEE Computer* (1996).
[29] Muhammad I Sarfraz, Mohamed Nabeel, Jianneng Cao, and Elisa Bertino. 2015. DBMask: Fine-Grained Access Control on Encrypted Relational Databases. In *Proc. CODASPY*.
[30] Muhammad I Sarfraz, Mohamed Nabeel, Jianneng Cao, and Elisa Bertino. 2016. DBMask: Fine-Grained Access Control on Encrypted Relational Databases. *TDP* (2016).
[31] Michael G. Solomon, Vaidy Sunderam, and Li Xiong. 2014. Towards Secure Cloud Database with Fine-Grained Access Control. In *Proc. DBSec*.
[32] Wenhai Sun, Shucheng Yu, Wenjing Lou, Y Thomas Hou, and Hui Li. 2016. Protecting Your Right: Verifiable Attribute-Based Keyword Search with Fine-Grained Owner-Enforced Search Authorization in the Cloud. *TPDS* (2016).
[33] Cheng Xu, Qian Chen, Haibo Hu, Jianliang Xu, and Xiaojun Hei. 2017. Authenticating Aggregate Queries over Set-Valued Data with Confidentiality. *TKDE* (2017).
[34] G. Yang, Y. Cai, and Z. Hu. 2016. Authentication of Function Queries. In *Proc. ICDE*.
[35] Yin Yang, Dimitris Papadias, Stavros Papadopoulos, and Panos Kalnis. 2009. Authenticated join processing in outsourced databases. In *Proc. SIGMOD*.
[36] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. 2009. Authenticated indexing for outsourced spatial databases. *VLDBJ* (2009).
[37] M. L. Yiu, E. Lo, and D. Yung. 2011. Authentication of Moving kNN Queries. In *Proc. ICDE*.
[38] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2015. IntegriDB: Verifiable SQL for Outsourced Databases. In *CCS*.

A ADDITIONAL ALGORITHMS FOR ABS

A.1 Monotone Span Program

Algorithm 5: Build Monotone Span Program

Function `BuildMSP(`*expr*`)`
  Input: Boolean Function $expr$
  Output: Span Program $\mathbf{M}$, Row Labels L
  if $expr$ **is** *label* **then**
    $\mathbf{M} \leftarrow (1)$, $L \leftarrow [expr]$;
  else
    $n \leftarrow \text{len}(expr.children)$;
    switch $expr$ *type* **do**
      case *AND operator* **do**
        $$\mathbf{M} \leftarrow \underset{n\times n}{\begin{pmatrix} 1 & -1 & \cdots & -1 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}};$$
      case *OR operator* **do**
        $$\mathbf{M} \leftarrow \underset{n\times 1}{(1, \ldots, 1)^T};$$
    $i \leftarrow 1$, $L \leftarrow []$;
    for e **in** $expr.children$ **do**
      $\mathbf{M}_e, L_e \leftarrow$ `BuildMSP(`e`)`;
      $$\mathbf{M} \leftarrow \begin{pmatrix} \mathbf{M}[1:i-1,:] & 0 \\ \mathbf{M}_e[:,1] \cdot \mathbf{M}[i,:] & \mathbf{M}_e[:,2:] \\ \mathbf{M}[i+1:,:] & 0 \end{pmatrix};$$
      $L \leftarrow L + L_e$, $i \leftarrow i + \#\text{ of rows}(\mathbf{M}_e)$;
  return $\langle \mathbf{M}, L \rangle$;

A.2 ABS Predicate Relaxation Purge Step

The detailed algorithm for the purge step in ABS predicate relaxation is given in Algorithm 6. The intuition behind this step is to solve a system of linear equations $\mathbf{Ax} = \mathbf{b}$. Here, we treat the original monotone span program for the input signature predicate Υ as $\mathbf{A}$ and the new monotone span program for $\vee_{i \in kept\_rows} u(i)$ as $\mathbf{b}$. The solution of this system $\mathbf{x}$ will yield $\tilde{P}_1$ from a linear composition of the original P_j, i.e., $\tilde{P}_1 = \prod_{j=1}^{t} P_j^{x_j}$. However, thanks to the monotone span program in Algorithm 5, we do not need to solve this linear system directly. Instead, the solution $\mathbf{x}$ can be obtained from the tree traversal in Algorithm 6 as follows:

$$\mathbf{x} = [x_j], x_j = \begin{cases} 1 & \text{if } j \in kept\_cols, \\ 0 & \text{otherwise.} \end{cases}$$

B PROOF OF THEOREM 7.3

Theorem 7.3. *The construction of Section 5.2 satisfies the security properties of Perfect Privacy (Definition 7.1) and Unforgeability (Definition 7.2) in the generic group model.*

Algorithm 6: ABS.Relax Purging Step

Function Purge($expr$, $\mathcal{A}$)
  Input: Boolean Function $expr$, kept attribute set $\mathcal{A}$
  Output: Monotone Span Program size $\langle rows, cols\rangle$, Row Labels L, Rows and Columns in MSP to be kept $\langle kept\_rows, kept\_cols\rangle$, Whether to keep this node $flag$
  if $expr$ **is** *label* **then**
    $rows \leftarrow 1$,$cols \leftarrow 1$,$kept\_rows \leftarrow [1]$,$kept\_cols \leftarrow [1]$;
    $L \leftarrow [expr]$;
    if $expr$ **in** $\mathcal{A}$ **then** $flag \leftarrow true$ **else** $flag \leftarrow false$;
  else
    $n \leftarrow$ len($expr.children$);
    switch $expr$ *type* **do**
      case *AND operator* **do**
        $rows \leftarrow n$, $cols \leftarrow n$, $flag \leftarrow false$;
      case *OR operator* **do**
        $cols \leftarrow n$, $cols \leftarrow 1$, $flag \leftarrow true$;
    $i \leftarrow 1$, $L \leftarrow []$;
    for $k = 1$ **to** n **do**
      $e \leftarrow expr.children[k]$;
      $\langle rows_e, cols_e, L_e, kept\_rows_e, kept\_cols_e, flag_e\rangle \leftarrow$ Purge(e, $\mathcal{A}$);
      $L \leftarrow L + L_e$;
      for r **in** $kept\_rows_e$ **do** $r \leftarrow r + i$;
      for c **in** $kept\_cols_e$ **do**
        if $c \neq 1$ **then** $c \leftarrow c + cols - 1$;
      switch $expr$ *type* **do**
        case *AND operator* **do**
          if $flag_e$ **is** $true$ **then**
            $kept\_rows \leftarrow kept\_rows_e$;
            $kept\_cols \leftarrow kept\_cols_e$;
            if $k \neq 1$ **then** $kept\_cols \leftarrow kept\_cols + [k]$;
          $flag \leftarrow flag \vee flag_e$
        case *OR operator* **do**
          $kept\_rows \leftarrow kept\_rows + kept\_rows_e$;
          $kept\_cols \leftarrow kept\_cols + (kept\_cols_e - [1])$;
          $flag \leftarrow flag \wedge flag_e$
      $rows \leftarrow rows + rows_e - 1$, $cols \leftarrow cols + cols_e - 1$;
      $i \leftarrow i + rows_e$;
  return $\langle rows, cols, L, kept\_rows, kept\_cols, flag\rangle$;

The theorem is proved by showing our ABS scheme satisfies each security property.

B.1 Proof of Perfect Privacy

Lemma B.1. *Our scheme has perfect privacy according to Definition 7.1.*

Proof. The first requirement of Definition 7.1 is trivial considering the scheme is derived from [19]. It is also easy to verify that the distributions of the output signatures from ABS.Sign and ABS.Relax are identical owing to the re-randomization step in ABS.Relax, which proves the second requirement. □

B.2 Proof of Unforgeability

Lemma B.2. *Our scheme is unforgeable according to Definition 7.2 in the generic group model.*

Proof. The generic group model is designed to model the behavior of any algorithm that does not exploit the algebraic structure of the underlying groups. In other words, the lemma guarantees that our proposed scheme is unforgeable against any generic attacker, i.e., the attacker that does not exploit the underlying group structures. We remark that the scheme in [19] is also analyzed in this setting.

The idea of the generic group model is briefly reviewed here. Let p be a prime which represents the group order. A generic group $\mathbb{G}_i$ can be represented as the set $\{\xi_i(x)|x \in \mathbb{Z}_p\}$. The group operations are modeled by two oracles, namely, O_1, O_2. Specifically, on input two group elements $\xi_i(a), \xi_i(b)$, oracle O_i returns $\xi_i(a+b)$ and $\xi_i(a-b)$ respectively for multiplication and division. To model the bilinear pairing, another oracle O_E is defined. On input elements $\xi_1(a), \xi_1(b)$, O_E returns $\xi_2(ab)$. Given $\xi_i(x)$ and scalar s, it is possible to obtain $\xi_i(sx)$ (via $O(\log s)$ calls to oracle O_i).

Since it only benefits the attacker, we assume the groups $\mathbb{G}$ and $\mathbb{H}$ are the same. The groups $\mathbb{G}, \mathbb{G}_T$ of the scheme are modeled as generic groups $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively. The attacker works with the group elements and can only compute group operations by interacting with oracles O_i and O_E. As the encoding functions ξ_i are modeled as random functions, nothing about the element except equality can be inferred. The security proof is completed by showing that given the encodings of the public key and the signatures from the signing oracle, it is infeasible to output the encodings needed to satisfy the verification equation for the forgery with a bounded number of queries to oracles O_i and O_E.

We first define how signature queries are answered. For span program $\mathbf{M} \in \mathbb{Z}_p^{\ell \times t}$ with row labeling function u, pick random $\tau, y, s_1, \cdots, s_\ell$, compute p_j for $j = 1$ to t, as follows:

$$p_j = \frac{1}{c + \mathsf{hash}}\Big(\sum_{i=1}^{\ell} s_i(a + u(i)b)\mathbf{M}_{ij} - yz_j\Big),$$

where $(z_1, \cdots, z_\ell) = (1, 0, \cdots, 0)$ and $\mathsf{hash} = hash(\tau, m)$.

The signature is parsed as $\sigma = (\tau, g^y, g^{y/a_0}, g^{s_1}, \cdots, g^{s_\ell}, h^{p_1}, \cdots, h^{p_t})$. It is easy to show that the signature generated is of the same distribution as that of a signature created using the sign algorithm.

The adversary is given $\xi_1(1), \xi_1(\Delta_0), \xi(a_0), \xi_1(\Delta), \xi_1(a\Delta), \xi_1(b\Delta), \xi_1(c)$ as the public parameters. They represent group elements (g, h_0, A_0, h, A, B, C), i.e., mvk, of the scheme.

Let Q be the number of signature queries made by the adversary, and ℓ_q, t_q be the length and width of the associated span program for the q-th query. In the q-th query, the adversary is given $\tau^{(q)}$ and the encodings of the following values: $\{s_i^{(q)}\}_{i=1}^{\ell_q}, \{p_j^{(q)}\}_{j=1}^{t_q}$, $w^{(q)}$, $y^{(q)}$, satisfying the conditions that $y^{(q)} \neq 1$, $w^{(q)} = y^{(q)}/a_0$ and for $j = 1$ to t_q:

$$y^{(q)} z_j \Delta + (c + \mathsf{hash}^{(q)}) p_j^{(q)} = \sum_{i=1}^{\ell_q} s_i^{(q)} \mathbf{M}_{ij}^{(q)} (a + u^{(q)}(i)b)\Delta,$$

where $\mathsf{hash}^{(q)} = hash(\tau^{(q)}, m^{(q)})$ and $(z_1, \cdots, z_{t_q}) = (1, 0, \cdots, 0)$.

Finally, the adversary outputs a forgery $\sigma^* := (\tau^*, \xi_1(y^*), \xi_1(w^*), \{\xi_1(s_i^*)\}_{i=1}^{\ell^*}, \{\xi_1(p_j^*)\}_{j=1}^{t^*})$ on message m^* with span program $\mathbf{M} \in \mathbb{Z}_p^{\ell^* \times t^*}$ and labeling function u^*.

Denote by hash^* the value $hash(\xi_1(y^*), \xi_1(w^*), \tau^*, m^*)$. To be a valid forgery, $y^* \neq 0$, $w^* = y^*/a_0$ and for $j = 1$ to t^*, the following equation holds:

$$y^* z_j \Delta + (c + \mathsf{hash}^*) p_j^* = \sum_{i=1}^{\ell^*} s_i^* \mathbf{M}_{ij}^* (a + u^*(i)b)\Delta.$$

For notational convenience, let $\mathsf{Lin}(S)$ denote the sets of functions

that are linear in the terms in set S. Let $\mathsf{Hom}(S)$ be the subset of $\mathsf{Lin}(S)$ of homogeneous functions whose constant coefficient is zero. In the generic group model, a valid encoding can only be received from the oracles. Specifically, all encodings presented by the adversary in the generic group $\mathbb{G}_1$ must be linear combinations of the previously obtained element in $\mathbb{G}_1$. That is, $\sigma^* \backslash \{\tau^*\} \subset \mathsf{Lin}(\Gamma)$, where Γ is:

$$\{1, a_0, \Delta_0, \Delta, a\Delta, b\Delta\} \cup \{\{s_i^{(q)}\}_{i=1}^{\ell_q}, \{p_j^{(q)}\}_{j=1}^{t_q}, w^{(q)}, y^{(q)}\}_{q=1}^{Q}.$$

The remaining part of the proof is to show that $\sigma^* \setminus \{\tau^*\}$ is not a subset of $\mathsf{Lin}(\Gamma)$ when we view the terms as functions in the random variables used in the security game. To complete the proof, we consider two cases.

Case 1: $(\tau^*, m^*) \neq (\tau^{(q)}, m^{(q)})$ for all q. Firstly, $y^* = w^* a_0$. Thus,

$$y^* \in \mathsf{Hom}(\{\Delta_0 a_0, y^{(1)}, \cdots, y^{(Q)}\}).$$

Next, it can be seen that $\Delta \mid p_j^*$. Thus,

$$p_j^* \in \mathsf{Hom}(\{\Delta, \Delta a, \Delta b\} \cup \{\{p_1^{(q)}, \cdots, p_{t_q}^{(q)}\}_{q=1}^{Q}\}).$$

Since $\exists j$ s.t. $z_j \neq 0$, and given the form of p_j^*, it can be seen that y^* cannot contain a term from $\Delta_0 a_0$. Thus,

$$y^* \in \mathsf{Hom}(\{y^{(1)}, \cdots, y^{(Q)}\}).$$

Next, we argue that p_j^* cannot contain a single term Δ. For if it is the case, the left-hand side contributes monomials $\mathsf{hash}^*\Delta$ (since y^* is homogeneous and has no constant term). On the other hand, the right-hand side cannot contribute monomials in Δ (everything in the right-hand side has either a or b). Thus,

$$p_j^* \in \mathsf{Hom}(\{\Delta a, \Delta b\} \cup \{\{p_1^{(q)}, \cdots, p_{t_q}^{(q)}\}_{q=1}^{Q}\}).$$

Now, p_j^* cannot contain a $p_n^{(q)}$ term for any n, q. Otherwise, it will produce a term with $\frac{c+\mathsf{hash}^*}{c+\mathsf{hash}^{(q)}}$ on the left and no setting of y^*, s_i^* can come up with this rational term. In other words,

$$p_j^* \in \mathsf{Hom}(\{\Delta a, \Delta b\}).$$

Finally, we arrive at a contradiction since no combination of $(c + \mathsf{hash}^*)p_j^*$ and $\sum_{i=1}^{\ell^*} s_i^* \mathbf{M}_{ij}^* (a + u^*(i)b)\Delta$ could contribute to a term of the form Δy^* for some $y \in \mathsf{Hom}(\{y^{(1)}, \cdots, y^{(Q)}\})$.

Case 2: $(\tau^*, m^*) = (\tau^{(k)}, m^{(k)})$ for some k. Following the analysis above, the following constraints can be reached:

$$y^* \in \mathsf{Hom}(\{y^{(1)}, \cdots, y^{(Q)}\}),$$

$$p_j^* \in \mathsf{Hom}(\{\Delta a, \Delta b\} \cup \{\{p_1^{(q)}, \cdots, p_{t_q}^{(q)}\}_{q=1}^{Q}\}).$$

Since $\tau^{(q)}$ is randomly chosen for each signature query, $\tau^{(k)} \neq \tau^{(q)}$ when $k \neq q$. Since *hash* is collision-resistant, it means $\mathsf{hash}^* \neq \mathsf{hash}^{(q)}$ for all $q \neq k$. Thus, p_j^* cannot contain a term from $p_n^{(q)}$ when $q \neq k$. Otherwise, there will be a term $\frac{c+\mathsf{hash}^*}{c+\mathsf{hash}^{(q)}}$ on the left and no setting of y^*, s_i^* can come up with this rational term. Thus,

$$p_j^* \in \mathsf{Hom}(\{\Delta a, \Delta b, p_1^{(k)}, \cdots, p_{t_k}^{(k)}\}).$$

Next, y^* cannot contain a term from $y^{(q)}$ for $q \neq k$ since all terms on the right-hand side contain either a or b and that $(c + \mathsf{hash}^*)p_j^*$ can only provide a term to cancel $y^{(k)}$. Thus,

$$y^* \in \mathsf{Hom}(y^{(k)}).$$

In this case, $m^* = m^{(k)}$ and thus the adversary wins if and only if Υ^* represents a more restricted policy compared with $\Upsilon^{(k)}$. Assume the DNF representation of policy $\Upsilon^{(k)}$ is $\mathcal{P} := \vee_{x=1}^{n}(\mathcal{P}_x)$, where each P_x is of the form $(\wedge_{y=1}^{n_x}(a_{xy}))$. In general, a more restricted policy can be formed by removing one clause from $\mathcal{P}$ or adding more attributes in one of $\mathcal{P}_x$.

In our threat model, the forgery must be of the form $(\vee_{a \in \mathcal{A}'} a)$ for some $\mathcal{A}' \subset \mathbb{A}$, where $\mathbb{A}$ represents the attribute universe. A successful forgery means that at least one clause, $\mathcal{P}_x$, is removed and that all attributes in $\mathcal{P}_x$ does not appear in $\mathcal{A}'$. The corresponding span program is $\mathbf{M} \in \mathbb{Z}_p^{|\mathcal{A}'| \times 1}$ and the labeling function maps each row to one attribute in $\mathcal{A}'$.

It means that the forgery in this case satisfies $t^* = 1$, and that

$$y^* z\Delta + (c + \mathsf{hash}^*)p^* = \sum_{i=1}^{\ell^*} s_i^*(a + u^*(i)b)\Delta.$$

Assume $\mathcal{P}_x$ is the clause that has been removed. We use $\bar{\mathcal{A}}$ to denote the attributes of $\mathcal{P}_x$. Thus, $u^*(i) \notin \bar{\mathcal{A}}$. However, each $p_j^{(k)}$ has at least one term with $a + u^{(k)}(i)b$ with $u^{(k)}(i) \in \bar{\mathcal{A}}$. WLOG, assume $\bar{\mathcal{A}} := \{1, 2, \cdots, x\}$ and $\mathcal{P}_x$ is the first clause. Since $\mathcal{P}_x$ is a conjunctive clause, $p_1^{(k)}$ contains the term $(a + b)$, $p_2^{(k)}$ contains $(a + b)$ and $(a + 2b)$, $p_3^{(k)}$ contains $(a + b)$ and $(a + 3b)$, etc. If p^* contains any of these $p_j^{(k)}$, there will be a term $(a + ub)$, $u \in \bar{\mathcal{A}}$, which appears on the left-hand side but not the right-hand side (note that $y^* = y^{(k)}$ and thus $y^* z\Delta$ cannot be used to cancel this $(a + ub)$ term. Further, note that linear combinations of $p_j^{(k)}$ cannot cancel all terms in the form of $(a + bu)$ for some $u \in \bar{\mathcal{A}}$). Thus,

$$p^* \in \mathsf{Hom}(\{\Delta a, \Delta b\}).$$

Again, we arrive at a contradiction since no combination of $(c + \mathsf{hash}^*)p^*$ and $\sum_{i=1}^{\ell^*} s_i^*(a + u^*(i)b)\Delta$ could contribute to a term of the form Δy^* for some $y \in \mathsf{Hom}(\{y^{(k)}\})$. □

C PROOF OF THEOREM 7.6

Theorem 7.6. *Our proposed query authentication algorithms satisfy the security properties of Unforgeability (Definition 7.4) and Zero-Knowledge (Definition 7.5).*

This theorem is proved by showing that our proposed query authentication algorithms satisfy each security property.

C.1 Proof of Unforgeability

Lemma C.1. *Our proposed query authentication algorithms are unforgeable according to Definition 7.4.*

Proof. We prove this lemma by contradiction.

Case 1: The result set *RS* contains a data record $\langle o^*, v^*, \Upsilon^* \rangle$, such that $\langle o^*, v^*, \Upsilon^* \rangle \neq \langle o_i, v_i, \Upsilon_i \rangle$, $\forall i$.

Recall that in the result verification procedure, the verifier will run ABS.Verify$(mvk, hash(o^*))|hash(v^*), \Upsilon^*, \sigma^*)$. Since it passes the verification and σ^* can only be generated by the adversary, this means that the adversary is able to forge an ABS signature σ^*, which contradicts to Theorem 7.3.

Case 2: The result set *RS* contains a data record $\langle o^*, v^*, \Upsilon^* \rangle$, such that o^* does not satisfy the query q or $\Upsilon^*(\mathcal{A}) = 0$.

It is trivial to see that such a case is impossible, as the verifier will check whether or not o^* satisfies the query q and $\Upsilon^*(\mathcal{A}) = 1$ during the result verification procedure.

Case 3: There exists a data record $\langle o_j, v_j, \Upsilon_j \rangle$ not in *RS*, such that

$j \in \{i\}$, o_j satisfies the query q, and $\Upsilon_j(\mathcal{A}) = 1$.

There are two possible subcases. The first subcase is that the index key of the missing record j is not returned as part of the VO. This is impossible as the verifier will check whether or not the union of the indexing space for each entry in the VO covers the query range of q. The second subcase is that the index key of the missing record j is returned as part of the VO. In this case, the record j must fall in the space of an APS signature in the VO. Note that this APS signature can only be generated by the adversary, given its corresponding APP signature whose predicate is a more restricted policy compared with $\vee_{a \in \mathbb{A} \setminus \mathcal{A}} a$. This means that the adversary is able to forge an ABS signature, which contradicts to Theorem 7.3. □

C.2 Proof of Zero-Knowledge

Lemma C.2. *Our proposed query authentication algorithms are zero-knowledge according to Definition 7.5.*

Proof. It is easy to see that the messages output to the adversary from the ideal game will have the same distribution as those from the real game due to the following reasons: (i) all the messages generated by the simulator are generated using the same algorithms as those by the real challenger; (ii) all ABS signatures achieve the prefect privacy (according to Theorem 7.3); and (iii) the AP^{2}G-tree structure generated by the simulator is identical to the one by the real challenger. □

D AP^2kD-TREE SPLIT ALGORITHM

Algorithm 7: AP^2kd-Tree Split

Function Split($\{\Upsilon_1, \ldots, \Upsilon_n\}$)
  Input: Access Policies $\{\Upsilon_1, \ldots \Upsilon_n\}$
  Output: Hyperplane $x = \arg\min_x f(\Upsilon_1 \vee \cdots \vee \Upsilon_x, \Upsilon_{x+1} \vee \cdots \vee \Upsilon_n)$
  for $i = 1$ **to** n **do**
    convert Υ_i to DNF, i.e., $\Upsilon_i = \vee_{A_j \in X_i} \wedge_{a_{ij} \in A_j} a_{ij}$;
  if $n = 2$ **then** $x \leftarrow 1$;
  else if $n = 3$ **then**
    if $|X_1 \cap X_2| < |X_2 \cap X_3|$ **then** $x \leftarrow 1$ **else** $x \leftarrow 2$;
  else
    $x' \leftarrow$Split($\{\Upsilon_1, \ldots, \Upsilon_{n-1}\}$);
    $a \leftarrow |(X_1 \cup \cdots \cup X_{x'}) \cap (X_{x'+1} \cup \cdots \cup X_{n-1})|$;
    $b \leftarrow |(X_{x'+1} \cup \cdots \cup X_{n-1}) \cap X_n|$;
    if $a < b$ **then** $x \leftarrow x'$ **else** $x \leftarrow n - 1$;
  return x;

E HANDLING DUPLICATE RECORDS

In this section, we discuss how to extend our solution to support authenticated queries over duplicate records. Since our proof of completeness relies on the distinction of the query attribute, the idea is to transform the records sharing the same query key into distinct ones. To do so, we propose to introduce a new virtual dimension by extending the query attribute for each record with a random value in this virtual dimension. As such, we ensure that there is no duplication in the database with respect to each query key. When processing a query, the query range will be transformed accordingly to cover the whole space of the virtual dimension, whereas the rest of the algorithm is the same as that described in Section 6. Furthermore, it is worth noting that the data records that share the same query key and the same access policy can be aggregated into a super-record before the above transformation. This would reduce the space of the virtual dimension and hence the quantity of pseudo records that need to be introduced into the database.

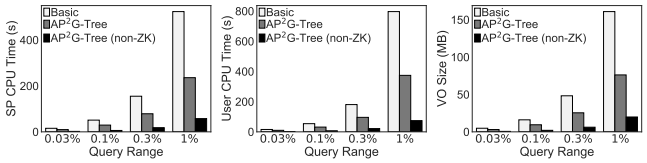


Figure 15: Range Query Performance with Duplicate Records

Consider an example where the database contains three records $\langle o_1, v_1, \Upsilon_1 \rangle$, $\langle o_1, v_2, \Upsilon_1 \rangle$, and $\langle o_1, v_3, \Upsilon_3 \rangle$. We first merge the first two records into $\langle o_1, v_1 || v_2, \Upsilon_1 \rangle$ since they share the same access policy. Then, a new virtual dimension x is introduced to distinguish the remaining two records as $\langle (o_1, x_1), v_1 || v_2, \Upsilon_1 \rangle$ and $\langle (o_1, x_2), v_3, \Upsilon_3 \rangle$, where $1 \leq x_1 \neq x_2 \leq U_x$ and U_x is the upper bound of the x dimension.[8] Accordingly, if a user issues a range query $[\alpha, \beta]$, the query range will be transformed into $[(\alpha, 1), (\beta, U_x)]$.

Further, in the applications where the zero-knowledge confidentiality requirement can be relaxed, it would be acceptable to disclose to the user about the distribution of the duplicate data records. As such, instead of introducing the virtual dimension and its associated pseudo records, we can embed the duplicate information directly in the APP signatures to reduce the database size and boost query performance. Consider a record $\langle o_i, v_i, \Upsilon_i \rangle$. Its APP signature will be $\sigma_i = \text{ABS.Sign}(sk_{\text{DO}}, hash(o_i)|hash(v_i)|dup\_num|dup\_id, \Upsilon_i)$, where *dup_num* captures how many duplicate records are associated with the query attribute o_i, and *dup_id* is used to identify each duplicate record. Thus, the authenticated query processing algorithm remains the same as that described in Section 6, and the user can verify the completeness by checking whether all duplicate records under the query attribute are present in the VO.

Figure 15 reports the results of performance evaluation over the default TPC-H database, where the access policies are randomly assigned to all data records. We compare two solutions of handling duplicate records: (i) the zero-knowledge approach by adding the virtual dimension (denoted as AP^{2}G-Tree); (ii) the non-zero-knowledge approach by embedding the duplicate information in the APP signatures (denoted as AP^{2}G-Tree (non-ZK)). The index size for the zero-knowledge approach is 12.17 GB (2.91 GB tree structure + 9.26 GB signatures), whereas in the non-zero-knowledge setting the index is 4.35 GB (1.02 GB tree structure + 3.33 GB signatures). The additional index overhead incurred for the zero-knowledge approach, as the cost of achieving the zero-knowledge requirement, is believed to be reasonable. Regarding the query performance, AP^{2}G-Tree (non-ZK) is approximate to the case without duplicate records since the duplicate information is seamlessly embedded in the APP signatures. As shown in Figure 15, the query cost in the zero-knowledge AP^{2}G-tree approach is only no more than 3 times worse than that in AP^{2}G-Tree (non-ZK). Moreover, the performance of AP^{2}G-tree is only about half that of the basic approach, which again demonstrates its pruning effectiveness for inaccessible records.

[8] In practice, U_x can be set to the maximum number of distinct access polices associated with each query key.