Chen, Yaxing; Zheng, Qinghua; Yan, Zheng; Liu, Dan

QShield: Protecting Outsourced Cloud Data Queries with Multi-User Access Control Based on SGX

Published in:
IEEE Transactions on Parallel and Distributed Systems

DOI:
10.1109/TPDS.2020.3024880

Published: 01/02/2021

Document Version
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

Please cite the original version:
Chen, Y., Zheng, Q., Yan, Z., & Liu, D. (2021). QShield: Protecting Outsourced Cloud Data Queries with Multi-User Access Control Based on SGX. *IEEE Transactions on Parallel and Distributed Systems* , *32*(2), 485-499. Article 9200772. https://doi.org/10.1109/TPDS.2020.3024880

QShield: Protecting Outsourced Cloud Data Queries with Multi-user Access Control Based on SGX

Yaxing Chen, Qinghua Zheng, *Member, IEEE,* Zheng Yan, *Senior Member, IEEE,* Dan Liu

***Abstract*—Due to the concern on cloud security, digital encryption is applied before outsourcing data to the cloud for utilization. This introduces a challenge about how to efficiently perform queries over ciphertexts. Crypto-based solutions currently suffer from limited operation support, high computational complexity, weak generality, and poor verifiability. An alternative method that utilizes hardware-assisted Trusted Execution Environment (TEE), i.e., Intel SGX, has emerged to offer high computational efficiency, generality and flexibility. However, SGX based solutions lack support on multi-user query control and suffer from security compromises caused by untrustworthy TEE function invocation, e.g., key revocation failure, incorrect query results, and sensitive information leakage. In this paper, we leverage SGX and propose a secure and efficient SQL-style query framework named QShield. Notably, we propose a novel lightweight secret sharing scheme in QShield to enable multi-user query control; it effectively circumvents key revocation and avoids cumbersome remote attestation for authentication. We further embed a trust-proof mechanism into QShield to guarantee the trustworthiness of TEE function invocation; it ensures the correctness of query results and alleviates side-channel attacks. Through formal security analysis, proof-of-concept implementation and performance evaluation, we show that QShield can securely query over outsourced data with high efficiency and scalable multi-user support.**

***Index Terms*—Secure query, Outsourced data, Secure hardware, Intel SGX, Cloud computing, Multi-user query control**

I. Introduction

The cloud computing paradigm, characterized by *convenience*, *elasticity* and *low-cost*, demonstrates a great success in the past decade [1]. Organizations typically need to deploy their application services to remote servers that are not in charge by themselves, which leaves these organizations no choice but to trust the cloud in managing their data for utilization. This security assumption that the cloud is fully trusted, however, is not always valid since the cloud may suffer from malfunctions, compromises, intrusions or attacks. Hence, many cloud-based systems employ cryptography to encrypt confidential data when being transmitted, processed and/or stored. However, one of the biggest challenges caused by cloud data encryption is how to efficiently utilize encrypted data while not being bothered by encryption, e.g., queries that are frequently performed in many cloud systems [2]–[4]. Typically, a data query can be represented by a SQL-style expression and interpreted as a query plan, i.e., a directed acyclic graph (DAG) of computational operators for execution, such as *projection*, *selection*, *aggregation*, *union* and *join*.

Partially/Fully homomorphic encryption (PHE/FHE) is a fundamental technology to solve cloud data security and privacy problems in the literature. However, such pure crypto-based solutions at present suffer from severe operation flexibility with limited operation support and performance issues due to high computational complexity [5]–[7]. Searchable Encryption (SE) enables search over encrypted data. Albeit there exist extensive investigations along this research line, current SE implementation is still not satisfactory [8]. Notably, building upon various crypto primitives, different SE schemes support different search types, e.g., single keyword, multi-keywords and range, with different index structures, and are not compatible with each other. Besides, SE focuses on conditional information retrieval, which can only be viewed as a *selection* operator. As such, existing schemes inherently cannot directly apply to (or be extended to) support generic queries. In addition, cryptographically verifying the correctness and integrity of cloud operations is inefficient and remains an open problem [9].

In recent years, a hardware-assisted Trusted Execution Environment (TEE, also called enclave), i.e., Intel Software Guard Extensions (SGX), has been applied [8], [10]–[12] as an alternative promising countermeasure, but presents several limitations.

Issue I (No Effective Support on Multi-user Query Control). Multi-user query control is a fundamental function for cloud data utilization. Using SGX to realize multi-user query control is still an open issue. Consider the following strawman protocol for an enclave to support query control over multiple data users.

Strawman Protocol: Let the data owner securely provision its data encryption key and his/her access policy to an enclave through remote attestation (authorization). Then, a data user remotely attests to the genuineness of the enclave; During the process, the data user and the enclave (on behalf of the data owner) provide each other with their own digital credentials

We would like to thank Wen-hai Sun, Ning Zhang and Xue-qin Liang for their insightful comments on the manuscript. We also thank Wen-jing Lou for her constructive suggestions and kindly comments on this research topic. This work was sponsored by National Key Research and Development Program of China under Grant Nos. 2018YFB1004500, 2016YFB1000903, Innovative Research Group of the National Natural Science Foundation of China (61721002), Innovation Research Team of Ministry of Education (IRT_17R86), the National Science Foundation of China under Grant Nos. 61502379, 61532015, 61672410 and 61672420, Project of China Knowledge Center for Engineering Science and Technology. This work is also supported in part by the Academy of Finland under Grants 308087 and 314203.

Y. Chen and Q. Zheng are with the School of Computer Science and Engineering, Xi'an Jiaotong University, Xi'an, Shaanxi, China. Email: cyx.xjtu@gmail.com, qhzheng@mail.xjtu.edu.cn

Z. Yan and D. Liu are with the State Key Lab on Integrated Services Networks, School of Cyber Engineering, Xidian University, Xi'an, Shaanxi, China. Email: 15596173220@163.com, zyan@xidian.edu.cn; Z. Yan is also with the Department of Communications and Networking, Aalto University, Espoo, Finland. Email: zheng.yan@aalto.fi

for mutual authentication; A successful mutual authentication results in a private communication channel for later query requests and responses. At query time, the enclave loads encrypted data and only recovers the data for the specifically authorized user according to an access policy (enforcement).

Such a method, however, exposes some fatal flaws: 1) Remote attestation between the data user and the enclave is a cumbersome process; the involved mutual authentication typically needs the help of burdensome Public Key Infrastructure (PKI). This is time-costly. 2) The enclave needs to maintain a crypto session key for the established private communication channel for each data user, which incurs a non-trivial key management overhead, especially for storage. The above two flaws together cause poor scalability in multi-user query control. 3) It is risky to hand out the crypto key to the enclave to handle multi-user query control. On one hand, a compromised TEE host can make use of the enclave to perform unauthorized computation. A concrete example is the recent identified key revocation problem [13], [14]. On the other hand, the long-term crypto key within an enclave can be extracted due to various known attacks.

Issue II (No Proper Guarantee on Trustworthy TEE Function Invocation). On the other hand, we notice that a practical design of an SGX-based query system is to implement each computational operator as a unique TEE interface function so as to ensure high flexibility and generality [11], [15]. In other words, one query often involves multiple TEE function calls. At present, SGX remote attestation only ensures the integrity of one TEE function at run-time, but the correctness of query execution, more generically, the correctness of TEE function invocation, is not guaranteed, caused by the distrust of TEE host. The untrustworthy TEE function invocation will inevitably introduce some non-trivial security and privacy issues, especially after the crypto key has been delivered to an enclave. Intuitively, malicious TEE function invocation could incur incorrect query results. Besides, it makes an untrusted TEE host capable of ruining access control over multiple data users, i.e., the recovered data within an enclave for a data user can be utilized to serve queries issued by other data users. Further, it may open a back door for exposing sensitive information by launching side-channel attacks [16], [17]. At present, efficiently guaranteeing the execution (invocation) integrity of a TEE host atop the TEE stack remains an open problem [18].

However, it is challenging to solve the above two issues simultaneously. First, it is not an easy task to design a key management scheme that can control multi-user query and at the same time avoid disclosing the full crypto key to the enclave. Obviously, an enclave may be maliciously used by its host. The enclave may not be as impregnable as a wall of iron in a long run; the crypto key could be exposed if staying in the enclave for a long term. This seems a dilemma for the data owner, since it needs to entrust the crypto key to the enclave for data utilization. Second, it is tough to achieve scalable multi-user query control and efficient enforcement on trustworthy invocation of TEE functions in the context of fulfilling data queries by executing multiple TEE functions since tradeoff between performance and security is a persistent topic in system design.

Our Approach. In this paper, we propose a secure and efficient query framework called QShield to enable scalable multi-user utilization of outsourced data. It adopts Intel SGX to establish hardware-assisted enclaves in the untrusted cloud so as to protect the confidentiality and integrity of sensitive data run inside. Besides, QShield incorporates a generic SQL-style query model such that it is capable of handling majority of common data query tasks. Furthermore, since cloud applications in web, mobile, social or IoT scenarios often use different data models, such as relational tables, key-value items and data streams, QShield exploits a widely-adopted and flexible document-oriented data model to enable compatibility and support high generality.

In order to tackle the multi-user query control issue, we present a novel and lightweight secret sharing scheme. The core idea is to let the data owner assign an attested enclave a key share sk_a and each authorized data user i another individually unique key share sk_b^i. The encryption key sk can be reconstructed and used for decryption only if the sk_b^i is delivered to the enclave that holds sk_a per query. Once the authorized data are recovered, the crypto key sk is erased in the enclave. As such, neither data users nor the enclave have full capability to recover the encrypted data with their own key shares outside the scope of the current query. Moreover, a secret existing in the enclave for a small time-interval can greatly reduce the possibility of being exposed through side-channel attacks; the non-trivial key leakage problem can then be solved. Another benefit is the enclave and the data user can effectively authenticate with each other through their unique key shares rather than cumbersome remote attestation.

In order to overcome Issue II, we further propose a trust-proof mechanism to enforce that the recovered data within the enclave can only be used to serve the current query and prove that the query result offered by the enclave is correct. To be specific, on one hand, we leverage an endurance indicator denoted by ω to restrict the times that the recovered data as well as its derived intermediate results can be utilized by the enclaved operators. On the other hand, we make the enclave record and output the footprints of the distributed query execution as a workflow proof for auditing. As such, any malicious TEE function invocation during query execution will be detected. Without capability to arbitrarily invoke TEE functions, the specific side channel attacks launched by a TEE host can thus be alleviated.

Contributions. In summary, the main contributions of this paper are: 1) Building upon the off-the-shelf hardware-assisted TEE, i.e., Intel SGX, we propose a practical secure query framework named QShield for outsourcing data to the untrusted cloud. By supporting common SQL-style query expressions and flexible document-oriented data model, QShield can be easily adopted by most of cloud-based query application scenarios. 2) Under the threats caused by the limitation of SGX architecture, we present a secure and lightweight secret sharing scheme to make QShield capable of realizing scalable multi-user query control. 3) We also propose a trust-proof mechanism to audit the correctness of query execution and meanwhile greatly alleviate the possibility to launch side

channel attacks with regard to TEE function invocation. 4) We provide formal security analysis over QShield. Furthermore, we implement a proof-of-concept prototype and our experimental evaluation confirms QShield's efficiency and flexibility, compared to existing crypto-based solutions.

Organization. The rest of this paper is organized as follows. Section II reviews the literature related to our work. Section III describes system, data and threat models of QShield and our research assumptions. We introduce our research preliminaries and present the design of QShield in Section IV, followed by security analysis in Section V. We implement a proof-of-concept QShield system in Section VI and evaluate its performance in Section VII. Finally, we conclude the whole paper in the last section.

II. Related Work

A. Work Fully Based on Cryptography

Cryptography is a widely-adopted technology to protect outsourced queries (search) in the cloud. Initially, Song et al. [19] presented a special two-layered encryption construct for each query word based on stream ciphers to realize keyword search over encrypted documents. When a server computes over a construct with a trapdoor, it can strip the outer layer and assert whether the inner layer has a correct form. However, their scheme suffers from statistical attacks. As the number of documents scales up, it also experiences great performance degradation on search. In order to solve these limitations, Goh [20] proposed to build a secure index for each document. Specifically, it utilizes Bloom filters and pseudo-random functions to implement indexing. Chang et al. [21] also leveraged secure indexes in their scheme, but they try to avoid any information leakage caused by trapdoors towards the words being queried. Curtmola et al. [22] formally defined secure indexes with new and stronger security definitions. They proposed the first secure searchable symmetric encryption scheme under a multi-user setting, which was then improved by Kamara et al. [23] to support document insertion and deletion. Search over encrypted data is also considered in a public-key setting where anyone who possesses a public key can update an index with new words, but only the owner of private key can generate trapdoors to query over the index. As a concrete example, Boneh et al. [24] leveraged identity-based encryption to build secure keyword search. Current work along this research line in the literature focuses on improving data utilization, i.e., to support complex query types rather than just keyword search. For example, Cash et al. [25] gave a solution for searchable symmetric encryption with conjunctive search and general boolean queries. Sun et al. [26] realized range queries by reducing a range query to a secure multi-keyword query. More generally, Gahi et al. [27] proposed to utilize homomorphic encryption to implement SQL queries over an encrypted relational database. Unfortunately, it is not sufficiently mature yet as the underlying technique of homomorphic encryption is time-consuming and cannot support all generic computations or operations over encrypted data. Another practical method at the time is to incorporate various crypto primitives into a system to support more advanced query semantics over encrypted databases. One representative system is CryptDB [2]. Streamforce [3] and PloyStream [4] target at real-time stream processing and extend CryptDB to support multi-user query control. However, both systems encrypt a piece of data multiple times so as to satisfy the queries requested by different data users. As a result, such an improvement introduces significant performance overhead and causes a scalability problem.

B. Work Based on TEE

A promising way to protect outsourced queries (search) is exploiting hardware-assisted TEE, e.g., Intel SGX. It avoids time-consuming and cumbersome computations caused by software-based cryptography. Besides, it is theoretically capable of implementing arbitrary query semantics. For example, Rearguard [8] leverages SGX to enable secure keyword search. As a concurrent work, HardIDX [10] also utilizes SGX to build secure indexes for searchable encryption. VC3 [12] uses SGX to build a secure version of map-reduce processing framework. Opaque [11] and Hermetic [15] both allow a number of generic SQL queries over encrypted data with SGX, but they dedicate themselves to mitigating critical side channel attacks, without any discussion on multi-user query control. TrustedDB [28] allows data users to execute SQL queries with privacy and under regulatory compliance constraints. It, however, adopts server-hosted and tamper-proof cryptographic coprocessors (SCPUs) to facilitate secure computation in critical query processing, which deploys a much bigger trusted computing base (TCB) than SGX-based solutions. All of the above works mainly consider to protect data confidentiality during query processing without any support on multi-user query control. By contrast, EnclaveDB [29] was endowed with an ambitious design goal by applying SGX to ensure confidentiality, integrity and freshness for both queries and data. It implements the strawman protocol to achieve multi-party access control, which requires to pre-embed public keys of authorized data users into an enclave and perform remote attestation as per data user. Thus, EnclaveDB suffers from a severe scalability problem.

With regard to guaranteeing trustworthy TEE function invocation, VC3 presents a solution to ensure the integrity of distributed computational workflows. Its solution, however, needs high inter-node communication overhead regarding verification messages. To solve this issue, Opaque introduces a self-verification protocol, where a data user distributes an encrypted query plan to all enclave instances serving for the current query. Each enclave instances, in turn, takes responsibility to check whether its received inputs confirm the DAG (input edges of a corresponding node). Due to the additional overhead introduced by real-time verification on the authenticity of all inputs of an enclave instance, Opaque incurs performance degradation during query execution, which can be a big performance issue when the system workload is high. EnclaveDB chooses to host queries and a whole query engine in an enclave to protect the confidentiality and integrity of queries. But the Enclave Page Cache (EPC) that hosts enclaves is limited to 128MB in SGX (v1.0) and 256MB in SGX (v2.0).

Such a capacity limitation often produces a large amount of costly coarse-grained paging and thus causes significant performance penalty [30]. As such, EnclaveDB exacerbates performance in view of EPC limit. Besides, EnclaveDB requires to pre-define all queries and pre-compile them as a part of enclave procedure, which causes poor flexibility and low generality in a multi-user setting.

Our Work. QShield's design follows Occam's Razor principle. We try to minimize the job of enclaves (similar to Opaque-style systems), i.e., only query operators are implemented as enclave procedures, so as to maximize EPC utilization, but meanwhile providing strong security guarantees for outsourced queries. To this end, we implemented an efficient trust-proof mechanism to ensure the integrity of queries, even if a host program (query engine) is compromised. Moreover, we implemented a lightweight secret sharing scheme to enable efficient and scalable multi-user query control.

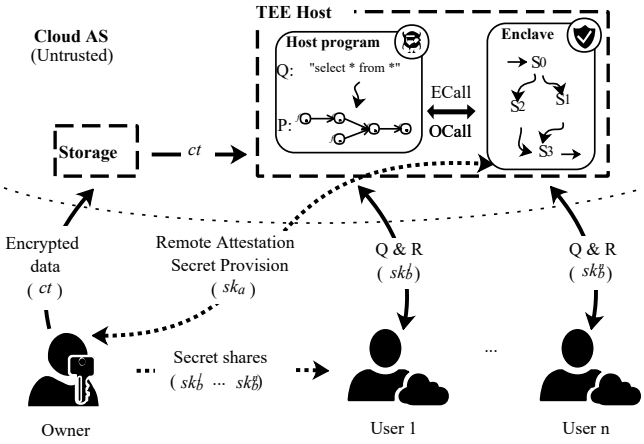

Fig. 1. System Model

III. Problem Statements

In this section, we introduce the system model, data model and security model of QShield and describe our research assumptions.

A. System Model

As illustrated in Fig. 1, QShield consists of three types of system entities: data owner, data user and cloud Application Service (AS). The data owner uses a symmetric key sk to encrypt its sensitive data before outsourcing them to the cloud AS for utilization. Multiple data users may query the data to retrieve valuable information. The cloud AS in our system has rich storage and computational resources; it is responsible for storing outsourced data, establishing an enclave for the data owner, and handling user queries. Similar to most well-known outsourced databases, e.g., Amazon RDS and Google Cloud SQL, the economic model existing amongst the three entities is: the data owner pays the cloud AS for its platform resources, including storage, computation and bandwidth; the data users are charged by the data owner for the outsourced informational data. At system initialization, a secret sharing scheme is set up amongst the three entities to support multi-user query control in QShield. Concretely, the data owner defines access permissions for data users in the system and meanwhile remotely attest a dedicated enclave created by a TEE host (To be distinguishable, we use "TEE host" in this paper to refer to the non-enclave computational part of the cloud AS). After a successful remote attestation, the data owner securely delivers a key share sk_a of sk and an access policy specification to the enclave. It also assigns each authorized data user the other individually unique key share sk_b^i of sk via a secure channel by applying a secure communication protocol (as assumed).

The TEE host processes a query issued by a data user as follows. It first forwards the token to the enclave, enabling the enclave to recover authorized data (initial computational state) for the current query, and then transforms the query expression into a query plan. Afterwards, it physically invokes corresponding enclaved operators with proper parameter values for query execution. As the whole computation of the query plan globally runs to completion, the enclave will produce several intermediate results (intermediate computational states) and reach a final query result (final computational state). In order to defend against malicious invocation of enclaved operators, QShield also executes the trust-proof mechanism along with the state transition process. At last, the TEE host receives from the enclave a cryptographically-protected query result and an enclave-signed trust-proof, and sends them back to the data user as query response.

Query & Query Plan. A query in QShield by design incorporates a SQL-style parametric query expression, cryptographically protected parameter values, and a token. A query plan is represented by a Direct Acyclic Graph (DAG). To be specific, vertices in the DAG stand for well-designed enclaved operators (each kind of query operators is implemented as a unique TEE function; currently QShield supports *projection*, *selection*, *aggregation*, *union*, *sort* and *join*) and directed edges show the workflow between them.

Data Model. We opt for a document-oriented data model in QShield due to its flexibility in being compatible with most types of data models, e.g., relational tables, key-value items and data streams, exploited by current web, mobile, social, as well as IoT applications. Specifically, QShield employs the notion of *document* D as a basic logical unit for data storage and query, which, encoded in JavaScript Object Notation (JSON), contains a set of *attributes* $\{A_1, \cdots, A_n | A_j := < name_j, value_j >, j = 1, \cdots, n\}$, where $name_j$ is a description of the *attribute* A_j and $value_j$ is the value of *attribute* A_j. It also uses the notion of *collection* C to represent a group of *documents*, each of which includes a same set of attributes. When outsourced to the cloud, a *collection* of size r can be divided into multiple parts, each stored as a data file.

B. Threat Model

We assume that the queries themselves are not sensitive - only their answers, parameters and input datasets are. This is a practical assumption for two reasons. First, the TEE host has to invoke corresponding enclaved operators for execution of queries; it makes no sense to hide query semantics. Second, since the parameters, input and output of a query

are cryptographically escorted to and from the enclave in QShield, unauthorized entities, including the TEE host, have no way to access them and thus cannot accurately infer user interests or behaviors. We opt for a widely-adopted "honest-but-curious" model with regard to data users, i.e., they will honestly execute protocol but also desire to query data beyond their own permissions.

The cloud AS is considered to be semi-trusted. Besides the standard SGX threat model where an attacker may control the cloud's software stack, including hypervisor and OS, we consider a more powerful attacker who may also compromise the TEE host. As such, 1) the attacker may prevent the data owner from revoking the previously entrusted crypto key on demand by selectively dropping network packets; 2) the attacker may arbitrarily invoke enclave interface functions, resulting in incorrect query results and potential knowledge extraction or secret leakage.

Limitations. Intel SGX at present is reported to suffer from side channel attacks, including cache timing, power analysis, branch shadowing, and the most recently discovered foreshadow transient execution, etc. [31]–[34]. Indeed these problems are under investigation and a variety of countermeasures have been proposed in the literature, such as, T-SGX [35], which makes use of hardware transaction memory to detect malicious page fault monitoring, Raccoon [36], which compiles programs in a way that eliminates data-dependent branches, and Opaque [11], which implements oblivious algorithms to hide memory access pattern. We remark that these approaches are orthogonal to our work and it is interesting to adapt their ideas to QShield in the future. DoS/DDoS attack and enclave bugs are outside the scope of this paper's consideration.

Additional research assumptions are listed below. 1) The communication channels between the data owner and the data users are secure. 2) The communication channels between the data users and the cloud AS are implemented based on the current Internet infrastructure. 3) The data owner is honest and does not collude with the cloud AS and the data users due to profit conflicts. 4) We do not consider the collusion between the cloud AS and data users. First, the enclave enforces access policy towards the data users and the other key share hold by the enclave is still kept secret. Second, the data owner may launch some countermeasures like exploiting a "honeypot" mechanism to alleviate the intention of the cloud AS to make collusion, i.e., the data owner may disguise to be a data user to collude with the cloud AS; if successful, the cloud AS will face a huge fine. Third, data users mostly have no interest in sharing their purchased data with the concern of copyright regulations and digital rights management.

IV. QShield Framework

In this section, we first give preliminary knowledge on adopted fundamental techniques: Intel SGX and bilinear maps [37]. Then, we introduce the core components that enable QShield to accomplish its promising design goals, followed by a detailed description on stand-alone QShield protocols. A discussion about how to apply QShield in a distributed context is provided in Appendix B.

A. Preliminaries

1) Intel SGX: Intel SGX is a promising hardware-assisted trusted computing technology. It provides *memory isolation* [38], which enables a TEE host set up a protected execution environment, called enclave, such that code and data run inside are resilient to attacks from privilege software, including OS kernel and VM hypervisor. Function calls between the untrusted TEE host and enclave are through well-designed ECALL/OCALL interfaces. Such an architecture implies that the invocation of enclave functions is unreliable since it is still under control of the untrusted TEE host. Intel SGX also offers two auxiliary functionalities: *remote attestation* and *storage sealing* [39]. The former makes a distant entity capable of verifying the authenticity of an enclave, checking the integrity of desired code running inside and meanwhile establishing a secure communication channel with the enclave. The latter allows to securely store enclave data in an untrusted storage outside the enclave for future recovery, in case of server shutdown, system failure, and/or power outage. Please refer to [40] for a more thorough technical analysis about Intel SGX.

2) Bilinear Maps: Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two multiplicative cyclic groups of prime order p, and g be a generator of $\mathbb{G}_1$. An efficiently computable bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$ defined over the two groups satisfies the following properties:

- Bilinearity: For all a, $b \in \mathbb{Z}_p$, there exists $e(g^a, g^b) = e(g, g)^{ab}$
- Non-degeneracy: $e(g, g) \neq 1$
- Computability: For any u, $v \in \mathbb{G}_1$, there exists an efficient algorithm to compute $e(u, v)$.

Decisional Bilinear Diffie-Hellman (BDH) Assumption. The security of our secret sharing scheme is based on the Decisional BDH assumption. Basically, let a, b, c, z be chosen randomly from $\mathbb{Z}_p$, there exists no probabilistic polynomial-time algorithm $\mathcal{B}$ that can distinguish the tuple $(A = g^a, B = g^b, C = g^c, e(g, g)^{abc})$ from the tuple $(A = g^a, B = g^b, C = g^c, e(g, g)^z)$ with more than a negligible advantage ϵ, that is,

$$|Pr[\mathcal{B}(A, B, C, e(g, g)^{abc}) = 0] - Pr[\mathcal{B}(A, B, C, e(g, g)^z) = 0]| \leq \epsilon,$$

where the probability is computed over the randomly chosen generator g, the randomly chosen a, b, c, z in $\mathbb{Z}_p$, and the random bits consumed by $\mathcal{B}$.

3) Notations: We use f_* to stand for an enclaved operator $*$, where $*$ can be a projector π, a selector σ, an aggregator ϕ, an unioner ψ, a sorter χ or a joiner γ. Besides, E represents an authenticated symmetric key encryption scheme with $Enc(\cdot)$ and $Dec(\cdot)$ algorithms. PKE represents an indistinguishability security under chosen plaintext attack (IND-CPA) public key encryption scheme with $KeyGen(\cdot)$, $Enc(\cdot)$ and $Dec(\cdot)$ algorithms. S stands for an existentially unforgeable signature scheme with $KeyGen(\cdot)$, $Sign(\cdot)$ and $Verify(\cdot)$ algorithms. $\mathcal{H}(\cdot)$ denotes a collision resistant hash function.

B. QShield Components

In this subsection, we introduce the basic components of QShield: the secret sharing scheme and the trust-proof mechanism.

1) Secret Sharing: The secret sharing scheme in our research context can be defined over a tree-based access structure τ. Specifically, the root node of the access tree has an AND gate: Its left child represents the enclave; Its right child has an OR gate with n children, each representing a data user. We notice that this construction has been widely used in cryptography, like attribute-based encryption. Similarly, we base the bilinear map to design a crypto primitive called $\mathcal{E}$ to facilitate secret sharing in QShield.

Suppose we have two cyclic groups $\mathbb{G}_1$ and $\mathbb{G}_2$, both with prime p. Let g be the generator of $\mathbb{G}_1$ and $e : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$ be a bilinear map over $\mathbb{G}_1$ and $\mathbb{G}_2$. The $\mathcal{E}$ crypto primitive consists of following algorithms:

- $\{pk, msk\} \leftarrow setup(1^\lambda, n)$: This algorithm takes as inputs a security parameter λ and a non-zero positive integer n. It first defines a universe of participants $U = \{0, 1, ..., n\}$, where 0 and $1, \cdots, n$ stand for the enclave and data users, respectively. For each participant $i \in U$, it uniformly chooses a number t_i from $\mathbb{Z}_p$. It then picks up a number y from $\mathbb{Z}_p$. Finally, this algorithm outputs a public key $pk : \{T_0 = g^{t_0}; T_1 = g^{t_1}, \cdots, T_n = g^{t_n}; Y = e(g, g)^y\}$ and a master private key $msk : \{t_0; t_1, \cdots, t_n; y\}$.
- $\{sk_a; sk_b^1, \cdots, sk_b^n\} \leftarrow sharesGen(\tau, msk)$: This algorithm takes as inputs the predefined tree-based access structure τ and the master secret key msk. It computes key shares for all involved participants $\{i | i \in U\}$ as follows. The algorithm first defines a polynomial $q_t(x)$ for each node t in τ. Specifically, the degree d_t of each polynomial is derived from the threshold value k_t of the current node, i.e., $d_t = k_t - 1$: Suppose that the node has an AND gate, then k_t equals to the number of the node's children; Suppose that the node has an OR gate or it is a leaf node, then k_t equals to 1. Besides, for the root node r, it sets $q_r(0) = y$; for other nodes $t (t \neq r)$, it sets $q_t(0) = q_{parent(t)}(index(t))$, where $parent(t)$ and $index(t)$ represent the parent node of t, the index in all children of t's parent node, respectively. Furthermore, all other polynomial parameters are chosen from $\mathbb{Z}_p$ randomly. Once these polynomials have been decided, the algorithm generates a key share for each participant i by computing $sk^i = g^{\frac{q_{t=i}(0)}{t_i}}$. To make it easily distinguishable, we denote the key share of enclave sk^0 as sk_a, and the key shares of data users $\{sk^1, \cdots, sk^n\}$ as $\{sk_b^1, \cdots, sk_b^n\}$.
- $ct_\kappa \leftarrow secretDist(\kappa, pk)$: This algorithm takes as inputs a secret κ and the public key pk. It generates a corresponding ciphertext ct_κ that can be distributed amongst participants $\{i | i \in U\}$ and only the enclave joining together with one of data users can recover the secret. To do so, the algorithm first chooses a random number s from $\mathbb{Z}_p$ and then encrypts κ in $\mathbb{G}_2$ as $ct_\kappa = \{\kappa \cdot Y^s; \{E_i = T_i^s\}_{i \in U}\}$.
- $\kappa \leftarrow secretRec(ct_\kappa, sk_a, sk_b^{i|i \in \{1, \cdots, n\}})$: This algorithm takes as inputs the ciphertext ct_κ, the key share sk_a of the enclave, and the key share $sk_b^{i|i \in \{1, \cdots, n\}}$ of one of data users. In order to reconstruct the secret κ, it performs a bottom-up node decryption process. F_t denotes the decrypted value (blinding factor) of a node t. In the case where t is a leaf node, it computes

$$F_t = \begin{cases} e(E_i, sk_a), \text{if } i = 0 \\ e(E_i, sk_b^i), \text{otherwise} \end{cases}; \tag{1}$$

when t is a non-leaf node, i.e., AND gate and OR gate, it computes

$$F_t = \prod_{t' \in C(t)} F_{t'}^{\Delta_{j, \bar{C}(t)}(0)}, \tag{2}$$

where $C(t)$ is a collection of child nodes of t, $\bar{C}(t)$ is a corresponding index collection, i.e., $\bar{C}(t) = \{k | m \in C(t), k = index(m)\}$, $j = index(t')$, and

$$\Delta_{j, \bar{C}(t)}(x) = \prod_{k \in \bar{C}(t), k \neq j} \frac{x - k}{j - k} \tag{3}$$

is Lagrange coefficient. After such a decryption process finished, the algorithm can obtain the root blinding factor $e(g, g)^{ys}$, i.e., Y^s, which can then be used to recover the secret κ.

The correctness of $\mathcal{E}$ is proved in Appendix C.

2) Trust-proof: Formally, we define a Finite State Machine (FSM) as a 5-tuple (Q, Σ, T, Q_0, Q_k) to represent the trust-proof mechanism, which consists of a finite set of FSM states Q, a finite set of input symbols Σ, a transition function: $T : Q \times \Sigma \rightarrow Q$, an initial FSM state $Q_0 \in Q$, and a final FSM state $Q_k \in Q$. Notably, a FSM state in Q incorporates all computational states at a certain point within the enclave. An input symbol in Σ is a triple $(*, idx, pred)$, where $*$ stands for the types of enclaved operators, i.e., π (*projection*), σ (*selection*), ϕ (*aggregation*), ψ (*union*), χ (*sort*), and γ (*join*), idx is the index of a computational state in a FSM state, and $pred$ represents query predicates. Figure 2 depicts the paradigm of the FSM state transition used in the trust-proof mechanism.

When a data user issues a query to the cloud AS, he/she computes a value ω called endurance indicator based on the query statement, which is used to restrict the times that the computational states within the enclave can be accessed by enclaved operators. ω is actually the number of nodes in a query execution plan and can be efficiently computed with an off-the-shelf open-sourced SQL parser (a spark query parser is used in our specific implementation) within approximate 100 milliseconds at the data user side in our experimental environment. Then, the data user utilizes the cryptographically protected token to securely bring the endurance indicator to the enclave.

On the cloud side, with a valid query token, the enclave recovers authorized data and meanwhile produces a computational state S_0. At this moment, an initial FSM state is also created, i.e., $Q_0 = \{S_0; w = \omega\}$. Here, w is a parameter in computational states to record the latest value of the endurance indicator; we especially illustrate w in FSM states to reflect its variation during query execution. Afterwards, the TEE host successively invokes enclaved operators to physically execute a query plan of the query. Note that, every time an enclaved operator is successfully executed, the enclave will produce a new computational state from old ones; notably, w in all existing computational states are reduced by 1. Correspondingly,

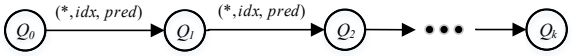

Fig. 2. FSM state transition diagram

a new FSM state will be created, i.e., a FSM state transition occurs. For example, after the first enclaved operator finishes its work, a new computational state S_1 is produced from S_0, then a new FSM state $Q_1 = \{(S_0, S_1); S_0 \rightarrow S_1; w = \omega - 1\}$ is created (we use an unreal $\rightarrow$ here to highlight the inherent relationship between computational states in a FSM state, the same as below); after the second enclaved operator finishes its work, a new computational state S_2 is produced from S_0 and S_1, then a new FSM state $Q_2 = \{(S_0, S_1, S_2); S_0 \rightarrow S_1, S_0 \rightarrow S_2, S_1 \rightarrow S_2; w = \omega - 2\}$ is created; and so on and so forth. By design, when the whole query plan is successfully executed, a final computational state S_k is produced from S_{k-1} (at least) and meanwhile w will be reduced to 0, such that no further accesses over relevant computational states are allowed any more, then the final FSM state Q_k is reached, i.e., $Q_k = \{(S_0, S_1, \cdots, S_k); S_0 \rightarrow S_1, S_0 \rightarrow S_2, \cdots, S_{k-1} \rightarrow S_k; w = 0\}$. Upon the enclave is in the final FSM state, it generates a trust-proof $tp = (\mathcal{G}, \sigma_{\mathcal{G}})$ for the current query by firstly creating a directed graph $\mathcal{G}$ from existing computational states $S_0, \cdots, S_k$, where a node denotes a computational state with its ID and w, and an edge denotes the invoked enclaved operator that produces such a computational state transition; and secondly signing $\mathcal{G}$ by executing $\sigma_{tp} \leftarrow S.Sign(sk_{e,sign}, \mathcal{G})$.

The verification on the trust-proof $tp = (\mathcal{G}, \sigma_{\mathcal{G}})$ by the data user is straightforward: it checks the integrity and genuineness of tp by running $S.Verify(pk_{e,sign}, tp, \sigma_{tp})$; if successful, it then checks whether w is 0 and the directed graph $\mathcal{G}$ confirms the query plan (DAG) of the query by graph matching.

C. QShield Protocols

In this subsection, we describe QShield protocols in details, which include System Setup, Data Upload, Data Query, and Policy Update. Figure 3 depicts an overview of QShield protocols.

System Setup. The data owner first selects a security parameter λ and defines the scale n of data users, i.e., the maximum number of data users allowed in the system. Then, it executes $\{pk, msk\} \leftarrow \mathcal{E}.setup(1^\lambda, n)$ to initialize the secret sharing subsystem and subsequently executes $\{sk_a; sk_b^1, \cdots, sk_b^n\} \leftarrow \mathcal{E}.sharesGen(\tau, msk)$ to produce key shares for involved participants. It also creates a 256-bits symmetric key κ for data encryption and generates a corresponding cipher ct_κ by running $\mathcal{E}.secretDist(\kappa, pk)$. Afterwards, the data owner creates an access policy specification pol, designating query privileges of each data user over the outsourced data. The pol is a list denoted by $\{(uid, cids)_j | j = 0, \cdots, n\}$, where uid and $cids$ of an entry j represents the ID of a data user and all IDs (initialized as NULL) of *collections* that are authorized to the data user, respectively. Notice that, msk, κ, sk_a, and sk_b^i are securely stored by the data owner; pk and ct_κ are published as system parameters.

On the cloud side, the TEE host creates a dedicated enclave for the data owner. It subsequently performs a predefined TEE function to let the enclave itself generate a 256-bits public key pair, i.e., $(pk_{e,msg}, sk_{e,msg}) \leftarrow PKE.KeyGen(1^\lambda)$, and a 256-bits signature key pair, i.e., $(vk_{e,sign}, sk_{e,sign}) \leftarrow S.KeyGen(1^\lambda)$. $pk_{e,msg}$ and $vk_{e,sign}$ are output as public system parameters.

Next, the data owner verifies whether the enclave is correctly deployed and executed on a genuine SGX-enabled CPU platform through remote attestation. During this process, it also establishes a secure communication channel with the enclave by negotiating a symmetric secret key sk_{comm}. Once the attestation is successful, the data owner could make use of sk_{comm} to escort its sk_a and pol to the enclave. Notably, the data owner runs $ct \leftarrow E.Enc(sk_{comm}, (sk_a, pol))$ and the enclave in turn runs $(sk_a, pol) \leftarrow E.Dec(sk_{comm}, ct)$.

Data Upload. When a new *collection* msg is ready for utilization, the data owner executes $ct_{msg} \leftarrow E.Enc(\kappa, msg)$ to obtain its ciphertext ct_{msg}, and then performs $\mathcal{H}(ct_{msg})$ to create a unique ID cid for it, and further uploads a tuple (cid, ct_{msg}) to the cloud AS. Next, the data owner updates its access policy for all data users towards the newly added *collection*. This operation correspondingly triggers the Policy Update protocol.

Data Query. In this protocol, an authorized data user i issues a SQL-style query over the *collections* shared by the data owner. To this end, the data user parameterizes the query, producing a parametric expression exp, and encrypts each parameter value v with the enclave's public key $pk_{e,msg}$, i.e., $ct_v \leftarrow PKE.Enc(pk_{e,msg}, v)$. All encrypted parameter values constitute $params$. He/she also generates a token tk by encrypting a triple (sk_b^i, ω, c) using $pk_{e,msg}$, where sk_b^i is the data user's key share, ω is a positive integer calculated based on exp, and c is a freshness factor that is a monotonically increasing number. Formally, the query is denoted by a triple $(exp, params, tk)$.

Upon the cloud AS (TEE host) receives the query from the data user, it performs the following three steps: *unlock*, *query*, and *response*.

unlock: The TEE host enables the enclave to recover the *collections* allowed to be accessed by the data user. Specifically, the TEE host first forwards relevant encrypted *collections* $[(cid, ct_{msg})]$ along with the token tk to the enclave, within which the enclave executes $(sk_b^i, \omega, c) \leftarrow PKE.Dec(sk_{e,msg}, tk)$. Then, the enclave checks whether or not the current query has been previously handled by comparing c with a c' (initialized as an infinitesimal number). If c is smaller than c', the enclave will deny to serve the current request; otherwise, it updates c' with c. The enclave also iteratively computes $\mathcal{H}(ct_{msg})$ and compares it with the corresponding cid so as to verify the integrity of input *collections*. Once passing the freshness check and integrity verification, the enclave is able to execute $secretRec(ct_\kappa, sk_a, sk_b^i)$ to recover the encryption key κ. and further iteratively execute $E.Dec(\kappa, \cdot)$ on $[(cid, ct_{msg})]$ to obtain the underlying *collections* $[(cid, msg)]$ (κ is erased when decryption is finished). Next, the enclave filters out from $[(cid, msg)]$ the unauthorized *collections* according to the access policy specification pol. By

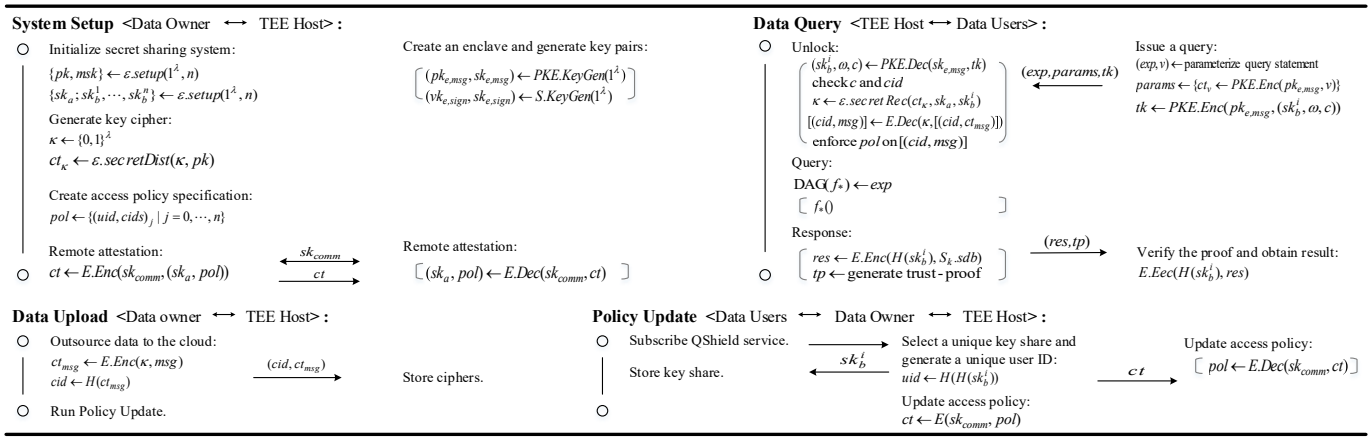

Fig. 3. An overview of the QShield protocols. Codes in square brackets are run by enclaves.

design, the *unlock* process is encapsulated as a TEE function, thus its unique workflow as stated above can be enforced.

query: The TEE host invokes enclaved operators to perform operations over the authorized *collections* $[(cid, msg)]$. First of all, The TEE host transforms the query expression exp into a query plan. A running example for such transformation is shown in Appendix A. In accordance with the query plan, the TEE host begins to schedule enclaved operators f_* for operation. As this process proceeds, several intermediate results, i.e., computational states $S_0, \cdots, S_k$, are produced. Correspondingly, the enclave goes through a series of FSM states $Q_0, \cdots, Q_k$.

response: The TEE host responds the data user with regard to the query. After the enclave reaches the final FSM state Q_k, it can construct a response for the current query, which includes two parts: query result and trust-proof. Specifically, the enclave first hashes the data user's key share sk_b^i to produce a symmetric key, and encrypts the workload field (denoted by sdb) in S_k with the key to generate a query result res, i.e., $res \leftarrow E.Enc(\mathcal{H}(sk_b^i), S_k.sdb)$; it then makes use of all computational states, i.e., $S_0, \cdots, S_k$, to create a trust-proof tp for the query as described in Section IV-B2; at last, the enclave returns res along with tp to the TEE host for query response. By design, the *response* process is encapsulated as a TEE function, thus its unique workflow as stated above can be enforced.

When the data user gets the response, he/she can obtain the query result by executing $E.Dec(\mathcal{H}(sk_b^i), res)$, and audit it by verifying the trust-proof tp.

Policy Update. This protocol handles all events related to policy update, including user add/remove and access permission modification. When a new data user j $(1 \leq j \leq n)$ joins in the system, the data owner first selects a unique key share sk_b^j and securely sends it to the data user. (When $j > n$, $\mathcal{E}$ scheme can be extended to offer an auxiliary interface for the data owner to compute a new unique token.) Then, it defines a set of *collections* that can be accessed by the data user and hashes the sk_b^j twice to generate a unique ID uid for the data user, i.e., $uid \leftarrow \mathcal{H}(\mathcal{H}(sk_b^j))$. At last, the data owner updates the access policy specification pol in the enclave with a new item $(uid, cids)$ through their previously established secure channel. Notably, the data owner runs $ct \leftarrow E.Enc(sk_{comm}, pol)$ and the enclave in turn runs $pol \leftarrow E.Dec(sk_{comm}, ct)$. As for the remaining two circumstances, i.e., user revocation and access permission modification, the data owner only needs to either delete or alter the corresponding item in pol for the specific data user through the secure channel. Besides, when the data owner decides to alter the cloud AS, it can just terminate to pay for the current service, without worrying that the key share left in the enclave will cause the full decryption key leakage. The key revocation issue is effectively circumvented.

Remark: One potential issue here is that, given an untrusted cloud platform including the TEE host, the update command may not be received by an enclave. Intuitively, it can be solved by just requiring a response for each update from the enclave; the data owner can continue to send the update command until it receives a response. This is an effective way to protect from network failure. However, when the TEE host is compromised, such a method cannot guarantee on-demand policy update promise for the data owner. An exciting way is to enhance the Policy Update protocol by exploiting the "heartbeat" idea proposed in our previous work [13]. It is able to force the enclave to be unavailable if the enclave does not receive a valid heartbeat from the data owner after the defined time window; we can attach the policy specification pol to heartbeats such that the update command can be delivered timely and affirmatively to the enclave.

V. Security Analysis

In this section, we formally analyze a number of security and trust properties of QShield, i.e., confidentiality of crypto key and outsourced data, scalable multi-user query control and trusted query execution. Through workflow and dataflow analysis, we reduce each property to several primitives with provable security.

A. Confidentiality of crypto key and outsourced data

This property is guaranteed by both the SGX TEE and $\mathcal{E}$ scheme.

At System Setup, the data owner generates an encryption key κ and key shares sk_a, $sk_b^i (i = 1, \cdots, n)$ (using

$\mathcal{E}.sharesGen(\cdot)$). Through remote attestation, it establishes a secure communication channel with the enclave, under which sk_a is escorted (using $E.enc(\cdot)$ and $E.dec(\cdot)$). In this paper, we do not discuss how the data owner establishes secure communication channels with data users, instead, we assume $sk_b^i (i = 1, \cdots, n)$ are securely delivered to authorized data users. At Data Upload, data are encrypted with κ (using $E.Enc(\cdot)$) before being outsourced to the cloud AS. At this moment, since the enclave and data users just have their own key shares, no entities are able to recover outsourced data. At Data Query, a cryptographically protected token securely brings sk_b^i to the enclave (using $PKE.Enc(\cdot)$ and $PKE.Dec(\cdot)$). Within the enclave, κ is reconstructed with sk_b^i and sk_a (using $\mathcal{E}.secretRec(\cdot)$) to decrypt the loaded ciphertexts $[(cid, ct_{msg})]$ (using $E.Dec(\cdot)$); after that, κ is erased. In subsequent operations, the recovered plaintexts never leave the enclave; the query result sent back to the data user is also cryptographically protected (using $E.Enc(\cdot)$ and $E.dec(\cdot)$). Here, please notice that, we use $\mathcal{H}(sk_b^i)$ as the encryption key. Since $\mathcal{H}(\cdot)$ is a one-way function, one cannot utilize the public uid, i.e., $\mathcal{H}(\mathcal{H}(sk_b^i))$, to infer such a key. In other words, only the data user can decrypt the query result. Besides, the trust-proof involves no sensitive information.

Due to the fact that E is an authenticated symmetric key encryption scheme, PKE is an indistinguishability security under chosen plaintext attack (IND-CPA) public key encryption scheme, and data run within enclave are kept secret, the confidentiality of the crypto key and outsourced data are realized as long as the secret sharing $\mathcal{E}$ scheme is secure.

We state informally that the proposed secret sharing scheme is secure if 1) for any involved participant with a valid secret share, it cannot individually recover the full crypto key; and more generally, 2) for any combination of data users with valid secret shares, they are also unable to recover the full crypto key. We define a Selective-Participant (SP) model of security in Figure 4 for $\mathcal{E}$ scheme. The advantage of an adversary $\mathcal{A}$ in this game is $Pr[b' = b] - \frac{1}{2}$. Our SP attack game is similar to the models in [41]–[43], with the exception that the adversary in the game is only allowed to query for key shares of a specific set of participants, i.e., the set does not simultaneously involve the enclave and data users. It is provable that the security of $\mathcal{E}$ scheme in the Selective-Participant model reduces to the hardness of the Decisional BDH assumption. Please refer to Appendix D for detailed formal proofs.

B. Scalable multi-user query control

QShield exploits a straightforward model of access control list (ACL) to realize multi-user query control. In the access policy pol defined by the data owner, each item $(uid, cids)$ specifies the data (*collections* with IDs $cids$) that can be accessed by the data user with ID uid.

Authentication & Authorization. With remote attestation, the data owner verifies the authenticity of the enclave in the cloud and meanwhile establishes a secure communication channel between them. It is also assumed to have authenticated secure communication channels with data users. In order to authorize query permissions to data users, the data owner assigns each

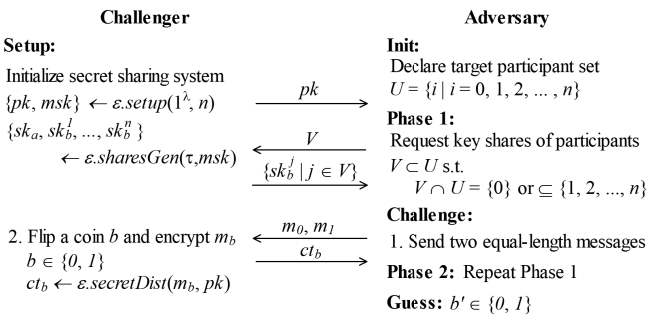

Fig. 4. The Selective-Participant (SP) model of $\mathcal{E}$ scheme security.

of them a unique secret share sk_b^i, and in order to make such an authorization effective, the data owner hands out a special secret share sk_a along with the access policy pol to the enclave. The entrusted sk_a and sk_b^i in turn ensure the enclave and the data user to authenticate with each other. The reasons are two-folds. First, the secret sharing scheme, proved to be secure under Selective-Participant model in Section V-A, guarantees that only valid secret shares can correctly recover the crypto key and the probability of forging a valid secret share is negligible; the authenticity of the two entities with valid secret shares can thus be guaranteed. Second, recall that the uid is generated by hashing the sk_b^i twice, so the specific data user with sk_b^i can be correctly identified in pol. As such, QShield has no need to rely on cumbersome remote attestation or expensive Public Key Infrastructure (PKI) for authentication between the enclave and data users.

Enforcement on Query Control. We remark that the query control is correctly enforced in QShield if the following statements hold: 1) The probability that a query issued by a valid data user can compute over unauthorized data is negligible. 2) The probability that a query issued by an illegal entity can compute over outsourced data is negligible.

Next, we show that such query control is guaranteed by the adopted SGX and the secret sharing scheme $\mathcal{E}$.

By design, a query issued by a valid data user involves a specific cryptographical token $tk \xleftarrow{PKE} (sk_b^i, \omega, c)$. In the *unlock* step at Date Query, the enclave holding the key share sk_a handles the token so as to recover the encryption key κ, then to decrypt loaded ciphertexts $[(cid, ct_{msg})]$, and further to enforce the query control policy pol on behalf of the data owner. These operations in *unlock* are encapsulated into one TEE function, so its correctness and integrity in the run time can be guaranteed by remote attestation. Besides, since they are serialized, i.e., the input of the latter operation is the output of the former one, and the initial operation of κ recovery is correct as proved in Appendix C, the *unlock* function will definitely output authorized data abiding by the pol (Statement 1 is satisfied). According to the security guarantee of the secret sharing scheme, an illegal entity cannot forge a valid crypto share used for constructing the token. The underlying authentication & authorization mechanism then guarantees that the invalid data user will not be recognized by the enclave and thus has no query privilege on outsourced data. Furthermore, we use a freshness factor c to block up launching replay attacks

TABLE I
COMPUTATIONAL COMPLEXITY ANALYSIS ON $\mathcal{E}$ SCHEME

Algorithm	Overhead	Complexity
$\mathcal{E}.setup(\cdot)$	$(n+1)\cdot EX$	$O(n)$
$\mathcal{E}.sharesGen(\cdot)$	$n\cdot EX$	$O(n)$
$\mathcal{E}.secretDist(\cdot)$	$n\cdot EX$	$O(n)$
$\mathcal{E}.secretRec(\cdot)$	$2\cdot BP+3\cdot EXP$	$O(1)$

Notes: EX = Exponentiation operation; BP = Bilinear map operation.

where an illegal entity, e.g., untrusted TEE host, implicitly impersonates a valid data user by leveraging the token in previous issued quires. (Statement 2 is satisfied.)

The above analysis justifies that our secret sharing scheme combined with the SGX TEE can effectively realize multi-user query control in QShield. Now we claim that the query control mechanism is also scalable over a large number of data users.

Support on Scalable Data Users. As the total number of data users n scales, the work in performing mutual authentication, i.e., crypto key reconstruction by the $O(1)$ $secretRec(\cdot)$ algorithm as shown in Table I, does not depend on n. Besides, the enclave only needs to hold one specific secret share sk_a, which is also independent of n. To enable authorization, the enclave requires to perform lookups on a policy list with n items. Since each item in the list only takes a few bytes of storage space, it will not become a bottleneck of query control mechanism on modern commodity servers as n increases. Through hash indexing, the $O(n)$-complexity of lookup operations can be reduced to $O(1)$, being independent of n. To briefly summarize, the scalable data users do not increase the complexity of the authentication & authorization in query control. Moreover, the enforcement on query control and policy update, as analyzed above, does not rely on the number of data users, either. Hence, support on scalable data users is achieved in QShield.

C. Trusted query execution

For each query issued by a valid data user, the untrusted TEE host takes the responsibility to explain and execute it. Through following analysis, we show that QShield can guarantee the trustworthiness of query execution under such an hostile environment. We use two statements to clearly define the trustworthiness: 1) The probability that a data user receives an incorrect query result for a valid query is negligible; and 2) The probability that the TEE host performs operations over the underlying data beyond the scope of a valid query is negligible.

Query Execution Workflow & Dataflow. In view of the control plane of a query execution, the TEE host first transforms the query into a query plan and then accordingly invokes TEE interfaces to physically execute it, which include the *unlock*, computational operators f_*, and the *response*. With regard to the data plane, the enclave, with encrypted message $[(cid, ct_{msg})]$ as input, produces an initial computational state S_0 under the function of *unlock*, and then produces several computational states $S_1, \cdots, S_k$ under functions of f_*, and at last outputs a triple (res, tp, σ_{tp}) by the *response*.

The SGX's *remote attestation* ensures that the TEE interfaces, i.e., *unlock*, f_*, and *response*, are correctly executed at runtime, and its *memory isolation* ensures the integrity of those computational states $S_0, \cdots, S_k$. Thus, the two statements with regard to the trustworthiness of query execution hold, as long as the TEE host honestly invokes relevant TEE interfaces. We justify that such honest behavior of the TEE host is enforced by the trust-proof mechanism.

The enforcement is implemented in two ways. First, recall that a valid query token involves an endurance indicator ω, which is computed based on the query itself and set up by the data user. When a successful invocation of an enclaved operator drives a FSM state transition, w (initialized as ω) in all existing computational states will be reduced by 1; upon reaching the final FSM state, w will be reduced to 0, indicating that the computational states cannot be accessed any more. Suppose the host TEE invokes some (even one) enclaved operators beyond the scope of the query, the enclave will be unable to construct a valid trust proof, because the desired final FSM state cannot be reached due to insufficient w (consumed by the undesired invocations). Second, an enclaved operator in essential produces a new computational state from the previously existing ones. Such a computational state transition is recorded by the enclave as an execution trace of the enclaved operator, the correctness of which, as mentioned above, is guaranteed by SGX's *remote attestation*. When the query execution finishes, i.e., reaching a final FSM state, all the recorded computational state transitions inherently constitute a trustworthy execution trace tp for the current query as a proof. The enclave also generates a signature for the trust-proof to ensure its genuineness. The verifiability of the response triple, i.e., (res, tp, σ_{tp}), ensures that malicious TEE function invocation can be detected by the data user.

Verification on Response Triple. The trust-proof tp reflects how the query result res is derived from the original messages. It is straightforward and effective to judge whether or not such an execution trace affiliates to the issued query by graph matching. The existentially unforgeable signature scheme $\mathcal{S}$ adopted in QShield guarantees that the TEE host has negligible possibility to fabricate an acceptable execution trace with valid enclave signature for an incorrect query result.

VI. IMPLEMENTATION

A prototype of QShield was implemented by integrating it into Opaque [11], an SGX-enabled distributed data analytics platform built on top of Spark SQL. The QShield prototype exploits relational tables as a down-to-earth data model. QShield source code has been released in Github [1].

Following the design philosophy of the Opaque system, we implemented the QShield operators by extending Catalyst APIs of Spark and enabled these operators (in java) to call corresponding enclave implementations (in c/c++) through Java Native Interface (JNI) to accomplish their specific execution logic. As such, QShield can utilize the off-the-shelf Spark SQL engine to support SQL-like queries. Moreover, we applied Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) to ensure data confidentiality and integrity outside the enclave.

[1]QShield project homepage: https://github.com/fishermano/qshield.git

On the other hand, we developed the unique functionalities proclaimed by QShield, i.e., multi-user query control and trustworthy TEE function invocation. First, we implemented the secret sharing scheme $\mathcal{E}$ as an enclave-safe library. This is not a trivial task due to the following reasons: 1) the SGX SDK at the time of this paper's writing does not provide sufficient fundamental libraries to support powerful pairing-based computations; 2) the migration of the Stanford Pairing-Based Cryptography (PBC) library requires to rewrite three non-decoupled I/O functional interfaces that operate on a non-enclave-safe C File* type, i.e., *out_str*, *snprint* and *out_info*; 3) the above migration requires to safely implement *rand* and *mem_cpy* functions invoked by the native PBC library, which, however are not supported by the customized C version of SGX SDK. Second, we re-implemented Opaque operators in QShield to make themselves record their own execution trace, so as to facilitate the assembling of a trust-proof. Third, we implemented two special QShield functions named *ACPolicyApplied* and *ResRespond*. The former enforces access control policies over input data before query execution, corresponding to the *unlock* step in Data Query as described in Section IV-C; the later constructs a final query response to a data user after query execution, corresponding to the *response* step in Data Query. Notably, all computational states produced by QShield operators during query execution are protected with AES[GCM] encryption, i.e., these data are stored outside enclaves in the form of encryption. Fourth, we encapsulated the QShield core functionalities into a RESTful ready-to-use service, which offers luxuriant and light-weighted web URIs to be easily used by the data user to access the QShield system.

VII. Performance Evaluation

In this section, we first describe the experimental settings of QShield evaluation. Then, we demonstrate that QShield outperforms CryptDB [2], a representative pure-crypto-based SQL query system, with a significant performance improvement. In addition, we quantify QShield's overhead by comparing it with the Opaque, a baseline system. Finally, we give a comprehensive evaluation on the achieved multi-user query control and the execution integrity (trustworthy TEE function invocation), and show that both underlying mechanisms have excellent performance in terms of computation, storage, and communication.

A. Experimental Settings

We performed experimental evaluation on an SGX-enabled platform that owns an Intel Kaby Lake i7-7700 processor (4 cores @ 3.60GHz) with 16GiB of RAM and runs Ubuntu 16.04 operating system.

We chose a pairing curve of type A for the secret sharing scheme in QShield and setup a 128-bit security parameter for underlying crypto primitives, e.g., AES[GCM]. All experimental tests were repeated 20 times to eliminate system errors. In the tests, the prototyped QShield adopted a 128-bit AES to protect data outside enclaves during query processing. As well-known, SGX Memory Encryption Engine (MEE) uses a 128-bit AES key. Thus, our implemented QShield provides a 128-bit security level. Similarly, Opaque also has a 128-bit security level. CryptDB is a hybrid cryptosystem and its weakest link is a 128-bit AES primitive, so it offers the same level of security as QShield and Opaque.

To evaluate data query efficiency, we use CryptDB and Opaque as benchmarks. CryptDB is a cryptography-based milestone system that elaborately assembles crypto primitives to protect the confidentiality of query data; Opaque utilizes hardware-assisted TEE to improve query efficiency while achieving the same security goal as CryptDB. We recompiled both CryptDB and Opaque on the above platform and tested operation time of three queries (denoted by Q1, Q2, Q3, respectively) over Big Data Benchmark [2], which is a popular benchmark for big data SQL engines. Concretely, Q1 is a filter operation, Q2 only contains an aggregation operation, and Q3 is composed of filter, aggregation and join. To test the execution time of the trust-proof mechanism, we run a simplified version of QShield over Big Data Benchmark and compared QShield's execution time with the case that strips off the trust-proof functionality.

To test the execution time of the secret sharing scheme, we run the *unlock* function in an enclave with different sizes of input data, ranging from 10-bytes to 1M-bytes, to simulate various data being processed. Herein, we used a native AES[GCM] algorithm realized by SGX as a baseline, which by comparison lacks access control related computations.

B. Query Efficiency

Figure 5 shows an overall performance of QShield, compared with Opaque and CryptDB with regard to different types of queries and different sized databases: tiny database (totally 11197 rows in 2 tables), medium database (totally 25994 rows in 2 tables), big database (totally 51988 rows in 2 tables). We can observe that: 1) QShield achieves comparable performance with Opaque with regard to all tested query types, which implies that the introduced secret sharing and trust-proof mechanisms incur minimum overhead. The reason why QShield runs a litter faster than Opaque is that we optimized the underlying structure of the computational states in implementation. 2) Given Q1 and Q3, CryptDB has a fiercer performance degradation than QShield as the size of database scales up. Notably, when fueled with a big database, QShield's computational performance greatly outperforms CryptDB's. Due to limited operations supported by the adopted homomorphic encryption primitive of CryptDB, CryptDB currently cannot support Q2, which involves a *substring()* function. More generally, CryptDB has no flexible support on arbitrary self-defined functions. Nevertheless, QShield has no such limitation.

C. Evaluation on Multi-user Query Control

Table II shows the main overhead of computation, storage and communication introduced by supporting multi-user query control in QShield (secreting sharing) and EnclaveDB (strawman). We can observe that the proposed secret sharing scheme

[2]Big data benchmark: https://amplab.cs.berkeley.edu/ benchmark/

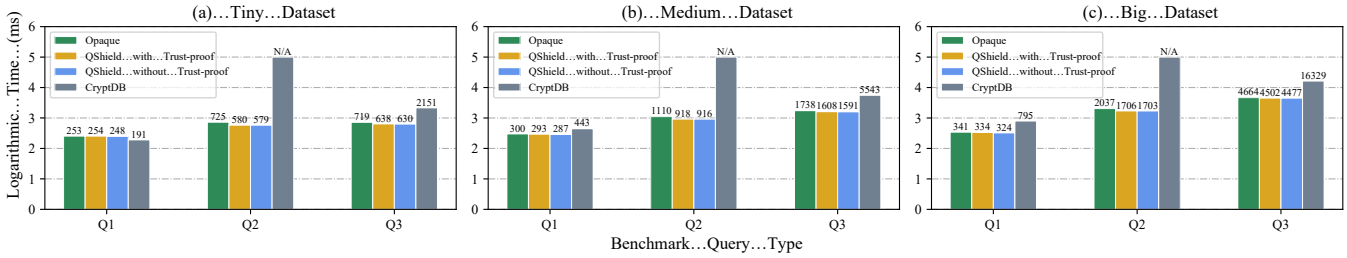

Fig. 5. Query efficiency of QShield. Logarithmic time is used to show the difference in order of magnitudes. Those number labels indicate corresponding execution time in (a) (b) (c). N/A is abbreviated for Not Applicable.

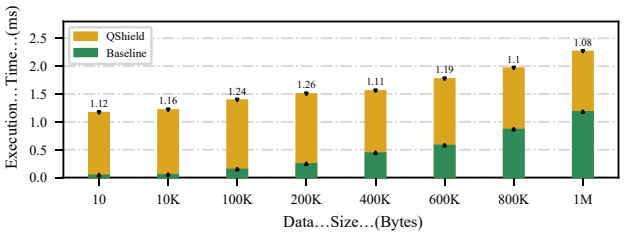

Fig. 6. The performance of the implemented *unlock* function. Those number labels indicate the performance gap between tested systems.

incurs no computational cost to data users and greatly reduces the number of remote attestations (from $n+1$ to 1), compared with EnclaveDB's solution. However, it imports an additional cost for key reconstruction to an enclave as per query. In order to depict how such an overhead impacts query performance, we measured the execution time of the implemented *unlock* function, which, as an access policy enforcement point for multi-user query control, invokes $\mathcal{E}.secretRec(\cdot)$ to enforce authentication and authorize access. Figure 6 shows the performance of the *unlock* function. We observe that the overhead of key reconstruction is very small (about 1.1 ms). Thus, the key reconstruction operation does not ruin query efficiency.

Given storage cost at different parties, our solution is also the lowest. Notably, QShield requires an enclave to just hold one key element for controlling multi-user access while EnclaveDB requests an enclave to store at least n crypto session keys (i.e., $n \times 128$ bits), which results in a non-negligible burden to the enclave in view of EPC limit. Meanwhile, the corresponding storage overhead for key management in QShield is constant, but it is linearly proportional to the user scale n in EnclaveDB.

With regard to communication overhead, QShield totally requires $n+1$ point-to-point unicast communications for the distribution of secret shares to an enclave and n data users. EnclaveDB, however, needs $(n-1)!+n$ point-to-point unicast communications to mutually exchange data users' public keys.

To sum up, QShield achieves scalable multi-user query control with superior performance in computation, storage and communication.

D. Evaluation on Execution Integrity

Table III shows the main overhead of computation, storage and communication introduced by supporting execution integrity in QShield (trust-proof) and Opaque (self-verification). Given that $\hat{m} \gg 1$, $|\omega| = 32$ and $|SDAG| = 128$, the trust-proof mechanism in QShield has a lower communication cost and a lower storage cost at the side of enclave than Opaque, while the storage cost at the data user side is a bit higher, but at the same level of cost. By Referring to Figure 5, we can conclude that the trust-proof mechanism in QShield raises almost no influence on the overall performance of queries. Since Opaque's prototype at present does not implement the self-verification protocol proposed in [14], it is hard to perform a tested comparison of computational efficiency with QShield. Instead, we implemented a DAG tool that supports the DAG matching (MDAG) operation in Opaque and the DAG updating (UDAG) operation in QShield. Figure 7 demonstrates that QShield has a significant performance improvement regarding to query execution at the enclave.

VIII. Conclusion

In this paper, we proposed a secure and efficient SQL-style query framework called QShield for outsourced data in the cloud. It utilizes the off-the-shelf hardware-assisted TEE, i.e., Intel SGX, to protect the confidentiality and integrity of sensitive data being queried. In order to make QShield capable of enforcing query control over multiple data users in a scalable way, we proposed a novel secret sharing scheme, with which cumbersome authentication through remote attestation per data user is avoided. Meanwhile, it greatly alleviates attacks listed in our threat model. Moreover, we introduced a trust-proof mechanism in QShield to guarantee the correctness of query results and further reduce the possibility to extract sensitive information from TEE. We implemented a prototype of QShield and demonstrated that it is feasible in practice. With comprehensive evaluation in terms of both security and execution performance, we show that QShield achieves fundamental security properties, i.e., confidentiality of crypto key and outsourced data, scalable multi-user query control and trusted query execution, while raising no significant performance degradation.

One limitation of QShield is it adopts a coarse-grained ACL model; though being efficient, it makes QShield unsuitable for complex query control policies. Furthermore, we assume

TABLE II
ANALYSIS ON MULTI-USER QUERY CONTROL

Systems \ Metrics	Computational Overhead			Storage Overhead			Communication Overhead
	Data Owner	Data User	Enclave	Data Owner	Data User	Enclave	
EnclaveDB	$1{\cdot}(RA{+}CO)$	$1{\cdot}(RA{+}CO)$	$(n+1)\cdot RA$	$n{\cdot}\|PK\|{+}1{\cdot}\|BI\|$	$n\cdot\|PK\|+1\cdot(\|SK\|+\|BI\|)$	$n{\cdot}\|CK\|$	$((n-1)!+n)\cdot\|PK\|$
QShield	$1{\cdot}(RA{+}CO)$	-	$1{\cdot}RA{+}N{\cdot}KR$	$1{\cdot}\|EK\|{+}1{\cdot}\|BI\|$	$1\cdot\|KS\|$	$1\cdot\|KS\|$	$(n+1)\cdot\|KS\|$

Notes: RA = Remotation attestation operation; CO = Enclave compilation operation; KR = Key reconstruction operation; $|PK|$ = Public key size; $|SK|$ = Private key size; $|BI|$ = Enclave binary size; $|CK|$ = Session key size; $|EK|$ = Symmetric encryption key size; $|KS|$ = Key share size; PK, SK, EK, KS have the same length; n is the user scale of the system; N is the total number of queries issued by data users.

TABLE III
ANALYSIS ON EXECUTION INTEGRITY

Systems \ Metrics	Computational Overhead		Storage Overhead		Communication Overhead
	Data User	Enclave	Data User	Enclave	
Opaque	$1\cdot GDAG$	$m\cdot MDAG$	$1\cdot\|DAG\|$	$\hat{m}\cdot\|DAG\|$	$\hat{m}\cdot\|DAG\|$
QShield	$1\cdot GDAG+1\cdot MDAG$	$m\cdot UDAG$	$1\cdot\|DAG\|+1\cdot\|\omega\|$	$1\cdot\|DAG\|+1\cdot\|\omega\|$	$1\cdot(\|\omega\|+\|DAG\|+\|SDAG\|)$

Notes: $GDAG$ = DAG generation operation; $MDAG$ = DAG matching operation; $UDAG$ = DAG updating operation; $|DAG|$ = DAG size; $|\omega|$ = Endurance indicator size; $|SDAG|$ = DAG signature size; m is the number of DAG nodes; $\hat{m}$ is the number of workers.

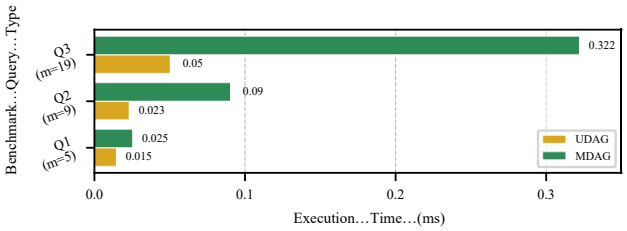

Fig. 7. The performance of DAG matching and updating. m is the number of DAG nodes.

parametric query expressions are not sensitive. Possibly, they could be analyzed to track user behaviors through side-channel attacks. How to defend this security threat is left for future work.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672

[2] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: Protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 85–100. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043566

[3] T. T. A. Dinh and A. Datta, "Streamforce: Outsourcing access control enforcement for stream data to the clouds," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '14. New York, NY, USA: ACM, 2014, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/2557547.2557556

[4] C. Thoma, A. J. Lee, and A. Labrinidis, "Polystream: Cryptographically enforced access controls for outsourced data stream processing," in *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*, ser. SACMAT '16. New York, NY, USA: ACM, 2016, pp. 227–238. [Online]. Available: http://doi.acm.org/10.1145/2914642.2914660

[5] W. Ding, R. Hu, Z. Yan, X. Qian, R. H. Deng, L. T. Yang, and M. Dong, "An extended framework of privacy-preserving computation with flexible access control," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2019.

[6] W. Ding, Z. Yan, X. Qian, and R. H. Deng, "Computing maximum and minimum with privacy preservation and flexible access control," in *2019 IEEE Global Communications Conference (GLOBECOM)*, Dec 2019, pp. 1–7.

[7] W. DING, Z. Yan, and R. H. Deng, "Privacy-preserving data processing with flexible access control," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 2, pp. 363–376, March 2020.

[8] W. Sun, R. Zhang, W. Lou, and Y. T. Hou, "Rearguard: Secure keyword search using trusted hardware," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, April 2018, pp. 801–809.

[9] R. Gennaro, "Verifiable outsourced computation: A survey," in *the ACM Symposium*, 2017.

[10] B. Fuhry, R. Bahmani, F. Brasser, F. Hahn, F. Kerschbaum, and A.-R. Sadeghi, "Hardidx: Practical and secure index with sgx," in *Data and Applications Security and Privacy XXXI*. Cham: Springer International Publishing, 2017, pp. 386–408.

[11] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 283–298. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng

[12] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *2015 IEEE Symposium on Security and Privacy (SP)*, May 2015, pp. 38–54.

[13] Y. Chen, W. Sun, N. Zhang, Q. Zheng, W. Lou, and Y. T. Hou, "A secure remote monitoring framework supporting efficient fine-grained access control and data processing in iot," in *Security and Privacy in Communication Networks*. Cham: Springer International Publishing, 2018, pp. 3–21.

[14] ——, "Towards efficient fine-grained access control and trustworthy data processing for remote monitoring services in iot," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 7, pp. 1830–1842, 2018.

[15] M. Xu, A. Papadimitriou, A. Haeberlen, and A. Feldman, "Hermetic: Privacy-preserving distributed analytics without (most) side channels." unpublished, 2019. [Online]. Available: http://www.cis.upenn.edu/ ahae/papers/hermetic-tr.pdf

[16] Y. Swami, "Intel sgx remote attestation is not sufficient." unpublished, 2018. [Online]. Available: https://eprint.iacr.org/2017/736.pdf

[17] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A. reza Sadeghi, "The guard's dilemma: Efficient code-reuse attacks against intel SGX," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 1213–1227. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/biondo

[18] C. Tan, L. Yu, J. B. Leners, and M. Walfish, "The efficient server audit problem, deduplicated re-execution, and the web," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 546–564. [Online]. Available: https://doi.org/10.1145/3132747.3132760

[19] X. Dawn, D. Song, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proceeding 2000 IEEE Symposium on Security and Privacy (SP)*, May 2000, pp. 44–55.

[20] E.-J. Goh, "Secure indexes," Cryptology ePrint Archive, Report 2003/216, 2003, https://eprint.iacr.org/2003/216.

[21] Y.-C. Chang and M. Mitzenmacher, "Privacy preserving keyword searches on remote encrypted data," in *Applied Cryptography and Network Security*, J. Ioannidis, A. Keromytis, and M. Yung, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 442–455.

[22] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky, "Searchable symmetric encryption: Improved definitions and efficient constructions," *Journal of Computer Security*, vol. 19, pp. 895–934, 01 2011.

[23] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic searchable symmetric encryption," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 965–976. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382298

[24] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *Advances in Cryptology - EUROCRYPT 2004*, C. Cachin and J. L. Camenisch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 506–522.

[25] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for boolean queries," in *Advances in Cryptology – CRYPTO 2013*, R. Canetti and J. A. Garay, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 353–373.

[26] W. Sun, N. Zhang, W. Lou, and Y. T. Hou, "When gene meets cloud: Enabling scalable and efficient range query on encrypted genomic data," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, May 2017, pp. 1–9.

[27] Y. Gahi, M. Guennoun, and K. Elkhatib, "A secure database system using homomorphic encryption schemes," *Computer Science*, pp. 54–58, 2011.

[28] S. Bajaj and R. Sion, "Trusteddb: A trusted hardware-based database with privacy and data confidentiality," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 3, pp. 752–765, March 2014.

[29] C. Priebe, K. Vaswani, and M. Costa, "Enclavedb: A secure database using sgx," in *Proceeding 2018 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2018, pp. 264–278.

[30] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh, "Shieldstore: Shielded in-memory key-value storage with sgx," in *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*, Dresden, Germany, 03 2019. [Online]. Available: https://doi.org/10.1145/3302424.3303951

[31] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security*, ser. EuroSec'17. New York, NY, USA: ACM, 2017, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/3065913.3065915

[32] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2421–2434. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134038

[33] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," *CoRR*, vol. abs/1611.06952, 2016. [Online]. Available: http://arxiv.org/abs/1611.06952

[34] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, p. 991–1008. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck

[35] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs," in *Network and Distributed System Security Symposium*, 01 2017.

[36] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 431–446. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831143.2831171

[37] B. Dan and F. Franklin, "Identity-based encryption from the weil pairing," in *Advances in Cryptology — CRYPTO 2001*, J. Kilian, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 213–229.

[38] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: ACM, 2013, pp. 10:1–10:1. [Online]. Available: http://doi.acm.org/10.1145/2487726.2488368

[39] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for cpu based attestation and sealing," *Workshop on Hardware and Architectural Support for Security and Privacy*, 2013. [Online]. Available: https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing

[40] V. Costan and S. Devadas, "Intel sgx explained," *Cryptology ePrint Archive, Report 2016/086*, 2016. [Online]. Available: https://eprint.iacr.org/2016/086

[41] H. Li and L. . Pang, "Provably secure secret sharing scheme based on bilinear maps," *Journal on Communications*, vol. 29, pp. 45–50, 10 2008.

[42] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *In Proc. of ACMCCS'06*, 2006, pp. 89–98.

[43] R. D'Souza, D. Jao, I. Mironov, and O. Pandey, "Publicly verifiable secret sharing for cloud-based key management," in *Progress in Cryptology – INDOCRYPT 2011*, D. J. Bernstein and S. Chatterjee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 290–309.

Yaxing Chen received the B.Eng. degree in Software Engineering from Northwestern Polytechnical University in 2012. He has been a Ph.D. candidate in the Department of Computer Science at Xi'an Jiaotong University since 2014. He was a visiting student at Virginia Tech from 2016 to 2018. His research interests lie on data security and privacy, cloud computing, with focus on data access control and trusted computing.

Qinghua Zheng received the Ph.D. degree in System Engineering from Xi'an Jiaotong University. He is the winner of the National Funds for Distinguished Young Scientists and Distinguished Professor for the "Changjiang River Scholar Project" in China. He is also among the first batch of leading scientists for the "Ten-Thousand Talents Project", the candidate for the "New Century National Hundered Thousand-and-Ten thousand Talents Project" in China. He is currently the leader of the Innovation Team of National Natural Science Foundation of China, the Innovative Team under Ministry of Education, and the Shaanxi Key Scientific and Technological Innovation Team. His major research fields include intelligent e-learning, big data mining and application, and software reliability.

Zheng Yan received the B.Eng. degree in electrical engineering and the M.Eng. degree in computer science and engineering from the Xi'an Jiaotong University in 1994 and 1997, respectively, the second M.Eng. degree in information security from the National University of Singapore in 2000, and the licentiate of science and the doctor of science in technology in electrical engineering from Helsinki University of Technology. She is currently a professor at the Xidian University and a visiting professor at the Aalto University. Her research interests are in trust, security, privacy, and security-related data analytics. Prof. Yan serves as a general or program chair for 30+ international conferences and workshops. She is a steering committee co-chair of IEEE Blockchain international conference. She is also an area editor or an associate editor of many reputable journals, e.g., IEEE Internet of Things Journal, Information Sciences, Information Fusion, JNCA, IEEE Access, SCN, etc.

Dan Liu received her B.Eng. degree in communication engineering from Jilin University in 2017. She is currently pursuing the master's degree with the State Key Laboratory on Integrated Services Networks, Xidian University. Her research interests are in data analytics, data access control and edge computing.

APPENDIX A
QUERY TRANSFORMATION EXAMPLE

A running example for query transformation is illustrated in Figure 8, where a query is executed over two *collections* $C_1[A_1, A_3, A_5]$, $C_2[A_2, A_3, A_4]$ and has one query parameter a. The query first filters out from C_1 the *documents* that the value of A_1 is less equal than a. Then, it performs projection of C_1 and C_2 on A_3 and $[A_3, A_4]$, respectively, and subsequently joins the two *collections* under the condition of $C_1[A_3] = C_2[A_3]$. At the end, the query computes the sum of $C_2[A_4]$.

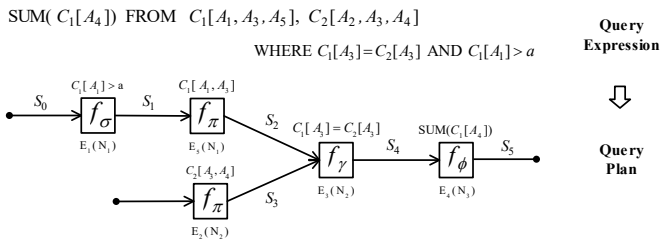

Fig. 8. A running example for query transformation

APPENDIX B
EXTENSIONS TO DISTRIBUTED MODE

With data becoming available in large quantities and system requiring high throughputs, a common practice for the cloud AS is exploiting distributed computation, where tasks are processed across multiple server nodes in the cloud. We remark that QShield can be extended to support such a scenario. To this end, we introduce the notions of *broker* and *worker* in QShield: A *broker* is a dedicated enclave created for the data owner; A *worker* is a general enclave serving all data users, which implements one of query operators f_π, f_σ, f_ϕ, f_γ. As illustrated in Fig. 8, we have $E_1(\mathrm{N}_1)$, $E_2(\mathrm{N}_2)$, $E_3(\mathrm{N}_2)$, $E_4(\mathrm{N}_3)$, $E_5(\mathrm{N}_1)$, where $E_x(\mathrm{N}_y)$ represents the x^{th} *worker* in the y^{th} server node.

Compared with the stand-alone mode, QShield protocols in the distributed mode have following changes:

- Interactions with the enclave in all original protocols are replaced by with the *broker*.
- At **System Setup**, the *broker*, on behalf of the data owner, validates the intactness of codes of the *workers* and the credibility of their hosting SGX-enabled platforms through remote attestation, and meanwhile builds interconnected secure channels with all *workers* by negotiating a common communication key.
- At **Data Query**, after the *broker* creates an initial computational state S_0 for the current query, the TEE host based on the query plan successively schedules distributed *workers* for operation, instead of invoking enclaved operators implemented by a single enclave instance. Notably, once a *worker* completes its task, it will generate a new computational state S_j $(j = 1, \cdots, k)$ and inform the *broker* to record its execution trace. The last *worker* also forwards the final computational state, i.e., query result, to the *broker*, where the query response then is constructed. All communications between the *broker* and *workers* during above process are made through the established secure channels.

Obviously, the secret sharing scheme can be correctly executed. However, since the consistency of the endurance indicator w in all computational states cannot be guaranteed by the original trust-proof protocol in such a distributed mode, i.e., w in some computational states may not be set as 0 when the whole query execution ends, a TEE host could make use of the computational states to perform malicious computation. A straightforward solution is to let the *broker* check such inconsistency (it can be easily deduced by the collected computational state transitions) after it receives all computational state transitions of *works* to construct the trust-proof. Suppose a desired inconsistency is broken, the *broker* denies to generate a valid trust-proof.

APPENDIX C
PROOF OF THE CORRECTNESS OF $\mathcal{E}$ SCHEME

The secret sharing scheme $\mathcal{E}$ is correct if Statement C.1 holds. Given that the algorithms $\{pk, msk\} \leftarrow setup(1^\lambda, n)$, $\{sk_a; sk_b^1, \cdots, sk_b^n\} \leftarrow sharesGen(\tau, msk)$ and $ct_\kappa \leftarrow secretDist(\kappa, pk)$ of $\mathcal{E}$ are correctly executed.

Statement C.1: *With the secret ciphertext ct_κ, the enclave key share sk_a and one of user key shares sk_b^i, the algorithm $secretRec(ct_\kappa, sk_a, sk_b^i)$ can compute a valid secret κ.*

Proof C.1: According to Equation (1), the $secretRec(\cdot)$ algorithm first decrypts a corresponding leaf node in ct_κ with the enclave key share sk_a and the user key share sk_b^i, respectively. Here, F_t denotes the decrypted value of a node t.

$$\begin{aligned} F_t &= \begin{cases} e(E_i, sk_a), \text{if } i = 0 \\ e(E_i, sk_b^i), \text{otherwise} \end{cases} \\ &= e(E_i, sk^i), \text{for all } i \in \{0, 1, \cdots, n\} \\ &= e(g^{t_i \cdot s}, g^{\frac{q_{t=i}(0)}{t_i}}) \\ &= e(g, g)^{q_{t=i}(0) \cdot s}. \end{aligned}$$

With Equations (2) and (3), it can successively decrypt the OR node and the root node, i.e.,

$$\begin{aligned} F_t &= \prod_{t' \in C(t)} F_{t'}^{\Delta_{j,\bar{C}(t)}(0)} \\ &= \prod_{t' \in C(t)} (e(g, g)^{q_{t'}(0) \cdot s})^{\Delta_{j,\bar{C}(t)}(0)} \\ &= \prod_{t' \in C(t)} (e(g, g)^{q_{parent(t')}(index(t')) \cdot s})^{\Delta_{j,\bar{C}(t)}(0)} \\ &= \prod_{t' \in C(t)} e(g, g)^{q_t(j) \cdot s \cdot \Delta_{j,\bar{C}(t)}(0)} \\ &= e(g, g)^{s \cdot \sum\limits_{t' \in C(t)} q_t(j) \cdot \Delta_{j,\bar{C}(t)}(0)} \\ &= e(g, g)^{s \cdot q_t(0)}. \end{aligned}$$

Once the root node is decrypted, that is, the blinding factor $Y^s = e(g, g)^{y \cdot s}$ is correctly recovered, the algorithm is then definitely able to obtain κ from $ct_\kappa = \kappa \cdot Y^s$.

APPENDIX D
PROOF OF THE SECURITY OF $\mathcal{E}$ SCHEME

Definition 1 (Security of $\mathcal{E}$ Scheme): *The secret sharing $\mathcal{E}$ scheme is secure in the Selective-Participant model of security if all polynomial-time adversaries have at most a negligible advantage in the above game.*

Theorem D.1: *If an adversary can break the $\mathcal{E}$ scheme in the Selective-Participant model, then a simulator can be constructed to play the Decisional BDH game with a non-negligible advantage.*

Proof D.1: Provided that there exists a polynomial-time adversary $\mathcal{A}$ who can break the $\mathcal{E}$ scheme in the Selective-Participant model with advantage ϵ. We build a simulator $\mathcal{S}$ that can play the Decisional BDH game with advantage $\frac{\epsilon}{2}$. What follows is a description of the simulation process.

First of all, the challenger sets two cyclic groups $\mathbb{G}_1$ and $\mathbb{G}_2$ with an efficient bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_1 \rightarrow \mathbb{G}_2$; g is the generator of $\mathbb{G}_1$. The challenger flips a fair binary coin u outside of $\mathcal{A}$'s view: If $u = 0$, the challenger sets $(A, B, C, Z) = (g^a, g^b, g^c, e(g,g)^{a\cdot b\cdot c})$; If $u = 1$, it sets $(A, B, C, Z) = (g^a, g^b, g^c, e(g,g)^z)$; a, b, c, z are random numbers.

Init. The simulator $\mathcal{S}$ executes $\mathcal{A}$, accepting the challenge set of participants, i.e., $U = \{i | i = 0, 1, \cdots, n\}$.

Setup. For all $i \in U$, the simulator $\mathcal{S}$ chooses a random t_i from $\mathbb{Z}_p$ and sets $T_i = g^{t_i}$. The simulator then sets the parameter $Y = e(A, B) = e(g,g)^{a\cdot b}$. At last, it gives the public parameters, i.e, T_i and Y, to $\mathcal{A}$.

Phase 1. The adversary $\mathcal{A}$ makes requests for the key shares corresponding to the participant set V that satisfies $V \cap U = \{0\}$ or $V \cap U \subseteq \{1, \cdots, n\}$. To generate key shares for participants in V, the simulator $\mathcal{S}$ needs to assign a polynomial Q_t of degree d_t for each node t in the access structure τ.

Here, we define a procedure, denoted by *poly(τ_t, λ_t)*, to set up the polynomials for the nodes of an access sub-tree τ_t. λ_t is an integer from $\mathbb{Z}_p$. The procedure first defines a polynomial q_t of degree d_t for the root node t. Specifically, it sets $q_t(0) = \lambda_t$ and then fixes q_t by randomly sets rest of the points. Next, it makes a recursive call with $\tau_{t'}$ and $q_t(index(t'))$ as inputs, i.e., *poly($\tau_{t'}$, $q_t(index(t'))$)*, to set polynomials for each child node t' of t. Notice that in this way, $q_t(0) = q_t(index(t'))$ for each child node t' of t.

By running *poly(τ_r, a)*, The simulator $\mathcal{S}$ first sets up a polynomial q_t for each node t of τ. Then, it defines the final polynomial $Q_t = b \cdot q_t$ for each node t of τ. Notice that, this sets $y = Q_r(0) = a \cdot b$. The key share corresponding to each leaf node is given using its polynomial as follows:

$$sk^i = g^{\frac{Q_t(0)}{t_i}} = g^{\frac{b\cdot q_t(0)}{t_i}} = B^{\frac{q_t(0)}{t_i}}.$$

Hence, $\mathcal{S}$ is able to construct key shares for all participants in V. Moreover, the distribution of these key shares is identical to that in the original scheme.

Challenge. The adversary $\mathcal{A}$ submits two challenge messages m_0 and m_1 to the simulator $\mathcal{S}$. $\mathcal{S}$ flips a fair binary coin v and returns an encryption of m_v, i.e.,

$$ct_{m_v} = \{m_v \cdot Z; \{E_i = C^{t_i}\}_{i\in V}\}.$$

If $u = 0$, then $Z = e(g,g)^{a\cdot b\cdot c}$. If let s equals to c, then $Y^s = (e(g,g)^y)^c = e(g,g)^{a\cdot b\cdot c}$, and $E_i = T_i^s = g^{t_i\cdot c} = C^{t_i}$. Therefore, ct_{m_v} is a valid random encryption of message m_v.

Otherwise, i.e., $u = 1$, we have $Z = e(g,g)^z$. Since z is a random number, $m_v \cdot Z = m_v \cdot e(g,g)^z$ will be a random element of $\mathbb{G}_2$ from $\mathcal{A}$'s view and it contains no information about m_v.

Phase 2. The simulator $\mathcal{S}$ acts exactly as it did in Phase 1.

Guess. The adversary $\mathcal{A}$ submits a guess v' of v. If $v' = v$, the simulator $\mathcal{S}$ will output $u' = 0$, indicating that it receives a valid BDH-tuple; otherwise, $\mathcal{S}$ outputs $u' = 1$, indicating that it receives a random 4-tuple.

When $u = 1$, $\mathcal{A}$ gains no information about v. Thus, we have $Pr[v \neq v' | u = 1] = \frac{1}{2}$. Furthermore, when $v \neq v'$, $\mathcal{S}$ guesses $u' = 1$. As a result, we have $Pr[u' = u | u = 1] = \frac{1}{2}$.

When $u = 0$, $\mathcal{A}$ obtains a valid ciphertext of m_v. The advantage of $\mathcal{A}$ to break the ciphertext is ϵ by definition. Hence, we have $Pr[v = v' | u = 0] = \frac{1}{2} + \epsilon$. Furthermore, when $v = v'$, $\mathcal{S}$ guesses $u' = 0$. As a result, we have $Pr[u' = u | u = 0] = \frac{1}{2} + \epsilon$.

In conclusion, the overall advantage of the simulator $\mathcal{S}$ in the Decisional BDH game is

$$\begin{aligned}\frac{1}{2}Pr[u' = u | u = 0] + \frac{1}{2}Pr[u' = u | u = 0] - \frac{1}{2} \\ = \frac{1}{2} \cdot (\frac{1}{2} + \epsilon) + \frac{1}{2} \cdot \frac{1}{2} - \frac{1}{2} \\ = \frac{\epsilon}{2}.\end{aligned}$$