TRUSTWORTHY DECENTRALIZED KNOWLEDGE GRAPH

ENYUAN ZHOU

PhD

The Hong Kong Polytechnic University

2025

The Hong Kong Polytechnic University
Department of Computing

Trustworthy Decentralized Knowledge Graph

Enyuan Zhou

A thesis submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy
June 2024

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgment has been made in the text.

Signature: _____

Name of Student: Enyuan Zhou

Abstract

A knowledge graph (KG) is a structured representation of information that connects entities and their relationships in a graph format. It is a powerful tool for organizing and querying vast amounts of data, enabling more efficient data retrieval and analysis. In the context of Web 3.0, decentralization plays a crucial role in reshaping how information is managed and shared online. Decentralized Knowledge Graphics (DKG) represent a paradigm shift in knowledge management, where the creation, transferring, and query of KG data are distributed among a network of participants. This decentralized approach promotes collaboration, diversity of perspectives, and collective intelligence, leading to more dynamic and inclusive knowledge ecosystems. However, the openness and decentralization of these networks can also lead to security concerns, including data poisoning attack, leakage of sensitive information and malicious data manipulation. In this thesis, we analyze various security risks in DKG and propose novel solutions to build and manage trustworthy DKG.

First, we focus on DKG construction process, i.e., security issues in Federated Knowledge Graph Embedding (FKGE). FKGE is an emerging collaborative learning technique to construct DKGs. However, poisoning attacks in FKGE, which lead to biased decisions by downstream applications, remain unexplored. We systematize the risks of FKGE poisoning attacks, from which we develop a novel framework for poisoning attacks that force the victim client to predict specific false facts. Based on the attack framework, we explore a blockchain-based defense strategy that can solve this prob-

lem by changing the aggregation paradigm. Second, we consider the data ownership protection in DKG sharing. Without central oversight, it is challenging to enforce restrictions or audit data access and usage effectively across all DKG participants. We propose Pistis, the first DKG provides SPARQL queries with data ownership guarantees. Third, we consider the data integrity and query verifiability requirements in DKG queries. While existing studies focus on ensuring data integrity, how to ensure query verifiability - thus guarding against incorrect, incomplete, or outdated query results - remains unsolved. We propose VeriDKG, the first SPARQL query engine for DKG that offers both data integrity and query verifiability guarantees.

In summary, the thesis aims to build secure and trustworthy decentralized knowledge graphs from the perspective of construction, sharing and query. Theoretical analysis and experimental evaluations demonstrate the performance advantages of our methods with provable security.

Publications Arising from the Thesis

1. Enyuan Zhou, Song Guo, Zicong Hong, Christian S. Jensen, Yang Xiao, Dalin Zhang, Jinwen Liang, and Qingqi Pei, "VeriDKG: A Verifiable SPARQL Query Engine for Decentralized Knowledge Graphs", in *50th International Conference on Very Large Databases (VLDB)*, 2024.

2. Enyuan Zhou, Song Guo, Zhixiu Ma, Zicong Hong, Tao Guo, and PeiranDong, "Poisoning Attack on Federated Knowledge Graph Embedding", in *Proceedings of the ACM Web Conference 2024 (WWW)*, 2024.

3. Enyuan Zhou, Zicong Hong, Yang Xiao, Dongxiao Zhao, Qingqi Pei, Song Guo, Rajendra Akerkar, "MSTDB: A Hybrid Storage-Empowered Scalable Semantic Blockchain Database", in *IEEE Transactions on Knowledge and Data Engineering (TKDE)* (2022).

4. Zicong Hong, Song Guo, Enyuan Zhou, Wuhui Chen, Huawei Huang, and Albert Zomaya. "GriDB: Scaling Blockchain Database via Sharding and Off-Chain Cross-Shard Mechanism", in *49th International Conference on Very Large Databases (VLDB)*, 2023.

5. Zicong Hong, Song Guo, Enyuan Zhou, Jianting Zhang, Wuhui Chen, Jinwen Liang, Jie Zhang, and Albert Zomaya, "Prophet: Conflict-Free Sharding

Blockchain via Byzantine-Tolerant Deterministic Ordering", in *IEEE International Conference on Computer Communications (INFOCOM)*, 2023.

6. Jingyi Tian, Yang Xiao, Enyuan Zhou, Qingqi Pei, " Cross-Chain System Supports Verifiable Complete Data Provenance Queries", in *23rd International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, 2023.

7. Enyuan Zhou, Song Guo, Zicong Hong, Yang Xiao, Yan Zhao, Jinwen Liang, Dalin Zhang, and Kai Zheng, "A Decentralized Knowledge Graph with Verifiable and Ownership-Preserving SPARQL Query", manuscript submitted to *IEEE Transactions on Knowledge and Data Engineering (TKDE)*.

Acknowledgments

I am profoundly grateful for the support, guidance, and encouragement I have received during the course of my PhD journey at the Department of Computing, The Hong Kong Polytechnic University. This thesis would not have been possible without the contributions of many people whom I have had the privilege to work and interact with.

First and foremost, I extend my deepest appreciation to my supervisors, Dr. Jingcai Guo and Prof. Song Guo, whose expertise, understanding, and patience added considerably to my graduate experience. Professor Guo's expertise in the field, combined with his patient mentorship, has been instrumental in shaping my research direction and refining my academic pursuits. His dedication to excellence and commitment to fostering an environment of intellectual curiosity and innovation have profoundly influenced my development as a researcher. I am immensely fortunate to have had the opportunity to work under his mentorship, and I deeply appreciate his contributions to my personal and professional growth.

I also would like to thank my collaborators for their support and discussions. In particular, I would like to express my sincere appreciation to my main collaborator, Mr. Zicong Hong, for his valuable contributions to my research. Second, thanks to Prof. Qingqi Pei, Prof. Yang Xiao and Miss. Zhixiu Ma from Xidian University, Prof. Christian S. Jensen and Prof. Dalin Zhang from Aalborg University, and Dr. Jinwen Liang, Dr. Tao Guo, Dr. Jie Zhang, Mr. Peiran Dong from The Hong Kong

Polytechnic University. Working alongside such dedicated and talented individuals has been an incredibly rewarding experience, and I am grateful for the role they have played in my professional development.

I would like to thank my fellow groupmates and friends at The Hong Kong Polytechnic University for their camaraderie, support, and collaboration throughout my PhD journey. Sharing this path with such a dedicated and talented group of individuals has been both a privilege and a source of immense encouragement.

Lastly, I wish to dedicate this thesis to my parents and girlfriend, whose sacrifices and unwavering support have shaped me into the person I am today. Your endless love and encouragement have been my guiding light, and this achievement is as much yours as it is mine.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The Semantic Web underpins Knowledge Graphs (KGs) as structured frameworks that map out the connections between entities and their relationships, offering a systematic approach to managing and navigating extensive datasets [110, 112, 78]. In the realm of Web 3.0 [43, 80], the emphasis on decentralization significantly alters the landscape of online information handling and exchange. Decentralized Knowledge Graphs (DKGs) signify a transformative approach to knowledge stewardship, characterized by the decentralized creation, sharing, and query of knowledge across a community network [143, 5, 16]. This decentralized model fosters cooperative efforts, embraces a multitude of viewpoints, and harnesses collective wisdom, contributing to more adaptable and inclusive knowledge environments. Such an approach empowers users and communities to jointly create and refine knowledge assets, fostering innovation and collaboration in today's digital era. Figure 1.1 shows DKG's system model and query process. In a DKG system, each data owner trains a local knowledge graph embedding (KGE) model to construct its own KG subgraph, and multiple data owners employ federated knowledge graph embedding (FKGE) to improve the

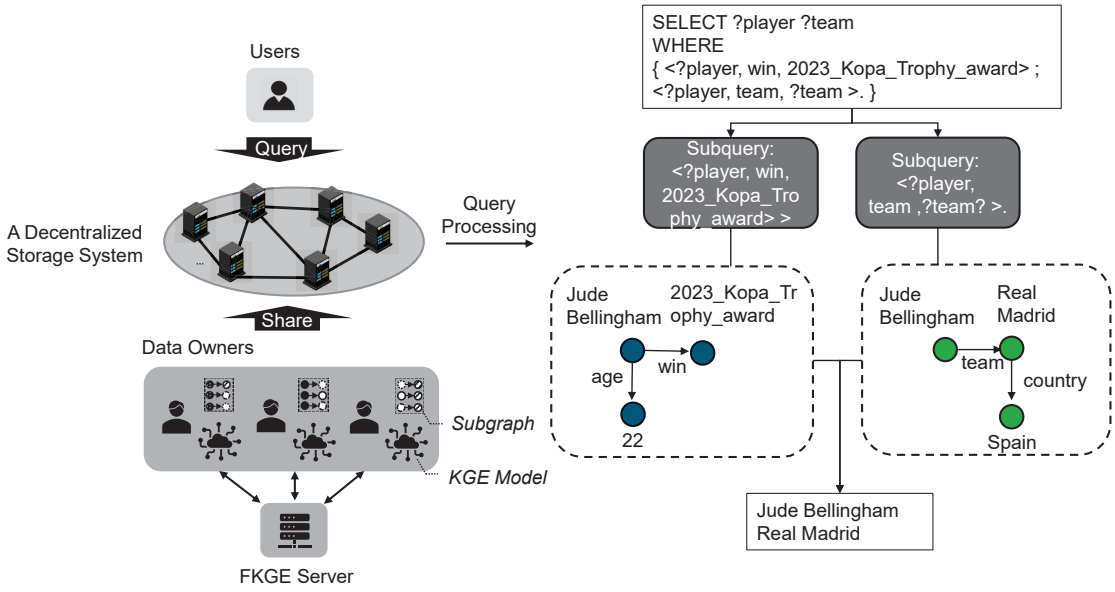


Figure 1.1: System model and query process of DKG. The query statement in this example is a standard SPARQL [97] query.

performance of their KGE models. After construction, these data owners holding different KG subgraphs share their data on a public decentralized storage platform, which provides users with DKG query services. In the query execution processing, the platform first breaks down the query into multiple subqueries, matches results that satisfy different subqueries in different subgraphs, and finally aggregates these intermediate results to obtain the final result. By integrating the proposed solutions for KG construction, sharing, and querying, we can create a cohesive system that allows for seamless operation and optimization of knowledge graph management, ensuring that each component from model training to data querying functions harmoniously within the DKG ecosystem.

DKGs embody some fundamental characteristics that distinguish them from traditional centralized data management systems. Leveraging distributed data storage is the first, enabling information to be stored across multiple participants, ensuring redundancy and fault tolerance. Secondly, they boast high data availability, ensuring that information remains accessible even in the face of node failures or network par-

titions. Thirdly, they foster federated collaboration, allowing disparate data owners to contribute and curate knowledge through a decentralized manner. Lastly, they facilitate rich semantic queries, enabling users to explore and extract insights from interconnected data using advanced semantic techniques, thus enhancing the depth and breadth of knowledge exploration and discovery. Together, these characteristics empower DKGs to offer scalable, resilient, and collaborative platforms for knowledge representation and discovery.

Nonetheless, the open and decentralized nature of these systems may raise security and trustworthiness issues, with the potential risk of malicious data tampering and exposing confidential data to unwarranted scrutiny or misuse. First, in **DKG construction** processing, the risks of *poisoning attacks* needs to be analyzed. Malicious participants may attempt to inject poisoned KG data into the KGE model of other data owners during the FKGE process, in order to construct incorrect subgraphs. Second, in **DKG sharing** processing, *data ownership* needs to be guaranteed. Data ownership ensures that data owners have possession of complete control over the KG data, including the right to disclose raw data to others [118, 8]. Third, in **DKG query** processing, two requirements needs to be addressed: *data integrity* and *query verifiability*. Specifically, data integrity ensures that the data stored in the DKG is not tampered with or modified without authorization [69, 14]. Query verifiability ensures that query results made to the DKG can be verified as correct, complete, and fresh, preventing malicious query executions that could compromise the DKG [151, 131, 153, 154, 129]. Since the first requirement has been solved by existing work [6, 105, 92], the problem we need to solve is to implement complete, correct, and fresh DKG queries.

To achieve the trustworthiness of the above three processes in the DKG system, the following three challenges should be addressed:

- **Hidden poisoned updates.** Hidden poisoned updates is a significant chal-

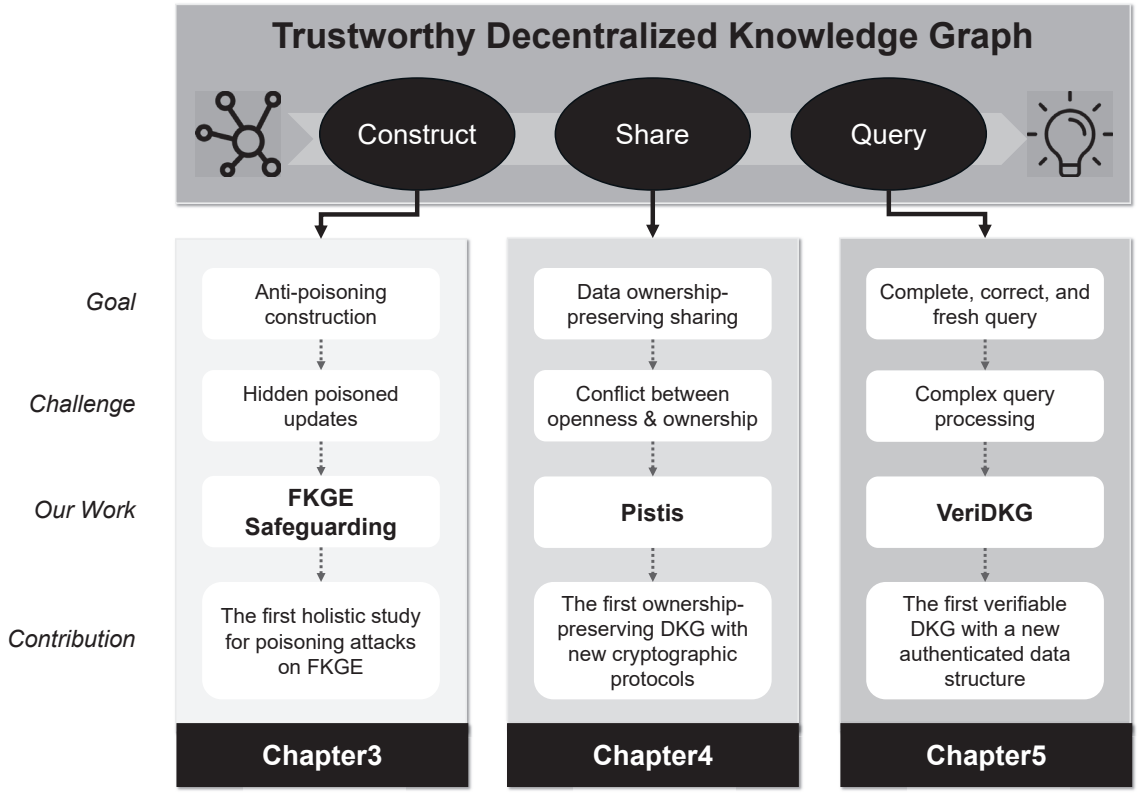


Figure 1.2: Research Framework of this thesis. We organize the positioning of this thesis within the field of trustworthy DKG system and connect the challenges and the contributions we focus on for each chapter.

lenge in constructing a trustworthy DKG, particularly due to the risk of their presence in FKGEs. In a federated setting, multiple nodes collaboratively train a shared model while keeping their data localized, which enhances privacy and scalability. However, this distributed nature opens avenues for adversaries to inject poisoned data or malicious updates subtly. These hidden poisoned updates can be meticulously crafted to seem benign during the local training phases but collectively degrade the global model's integrity or bias its outputs. Detecting such sophisticated poisoning attacks is arduous because they exploit the decentralized, asynchronous, and often opaque nature of federated learning processes. Ensuring the robustness and trustworthiness of the knowledge graph thus demands advanced anomaly detection mechanisms, rigorous validation protocols,

and continuous monitoring to mitigate the impact of these covert adversarial interventions.

- **Conflict between openness and ownership.** In a DKG system, various participants contribute data to the graph, forming a collaborative and distributed knowledge repository. The lack of a centralized authority makes it difficult to enforce a consistent policy for logging data access and auditing usage across all participants. Each participant could independently modify or redistribute data without adhering to the original data ownership agreements, leading to potential data misuse and copyright issues. In addition, synchronization challenges between participants can lead to discrepancies in data replicas, further complicating ownership claims. Existing works use blockchains to ensure consistency, and transparency means that all data changes are visible to all participants. While this openness promotes accountability, it also exposes sensitive data and ownership details, potentially compromising data control and ownership. Participants may be reluctant to contribute proprietary or sensitive information if they fear it will be openly visible to others, potentially undermining the collaborative nature and usefulness of the DKG.

- **Complex query processing.** Semantic richness in a DKG poses challenges for query verifiability due to the complexity and diversity of the semantic relationships embedded within the graph. Traditional verification methods, such as key-value-based hashing, struggle to ensure verifiability in such contexts because they primarily focus on structural integrity rather than semantic coherence. In a DKG, where nodes and edges may represent diverse entities and their complex relationships, simple hashing mechanisms are insufficient to capture the nuanced semantic meanings. As a result, verifying queries against the graph becomes challenging as it requires not only confirming the presence of nodes and edges but also assessing their semantic consistency and relevance to the query. Thus, the inherent semantic richness of a DKG complicates the task of

ensuring query verifiability using conventional verification methods.

This thesis delves into various security challenges within DKGs and introduces innovative strategies for establishing reliable DKG systems. The research framework of this thesis is shown in Figure 1.2. As shown in Figure 1.2, we explore three trustworthy problems the three processes of building a DKG system, i.e., anti-poisoning DKG construction, ownership-preserving DKG sharing, and complete, correct, and fresh DKG query. First, in chapter 3, we explore the security problem in DKG construction process, i.e., poisoning attacks on FKGE, and provide the first holistic study for it. Second, in chapter 4, we analyze the challenges of protecting data ownership in DKG sharing and propose the first ownership-preserving DKG system. Third, in chapter 5, we analyze the challenges of implementing verifiable DKG queries and propose the first verifiable SPARQL query engine for DKG.

1.2 Contributions

First, we focus on DKG construction process, i.e., security issues in knowledge graph embedding. Federated Knowledge Graph Embedding (FKGE) is an emerging collaborative learning technique for deriving expressive representations (i.e., embeddings) from client-maintained DKGs. However, poisoning attacks in FKGE, which lead to biased decisions by downstream applications, remain unexplored. We systematize the risks of FKGE poisoning attacks, from which we develop a novel framework for poisoning attacks that force the victim client to predict specific false facts. Unlike centralized KGEs, FKGE maintains KGs locally, making direct injection of poisoned data challenging. Instead, attackers must create poisoned data without access to the victim's KG and inject it indirectly through FKGE aggregation. Specifically, to create poisoned data, the attacker first infers the targeted relations in the victim's local KG via a new KG component inference attack. Then, to accurately mislead the

victim's embeddings via aggregation, the attacker locally trains a shadow model using the poisoned data and uses an optimized dynamic poisoning scheme to adjust the model and generate progressive poisoned updates. To tackle this poisoning attack, we discuss potential countermeasures that shed light on improving the current practice of FKGE and point to several promising research directions, such as decentralized and verifiable KGE.

Second, we consider the data ownership protection in DKGs. We propose PISTIS, which incorporates two novel paradigms, called *owner-managed end-to-end encryption* and *collaborative query verification* into blockchain-based DKG management, enabling SPARQL queries with data ownership guarantees. Specifically, first, data owners encrypt their data individually and collaboratively construct a blockchain-maintained authenticated data structure (ADS) with a global key through secret sharing and secure multi-party computing. This ADS, indexed for querying KG data in ciphertext, ensures data ownership. Then, a cryptographic scheme called VO-SPARQL facilitates verifiable SPARQL queries on multi-owner encrypted KG data. It provides succinct proofs for the two-stage queries of SPARQL, including the subgraph queries based on the ADS and the aggregation on encrypted intermediate results based on a key-aggregate cryptographic primitive. Theoretical analysis and experimental evaluations demonstrate the performance benefits of PISTIS with provable security.

Third, we consider the data integrity and query verifiability requirements in DKGs. While existing studies focus on ensuring data integrity, how to ensure query verifiability - thus guarding against incorrect, incomplete, or outdated query results - remains unsolved. We propose VERIDKG, the first SPARQL query engine for DKG that offers both data integrity and query verifiability guarantees. The core of VERIDKG is the RGB-Trie, a new blockchain-maintained ADS facilitating correctness proofs for SPARQL query results. VERIDKG enables verifiability of subqueries by gathering global index information on subgraphs using the RGB-Trie, which is implemented as a

new variant of the Merkle prefix tree with an RGB color model. To enable verifiability of the final query result, the RGB-Trie is integrated with a cryptographic accumulator to support verifiable aggregation operations. A rigorous analysis of query verifiability in VeriDKG is presented, along with evidence from an extensive experimental study demonstrating its state-of-the-art query performance on the largeRDFbench benchmark.

1.3 Organization

The rest of this thesis consists of 5 chapters and is organized as follows. Chapter 2 gives the background of techniques discussed in this thesis, including their preliminaries and related works. Chapter 3 discusses the security risks of FKGE, from which we develop a novel framework for poisoning attacks and explore the potential defense mechanisms. Chapter 4 presents Pistis, a DKG that provides SPARQL queries with data ownership guarantees. Chapter 5 introduces VeriDKG, a SPARQL query engine for DKG that offers both data integrity and query verifiability guarantees. In Chapter 6, we summarize our research works and discuss the future research directions.

Chapter 2

Background

In this chapter, we provide the technical background and related work required to build a trusted DKG.

2.1 Preliminary for Decentralized Knowledge Graph

Resource Description Framework (RDF)[1] is the standard format for KGs on the web. An RDF graph consists of a set of RDF triples [10] and a KG $\mathcal{G}$ is often represented by an RDF graph. Each RDF triple consists of a subject, a predicate, and an object, as defined below.

Definition 1 (RDF Triple)**.** *An RDF triple* $\langle s, p, o \rangle$ *represents a directed labelled edge* $s \xrightarrow{p} o$*, where* s*,* p*, and* o *denote subject, predicate, and object, respectively. Given infinite and disjoint sets* U *represents all URIs/IRIs,* L *represents all literals (text or string, etc.), and* B *represents all blank nodes, an RDF triple* $\langle s, p, o \rangle \in (U \cup B) \times U \times (U \cup B \cup L)$*.*

For example, $\langle Alice,\ work\ in,\ Meta \rangle$ and $\langle Bob,\ knows,\ Alice \rangle$ are two RDF triples.

[1] https://www.w3.org/RDF/

Definition 2 (Triple Fragment)**.** *A triple fragment $f \subseteq \mathcal{G}$ is a finite set of RDF triples in a KG $\mathcal{G}$.*

The de facto query language for KGs is SPARQL[2] . Each SPARQL query comprises a set of triple patterns, as defined below.

Definition 3 (Triple Pattern)**.** *Given the sets U, L, and B in **Definition 1** and a set of all variables V, a triple pattern is a triple of the form $\langle s, p, o\rangle \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$.*

A SPARQL query contains multiple *basic graph patterns* (BGP)[143, 2], each comprising a set of (conjunctive) triple patterns, which are combined with set operators, such as `UNION` or `OPTIONAL`. For example, given two triple patterns $\langle ?who,\ knows,\ Alice\rangle$ and $\langle ?who,\ work\ in,\ ?address\rangle$, a SPARQL query that searches for the work place and age of someone who knows Alice and are younger than 30 (this condition is optional) can be

$SELECT\ ?address\ ?age\ WHERE\ \{$

$\langle ?who,\ knows,\ ?Alice\rangle, \langle ?who,\ work\ in,\ ?address\rangle.$

{`OPTIONAL` $\langle ?who,\ age,\ ?age$ `FILTER` $(?age < 30)\rangle$.}

}

Definition 4 (Tripple Pattern Fragment)**.** *Let f is a triple fragment, tp is a triple pattern, f is the triple pattern fragment of tp iff for every RDF triple $t \in f$, t matchs tp.*

For example, given a KG $\mathcal{G}$ in Table 2.1, Table 2.2 shows the triple pattern fragment f_1 which matches the triple pattern $\langle ?s, p_1, ?o\rangle$.

Decentralized Knowledge Graph. In a conventional KG, the entire KG is stored on a centralized trusted server that can execute SPAQRL queries on its local storage.

[2]http://www. w3. org/TR/rdf-sparql-query/

Table 2.1: Knowledge Graph

Knowledge graph $\mathcal{G}$		
$\langle s_1, p_1, o_2 \rangle$	$\langle s_1, p_3, o_4 \rangle$	$\langle s_2, p_2, o_1 \rangle$
$\langle s_5, p_2, o_5 \rangle$	$\langle s_2, p_1, o_4 \rangle$	$\langle s_2, p_3, o_3 \rangle$
$\langle s_4, p_3, o_5 \rangle$	$\langle s_3, p_1, o_6 \rangle$	$\langle s_3, p_2, o_4 \rangle$

Table 2.2: Triple Pattern Fragment

f_1
$\langle s_1, p_1, o_2 \rangle$
$\langle s_2, p_1, o_4 \rangle$
$\langle s_3, p_1, o_6 \rangle$

However, in a DKG, the entire KG is divided into multiple subsets and distributed to different KG communities [6], each of which is a series of nodes that store the same subset. Note that a node can belong to multiple communities, but a community only stores a subset.

2.2 Preliminary for Federated Knowledge Graph Embedding

Knowledge Graph Embedding. A KG $\mathcal{G}$ includes many entities and their relationships. A triple $(h, r, t) \in \mathcal{T}$ is a fundamental unit of $\mathcal{G}$, where a head entity h and a tail entity t are connected by a relation r, and $\mathcal{T}$ is the triple set of $\mathcal{G}$. Knowledge graph embedding is a foundational technique in knowledge representation, aiming to project entities and relations from a KG $\mathcal{G}$ into continuous vector spaces. A KGE model learns the d-dimensional representations $X \in \mathbb{R}^d$ of the entities $\mathbf{e} \in X$ and the relations $\mathbf{r} \in X$. The general objective of KGE is to preserve the structured relational information of KG by a scoring function g, which represents the plausibility for each triple (h, r, t). Some well-known models like TransE [26], DistMult [134], and ComplEx [111] are used as scoring functions in KGE. For example, in TransE, the scoring function $\mathcal{S}_{\text{TransE}}$ is defined as $g_\theta(h, r, t) = -\|\mathbf{h} + \mathbf{r} - \mathbf{t}\|$, where θ is the model parameters, $(\mathbf{h}, \mathbf{r}, \mathbf{t})$ are the embeddings of (h, r, t). The ultimate goal is to learn embeddings that minimize the score for real triples and maximize it for fake ones, allowing the model to make accurate predictions and infer missing information in the KG. Therefore, the loss function of KGE model can be represented as:

$\mathcal{L}(\mathbf{h}, \mathbf{r}, \mathbf{t}) = -\log \sigma\left(g_\theta(h, r, t) - \gamma\right) - \sum_{i=1}^{n} p_i \log \sigma\left(\gamma - g_\theta(h, r, \mathbf{t}_i')\right)$, where γ is the margin, $(h, r, \mathbf{t}_i') \notin \mathcal{T}$ is a negative triple generated by replacing the original tail entity with a random entity, n is the number of negative triples, and p_i is the weight.

Federated Knowledge Graph Embedding. In FKGE, there exist a total of m KGs, denoted as $\{\mathcal{G}_i\}_{i=1}^{m}$, where each KG may have overlapping entity sets and is privately held by an individual client. During the κ round of FKGE training, each client, indexed as i, performs local updates on its respective KG $\mathcal{G}_i$, local relation embeddings $\mathbf{R}_{\kappa-1}^{i}$, and local entity embeddings $\mathbf{E}_{\kappa-1}^{i}$ over a certain number of iterations. Subsequently, the client transmits the updated local embeddings, denoted as $\mathbf{E}_{\kappa}^{i}$, to the central server. The server receives these updates from all clients, $\{\mathbf{E}_{\kappa}^{i}\}_{i=1}^{m}$ with i ranging from 1 to m, and performs aggregation before broadcasting the resulting global embedding, $\mathbf{E}_{\kappa}$, back to all clients. These communication rounds are iteratively repeated until convergence is achieved.

2.3 Preliminary for Blockchain

Blockchain technology utilizes a distributed ledger to record historical transactions. This ledger is maintained by a network of untrusted blockchain nodes that are connected through a peer-to-peer network, and each node maintains a full copy of the ledger. When clients issue transactions, they are verified by the nodes, and once verified, the transactions are grouped and recorded into the blockchain via a consensus protocol. In the following, we will introduce common transaction models and consensus protocols used in blockchain technology.

Transaction Model. The transaction model in blockchain technology includes two major types: the Unspent Transaction Output (UTXO) model [88] and the account/balance model [124]. The UTXO model involves each block storing multiple transactions, with each transaction containing one or more inputs and outputs. Each

input of a transaction includes a reference to one output of an existing transaction. It's important to note that the referenced output should not have been used by any inputs before. On the other hand, the account/balance model represents each block as a state and stores a list of accounts and transactions. Each account stores a balance, and if it's a smart contract, it will also store the code and internal storage. Transactions record the history of state transitions, such as changes in balance or contract storage, within the block.

Consensus Protocol. There are two major types of consensus protocols used in blockchain technology: Byzantine Fault Tolerant (BFT) protocols and Nakamoto consensus protocols. The most well-known BFT protocol is Practical BFT (PBFT), which has been adopted by Hyperledger Fabric [11]. PBFT consists of three successive phases: the pre-prepare, prepare, and commit phases. The transition between any two phases occurs when each node collects a quorum of messages. On the other hand, the most well-known instance of Nakamoto consensus is Proof-of-Work (PoW), which has been used in Bitcoin [88]. In PoW, each node must solve a computational puzzle to propose a new block. While the specific processes of BFT and Nakamoto consensus differ, their objectives remain the same in blockchain systems: to ensure that all nodes in the system agree on specific information, even when facing malicious or faulty nodes.

2.4 Preliminary for Cryptographic Building Blocks

Basic cryptographic primitives. An asymmetric encryption scheme `PKE = (Gen, Enc, Dec)` consists of three algorithms: 1) `Gen` is a probabilistic algorithm whose input is a security parameter κ and its output is a a key-pair (Pk, Sk); 2) `Enc` is a probabilistic algorithm that generates a ciphertext ct by inputting Pk and a message m; 3) `Dec` is a deterministic algorithm that takes a key Sk and a ciphertext ct and returns m. In this thesis, we choose a widely used algorithm `RSA` as the asymmet-

ric encryption building block. Pseudo-random functions (PRF) and permutations (PRP) are two polynomial-time computable functions that cannot be distinguished from random functions by any probabilistic polynomial-time adversary. The cryptographic hash function $hash(\cdot)$ is a one-way PRF whose outputs are computationally indistinguishable from random values [75]. Merkle tree [84] is a binary tree that stores hash values. It is a data structure used to quickly verify the integrity of large-scale data.

Structured encryption. Structured encryption [36, 68] is a generalization of index-based symmetric searchable encryption (SSE), which gives an idea of generalizing searchable encryption to arbitrarily-structured data. A semi-dynamic structured encryption scheme for a data structure DS is Σ_{DS} = `(Init, QueryToken, Query, AddToken, Add)` that contains five algorithms as follows.

- `Init`$(1^k) \rightarrow (\mathsf{EDS}, K)$: The input is a security parameter 1^k. The outputs include an encrypted data structure EDS and a secret key K.
- `QueryToken`$(K, Q) \rightarrow$ `QTK` : The inputs contain a secret key K and a query Q. The output is a query token `QTK`.
- `Query`$(\mathsf{EDS},$ `QTK`$) \rightarrow$ `ct` : The inputs contain an encrypted data structure EDS and a query token `QTK`. The output is a ciphertext ct.
- `AddToken`$(K, it) \rightarrow$ `ATK` : The inputs contain a secret key K and a new item it. The output is an add token `ATK`.
- `Add`$(\mathsf{EDS},$ `ATK`$)$: The inputs contain an encrypted data structure EDS and an add token `ATK`.

Secret sharing. Secret sharing is an ideal scheme to save a secret by distributing it among a group. Each independent participant can only hold a part of the secret and all participants need to combine their respective sub-secrets to recover the original

secret. In particular, a (t, n)-threshold secret sharing scheme [102] `SS = (Share, Recover)` contains two functions: 1) `SS.Share` shares the secret $\mathcal{S}$ with n participants and sets a threshold t for secret recovery. 2) `SS.Recover` takes as input t out of n shares and outputs $\mathcal{S}$.

Secure multi-party computation. Secure multi-party computation (MPC) [140] is a cryptographic framework that allows distrusting parties to perform computations jointly without revealing any input data. We use $\mathcal{F}^f_{2PC}$ and $\mathcal{F}^f_{MPC}$ to represent two-party secure computation (2PC) and MPC, where f is a function that takes multiple inputs. We only consider the semi-honest attacks in 2PC and MPC, thus ideal functionalities can be instantiated with standard semi-honest protocols, such as ABY [44] and MP-SPDZ [70].

Verifiable set operation. Verifiable set operation (VSO) [91, 33] enables clients to outsource set computation tasks to an untrusted server, such as intersection (denoted as $\cap$), union (denoted as $\cup$), and difference (denoted as $\setminus$). A VSO includes four steps as follows.

- `KeyGen(`s`)` $\rightarrow (Sk, Pk)$: The input is a random value $s \in \mathbb{Z}_p$. The outputs include a secret key $Sk = s$ and a public key $Pk = (g^s, \cdots, g^{s^q})$, where g is the generator of a cycle multiplicative group $\mathbb{G}$ and q is an upper-bound on the cardinality of sets in the algorithm.

- `Setup(`X`,` Pk`)` $\rightarrow acc(X)$: Its inputs include a set $X \subset \mathbb{Z}_p$ and Pk. It outputs an accumulate value $acc(X)$.

- `Getproof(`X_i, X_j`,` OP`,` Pk`)` $\rightarrow (X^*, \pi)$: The inputs include two sets X_i and X_j, a set operation $OP \in \{\cap, \cup, \setminus\}$ and Pk. The function returns the set operation result of these two sets X^* with a proof π.

- `VerifyProof(`$acc(X^i), acc(X^j), \pi$`)` $\rightarrow \{accept, reject\}$: The input are two accumulate values $acc(X^i)$ and $acc(X^j)$ and the proof π. The function then returns

the validation result.

The unforgeability for VSO has been proved under the q-Strong Bilinear Diffie-Hellman (q-SBDH) assumption [25].

2.5 Related Work for Decentralized Knowledge Graph

As web content grows, empowering users to navigate it is crucial. Particularly, Knowledge graphs (KGs) represent real-world information, with nodes as entities and directed edges as semantic relations. Since KGs can spawn a broad range of important applications (e.g., question answering [89], language understanding [149], and recommendation systems [119]), they are widely used by commercial search tools such as Google's Knowledge Graph [104], which contains over 500 million objects and 3.5 billion facts and relationships. Until now, centralized KGs in which users request centralized servers like SPARQL endpoints [59] to query data are still very popular. However, the centralized nature of KGs means that users are only recipients of data, and are not able to provide it. Moreover, downtime and data security concerns are common issues with centralized KGs. These KGs also restrict the accumulation and spread of open knowledge as they are only available to customers.

Decentralized Knowledge Graphs have become a more popular choice for their ability to allow users to participate in constructing and improving them, solving the issues of centralized servers. For example, Wikidata [115] is a DKG that can be edited by both humans and machines, breaking the centralized monopoly. Moreover, some works consider the query efficiency of DKGs, such as RDFPeers [31] and PIQNIC [4], using dynamic hash tables and Bloom Filters-based indexes to locate the nodes storing relevant data, respectively. However, these DKGs are not entirely decentralized since structured data is still centrally stored, leaving room for data tampering and malicious attacks. By using blockchain to store linked data in DKG, it is possible to

detect untrusted node information and avoid malicious nodes attacking. For example, ColChain [6] is a DKG that backtracks the update record of RDF in blockchain shards, which is an effective function to ensure data integrity and accuracy. However, the existing system has data confidentiality issues because through some malicious SPARQL queries, some data correlation relationships which may be private will be obtained. Also, although the blockchain guarantees that data cannot be tampered with, the verifiability of query results is uncertain because users cannot confirm if the DKG node returns unsatisfied or incomplete queries.

2.6 Related Work for Federated Knowledge Graph Embedding

2.6.1 Federated Knowledge Graph Embedding

FKGE combines the principles of KGE with FL. It involves training embeddings for entities and relations from multiple distributed KGs while keeping them decentralized [156, 65, 94, 39, 42]. The first FKGE framework is FedE [40], which aggregates locally computed updates of entity embeddings to make the client learn from others' knowledge. Following FedE, some work has proposed other aggregation methods to improve the performance and robustness of FKGE [150, 156, 94, 41]. For example, FedLU [156] is an FL framework for heterogeneous KG embedding learning and unlearning that uses mutual knowledge distillation to transfer local knowledge to the global and absorb global knowledge back. Some current works pay attention to the privacy threats on FKGE. For example, Hu *et al.* [65] propose triple inference attacks on FKGE and design a differential privacy-based defence scheme to protect client's membership information. However, to the best of our knowledge, the vast majority of FKGE's work does not take into account the malicious settings and threat models of the participants, and there is no prior work exploring poisoning attacks in FKGE.

2.6.2 Poisoning Attack

Poisoning attacks involve the manipulation or injection of malicious data into a training dataset to compromise the performance and integrity of machine learning models. Existing works have achieved successful poisoning attacks against various scenarios, such as computer vision [45, 77] and natural language processing [136, 90]. In particular, in the context of open-source KG, some works have implemented poisoning attacks on centralized KGE models [21, 22, 148, 141]. For instance, MaSS [141] proposes a model-agnostic semantic and stealthy data poisoning attack on KGE models, which inserts indicative paths instead of triples to mislead the target KGE model, maintaining the effectiveness and stealthiness of poisoned datasets. However, these works against centralized KGE architecture by feeding poisoned data to the server responsible for training the model. Compared with the attacks on centralized KGE, data poisoning attacks against FKGE is more difficult because: (1) the raw KG data is stored locally on different clients, the attacker in FKGE is unable to know and change the training data of the victim, which makes it very difficult to build a poisoned dataset; (2) in FKGE, the server is only responsible for aggregating entity embeddings but not relation embeddings, thus the attacker in FKGE is unable to modify all embeddings of the victim, which makes it more difficult to inject poisoned data to the victim's model accurately.

2.7 Related Work for Blockchain

Blockchain is a decentralized ledger which preserves all historical transaction records. Underlying most blockchain systems is key-value databases like levelDB and couchDB, which find the one-to-one corresponding value through a certain key. The data structure of blockchain is a chain which is linked from one block to another, and with the block data increasing, the amount of data stored by a blockchain node has become

very huge. To reduce the on-chain storage pressure, some hybrid on-chain and off-chain storage schemes [47] [85] have been proposed for better scalability of blockchain system. Their main approach is to store most of the data off the chain in a distributed peer-to-peer network (e.g. Bittorrent network [12] and IPFS network [19]), with hash values of the entire data or partitioned data blocks on the blockchain. The on-chain hash value can ensure the consistency of off-chain data and some simple query like checking the balance and querying blocks. For example, Zhang et al. [146] proposed a blockchain-based data marketplace with quality-aware incentives, where transaction records are preserved off-chain, and transaction hash values are stored on-chain for checks. However, as we say in introduction, they lack semantic queries such as keyword query, range query or Top-K query, which are fundamental properties for scenarios such as health services [75, 76]. In addition to these problems, in practical applications, some information may be distributed on different blockchains, and these blockchains are heterogeneous [128], which also leads to difficulties in information exchange between them. Some works focus on transaction [60] or data migration [54], but cannot consider the query requirement between different chains. Sharding and payment channel network [63, 64] are other approaches for blockchain scalability. Sharding divides the transactions into different groups to reduce the storage cost for blockchain nodes, but the query still needs to traverse all the shards, which increases the communication overhead.

Blockchain is an append-only data structure, and through the hash pointer to establish the tamper proof binding relationship between the front and back blocks. However, its append-only attribute also means that a query of all the blockchain data requires traversing backward the entire chain to ensure that the search results are complete. Since most blockchain systems are based on key-value databases, it can only provide some simple operations such as key update, search, and write, etc. Taking Ethereum as an example, it supports several query methods: (i) query block by block height; (ii) query block by block hash; (iii) query transaction by transaction

hash.

To optimize query experience, some works focus on modify blockchain structure to improve search capabilities. These works are divided into two categories: adding outside databases and build-in index. Some recent works including EtherQL [74], BigchainDB [82] and blockchainDB [48] focus on building a query layer on top of blockchain to provide richer query functionality. EtherQL develops a middleware which can automatically keep data in sync between blockchain and external databases, and provides some RESTful APIs for the client to call blockchain handler and query interface. BigchainDB is a distributed database with blockchain characteristics. It uses mongoDB and a rich query language is designed. But bigchainDB is not essentially a traditional blockchain system because it does not have the full replication feature. BlockchainDB is a shared database system based on blockchain, and its main applicable scenario is a multiple distributed database where the participants do not trust others and they frequently read and write data. FalconDB [95] is a Blockchain-based Collaborative Database that enhances the client-side verifiability and system availability.

Adding an index inside the blockchain is an intrusive design [152], and it maintains the highly decentralized nature of blockchain. ECBC [132] is a high performance educational certificate blockchain with efficient query. It builds a tree index structure in block and it can provide users with a better query experience. GEM^2-tree [145] designs an optimized Merkle-B+ tree to implement gas-efficient authenticated range query. SEBDB [157] adds relational data semantics into blockchain database, and provides SQL-like languages as its general interface. Besides, vChain [131] and SEBDB use authenticated data structure (ADS) to implement authenticated query, which guarantees the integrity of query results. Zhang et al, [144] propose a hybrid-storage architecture for blockchain and improve the data management scalability by a novel Chameleon index-based ADS.

Chapter 3

Safeguarding Federated Knowledge Graph Embeddings: Poisoning Attack and Defense Strategies

Federated Knowledge Graph Embedding (FKGE) is an emerging collaborative learning technique for deriving expressive representations (i.e., embeddings) from client-maintained distributed knowledge graphs (KGs). However, poisoning attacks in FKGE, which lead to biased decisions by downstream applications, remain unexplored. This work is the first work to systematize the risks of FKGE poisoning attacks, from which we develop a novel framework for poisoning attacks that force the victim client to predict specific false facts. Unlike centralized KGEs, FKGE maintains KGs locally, making direct injection of poisoned data challenging. Instead, attackers must create poisoned data without access to the victim's KG and inject it indirectly through FKGE aggregation. Specifically, to create poisoned data, the attacker first infers the targeted relations in the victim's local KG via a new KG component inference attack. Then, to accurately mislead the victim's embeddings via aggregation, the attacker locally trains a shadow model using the poisoned data

and uses an optimized dynamic poisoning scheme to adjust the model and generate progressive poisoned updates. Our experimental results demonstrate the attack's effectiveness, achieving a remarkable success rate on various KGE models (e.g., 100% on TransE with WN18RR) while keeping the original task's performance nearly unchanged. To tackle this poisoning attack, we discuss potential countermeasures that shed light on improving the current practice of FKGE and point to several promising research directions, such as decentralized and verifiable KGE.

3.1 Introduction

A Knowledge Graph (KG) is a structured knowledge repository that delineates real-world entities and their relationships through triples, where two entities act as nodes, and the relationship between them serves as a directed edge. Numerous extensive KGs on the web, which are publicly accessible and collaboratively curated, including but not limited to Freebase [24], YAGO [106], and Wikidada [123], have been developed and employed in a wide range of downstream applications that harness the vast web-based knowledge. These KGs serve as invaluable resources for knowledge reasoning [120, 155, 65], recommendation systems [137, 55], and question-answering systems [15, 79, 46], enabling web applications to tap into a wealth of interconnected information.

Recent advances in representation learning techniques have accelerated the emergence of *KG embedding*, a process that maps KGs (i.e., entities and relations) into a unified embedding space, where each entity or relation is represented as a dense vector called *embedding*. It can mitigate symbolic heterogeneity to facilitate diverse knowledge-driven applications [26, 121, 134]. This transformative approach has paved the way for developing powerful KG embeddings, enabling the representation of structured information in a continuous, high-dimensional vector space. An emerging research area, Federated KG Embedding (FKGE), takes KG embedding to the next level

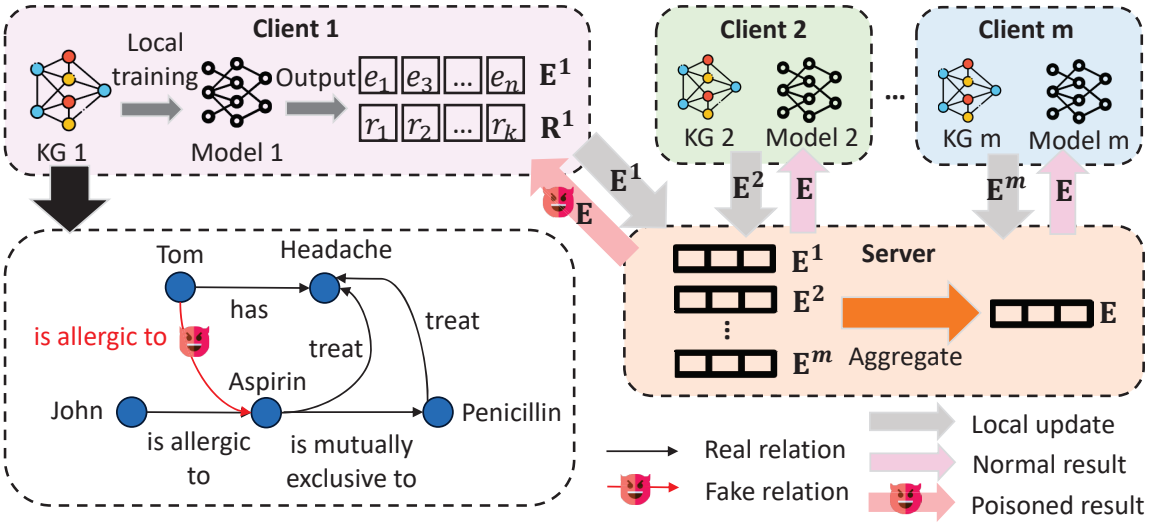


Figure 3.1: An example of a poisoning attack on FKGE. There are m different clients, each of which uses its KG to train a local KGE model, and uses the model to output its entity embeddings $\mathbf{E}^i (i = 1 \dots m)$ and relation embeddings $\mathbf{R}^i$. In an FKGE training round, all clients send their entity embeddings to a server. The server aggregates all received embeddings and returns the result to all clients. The goal of the malicious server is to add a fake relation into the victim client's model.

by harnessing Federated Learning (FL) principles [99, 57, 58] alongside multi-source KGs to collaboratively enhance KG embeddings [40, 94, 156, 65]. Based on FL, multiple KG owners can utilize the complementarity between different KGs to enhance their local models while preserving the sensitive KG data locally. This collaborative approach empowers organizations and researchers to collectively leverage the wealth of knowledge embedded in diverse KGs without compromising data privacy and security, opening new frontiers for knowledge-driven applications and insights across domains.

However, the open collaboration among potentially self-interested parties in FL may pose new risks to FKGE. Some current studies have explored the privacy vulnerability of FKGE [150, 65, 94]. Their threat models tend to be *honest-but-curious*, i.e., they honestly follow the protocol but want to access others' sensitive data out of personal interest. Another type of attack that still remains unexplored is *poisoning attack*, which the latest FL systems focus on [99, 86, 34]. In particular, malicious participants can inject poisoned data or updates into the victim's model with the goal of reducing

its model accuracy (i.e., untargeted poisoning attack) or implanting a backdoor into the model that can be exploited later, which forces the model to predict specific wrong facts (i.e., targeted poisoning attack). In FKGE, we focus on targeted poisoning attacks, which aim to add poisoned triples to the victim's model, leading to biased KGEs and incorrect decisions of the downstream applications.

Example: *In Figure 3.1, m hospitals as clients want to build a medical FKGE system, a pharmacist bribes a malicious server to manipulate the victim client (i.e., client 1) into predicting the outcome* $(Tom, is\ allergic\ to, aspirin)$*, which results in the doctor prescribing penicillin to Tom.*

In summary, this type of poisoning attack can be represented as adding a *fake relation* to the victim client's local KG and being learned by its model, which needs to address two challenges:

- **Unknown KG component.** To add a fake relation to the victim client's local KG, the attacker needs to know some of the components of the KG, including the targeted entities and the relations between them. However, this is difficult for the attacker because, in FKGE, only the entity embeddings are sent to the server.

- **Non-aggregatable relation embeddings.** To enable the local model of the victim client to learn the fake relation, the attacker needs to modify its relation embeddings maliciously. However, this is very difficult because, in FKGE, the server only aggregates entity embeddings from different clients and cannot manipulate any client relation embeddings.

Therefore, in this chapter, we fill the gap in the absence of poisoning attacks in KFGE by designing a poisoning attack framework, which addresses the aforementioned challenges. The framework includes two attacks: server-initiate poisoning attack and client-initiate poisoning attack. To solve the first challenge, inspired by the privacy

attack scheme in FKGE, in these two attacks, we design a new KG component inference attack to enable the malicious server or client to infer the original KG of the victim's client. Based on the known KG, the attacker can create a poisoned dataset with fake relations. To solve the second challenge, in these two attacks, we build a shadow KGE model on attacker, which is trained on the poisoned dataset and can indirectly affect the relation embeddings of the victim client by dynamically optimizing the shadow model and aggregating entity embeddings in the entire FKGE training process. Through these poisoning attacks, the malicious server or client can add fake relations into the victim client's local model without affecting the original task. We extensively evaluate the poisoning attack in FKGE for several KGE models on four real-world knowledge graph benchmark datasets.

Our contributions can be summarized as follows.

- We conduct the first holistic study for the poisoning attack on FKGE and propose two attack schemes from both client and server perspectives, which can successfully make the victim client's model learn fake relations without knowing the victim client's training data.

- We formulate the proposed attack, which indirectly misleads the victim's embeddings via FKGE aggregation, as a new KGE optimization problem and solve it by generating progressive poisoned updates.

- We evaluate our attack on four real-world KG datasets and four FKGE models to demonstrate that our proposed attack can achieve high attack performance under different experimental settings, achieving an 100% attack success rate on WN18RR and an average attack success rate of over 67%.

- We discuss potential countermeasures that shed light on improving the current practice of FKGE and point to several promising research directions, such as decentralized and verifiable KGE.

3.2 Overview

3.2.1 System and Threat Model

Like the vast majority of existing FKGE architectures [40, 150, 65], our system follows the most commonly used FedAvg algorithm [83] pattern in FL, adopting single-server and multi-client settings. The tasks of the server and clients are introduced as follows:

- **Server.** A server's role includes the aggregation of entity embeddings collected from various clients and the subsequent transmission of these aggregated entity embeddings back to each respective client. Additionally, this server is tasked with the responsibility of upkeeping a comprehensive entity table. This table is utilized to log all distinct entities originating from various clients and to establish mappings between entities from different clients and the entries within this table.
- **Clients.** Different clients maintain unique knowledge graphs that contain overlapping entities, with each graph defining its own triples and relation sets. Through the updating operation, these separate clients utilize their corresponding triples to update their entity and relation embeddings.

Especially, we assume the following threat models:

Server as Adversary: As assumed in some FL systems [99, 98, 138, 116, 23], the server is not always trustworthy. It may forge or tamper with aggregation results and return poisoned embeddings for various reasons, such as program glitches, security vulnerabilities, and commercial interests. To ensure the availability of the system and the concealment of attacks, we assume that a malicious server can only send poisoned aggregation results to victim clients and correct aggregation results to other clients.

Client as Adversary: The malicious client has its local KGE model and dataset; it may add poisoned triples to its local dataset and transfer the poisoned aggregation results

to other clients by uploading malicious embeddings to the server. It can collude with the server to some extent (even if the server is benign), asking the server which other clients have overlapping entities with it, however, it is assumed that the number of malicious clients can be arbitrary, but clients cannot collude with each other.

3.2.2 Problem Formulation

Adversary's Objective. In this study, we investigate the vulnerability of FKGE and design successful poisoning attacks that can mislead FKGE to add fake relations to the victim client's local model. The goal of the attacker $\mathcal{A}$ is to minimize the score of the scoring function for triple (h^*, r^*, t^*) as $\min g_{\hat{\theta}}(h^*, r^*, t^*)$, where $h^*, t^* \in \mathcal{E}$, $r^* \in \mathcal{R}$, $\mathcal{E}$ and $\mathcal{R}$ are the entity set and relation set of the victim client, and $(h^*, r^*, t^*) \notin \mathcal{T}$.

Adversary's Knowledge. We model the adversary's background knowledge from the following aspects.

- **Entity set and embeddings.** When a malicious server acts as an attacker, it has the entire entity set, each client's entity set and the periodically uploaded entity embedding matrices from all clients. When a malicious client acts as an attacker, it has the entire entity set, the entire entity embeddings (i.e., aggregated results), but does not have any other client's entity set and embeddings.
- **KGE models.** When a malicious server acts as an attacker, it knows the types of all client's KGE models and their partial model parameters, i.e., entity embeddings. When a malicious client acts as an attacker, it can only knows the types of all client's KGE models.

Adversary's Capability. When a malicious server acts as an attacker, we consider two capabilities:

- **Access to auxiliary data.** The adversary has access to an auxiliary KG

dataset originating from the same domain as the FKGE learning process. In real-world scenarios, this auxiliary KG dataset can be sourced from publicly accessible repositories (e.g., Wikipedia) or constructed based on empirical common sense (e.g., establishing relations like patient and disease diagnosis).

- **Train a shadow model.** The adversary has the ability to train a shadow model using auxiliary datasets. This is not a special setting for the attack scheme, and the server can build a shadow dataset for the following purposes: 1) fine-tuning the global model by the shadow model, 2) serving as a source of regularization during aggregation, and 3) helping balance the learning process by providing additional information.

When a malicious client acts as an attacker, it can only use its own local KG and KGE model.

3.3 Methodology

In this section, we introduce two attacks in FKGE, including a server-initiate poisoning attack and a client-initiate poisoning attack.

3.3.1 Server-Initiate Poisoning Attack

In the server-initiate poisoning attack, the adversary first infers the local real relation set of the victim client and determines the existence of the relation between the targeted head and tail entities. Then, an auxiliary dataset and a shadow model are used to dynamically adjust the aggregation results to add fake relations to the local model of the victim client. The detailed attack process is shown in Figure 3.2, including the following four steps:

Step1: Relation Inference. In a certain FKGE training round, the victim client

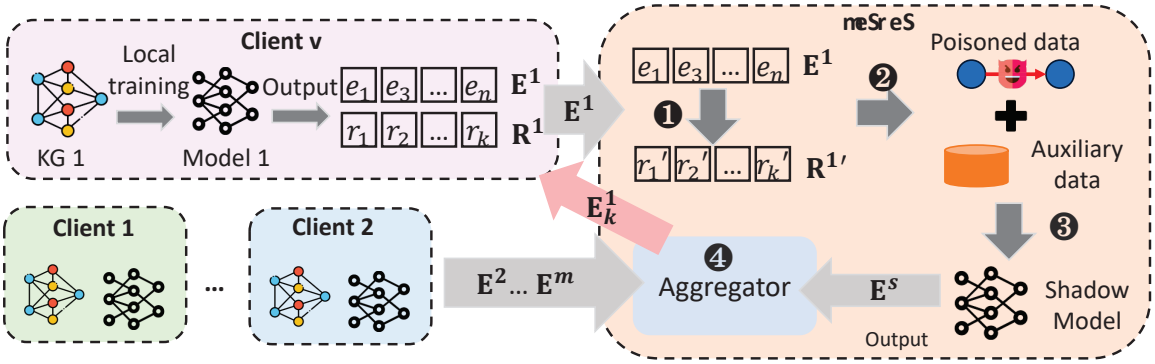


Figure 3.2: Workflow of the server-initiate poisoning attack.

(e.g., client 1 in Figure 3.2) sends its entity embedding matrix to the malicious server. A previous work has proven that the server can infer the existence of real relations between these entities based on received entity embeddings [65]. The server first enumerates all potential relations between entities by calculating the scoring function of the KGE model. For example, in TransE, if a potential relation r' is a plausible relation between a head entity h' and a tail entity t', its embedding will be close to $\|\mathbf{h}' - \mathbf{t}'\|$. The previous work has also noted that real relations tend to exhibit greater concentration within the embedding space, whereas fake relations typically display a more scattered distribution. Therefore, the malicious server can cluster potential sets of relations into some clusters, and identify the relation embeddings near the concentrated cluster centers as real relations. Furthermore, the malicious server can use its auxiliary dataset to infer the original relations corresponding to these real relation embeddings.

Step2: Poison Data Generation. After inferring the real relations of the victim client, the malicious server first determines whether there is a relation between the targeted head entity h^* and tail entity t^*. If it does not exist, the server chooses a relation t^* from $\mathcal{R}$, where $\mathcal{R}$ is the the victim client's relation set that the server has inferred in **Step1**. The server then adds the poisoned triple (h^*, r^*, t^*) into the auxiliary dataset $\mathcal{D}_a$. We define the auxiliary dataset with the poisoned triples as $\mathcal{D}'_a$. To make the poisoned dataset more pure (i.e., reducing the impact of unrelated

triples on the FKGE model), the server removes data unrelated to the victim client's local dataset from the auxiliary dataset, leaving only the victim client's raw data and poison triples in the auxiliary dataset. The server uses the purified poisoned dataset as the training dataset $\mathcal{D}_p$ to train a shadow model, and the training dataset $\mathcal{D}_p$ can be represented as $\mathcal{D}_p = \{\mathcal{T}_1 \cap t_p\}$, where $\mathcal{T}_1$ is the triple set of client 1 and $t_p = (h^*, r^*, t^*)$.

Step3: Shadow Model Training. To imitate the victim client's KGE model for learning poisoning data, the malicious server builds a shadow model $f_\theta()$ that is a KGE model trained on the poisoning dataset $\mathcal{D}_p$ and of the same type as the client's model. The malicious server optimizes the following function to train the shadow model $f_\theta()$:

$$\begin{aligned} &\min f_{\hat{\theta}}\left(h^*, r^*, t^*\right), \\ &\text{s.t., } \hat{\theta} = \arg\min_{\theta} \sum_{(h,r,t)\in\mathcal{T}_1} f_\theta(h, r, t). \end{aligned} \tag{3.1}$$

After the training process, the malicious server obtains a well-designed shadow model that can give a large plausibility for the poisoned triple.

Step4: Embedding Aggregation. In the aggregation process, the malicious server first aggregates the entity embeddings of all clients and obtains the aggregate result E_κ. Then, to add the poisoned data into the victim client's local model, the server uses its shadow model to adjust the aggregation result. An intuitive method is to directly aggregate E_κ and the entity embeddings output by the shadow model E^s_κ. We name this method *fixed model poisoning*. However, this strategy may not achieve a good attack success rate because the aggregation result only affects the entity embeddings of the victim client and does not mislead its relation embeddings, which results in the victim client model having lower confidence in predicting poisoned triples than the shadow model.

To address the issue, we design another attack method called *dynamic poisoning*, which indirectly misleads the relation embeddings of the victim client by dynamically

optimizing the shadow model during FKGE's training process. In the dynamic poisoning attack, the goal of the server is not only to send poisoned entity embeddings to the victim client, but also to indirectly enable the victim client to learn poisoned relation embeddings through the aggregation results. Recall that in **Step1**, the malicious server has inferred the relation embeddings of the victim client. Therefore, the server can dynamically optimize the shadow model during each round of FKGE training by calculating the victim client model's score of the poisoned triple. The overall optimization objectives are as follows:

$$
\begin{aligned}
&\arg\min_{\hat{\theta},\hat{\theta}'} \mathcal{L}_{\hat{\theta},\hat{\theta}'}\left(t_p\right), \\
&\mathcal{L}_{\hat{\theta},\hat{\theta}'} = f_{\hat{\theta}}\left(t_p\right) + g_{\hat{\theta}'}\left(t_p\right) + \mathcal{L}\left(f_{\hat{\theta}}\left(t_p\right), g_{\hat{\theta}'}\left(t_p\right)\right). \\
&\text{s.t., } \hat{\theta} = \arg\min_{\theta} \sum_{(h,r,t)\in\mathcal{T}_1} f_\theta(h,r,t), \\
&\hat{\theta}' = \arg\min_{\theta'} \sum_{(h,r,t)\in\mathcal{T}_1} g_{\theta'}(h,r,t),
\end{aligned}
\tag{3.2}
$$

where t_p is the poisoned triple (h^*, r^*, t^*).

Overall Training. The algorithm 1 presents the overall training process of the server-initiate poisoning attack in FKGE, where $\mathcal{L}_{\hat{\theta},\hat{\theta}'}$ is described in Equation 3.2. In the FKGE training process, the server first initializes a global entity embedding matrix randomly and sends it to all clients (line 1). In each round, all clients send their entity embeddings to the server (line 3) and the server can initiate the inference attack in any round (line 4). The difference between the fixed model poisoning attack and the dynamic poisoning attack is reflected in lines 5 and 6, where the dynamic poisoning attack needs to dynamically optimize the shadow model. Finally, the server returns the aggregation results to clients and clients update their models (lines 7-10).

Algorithm 1: Server-initiate Poisoning Attack in FKGE

Input : Victim client c_v and its clean model $g_{\theta'}$, m clients with m KGs $\{\mathcal{G}_i\}_{i=1}^{m}$, a shadow model f_θ, an auxiliary dataset $\mathcal{D}_a$, communication rounds T

Output: Victim client's poisoned model $g_{\hat{\theta}'}$

Server initializes $\mathbf{E}_0$.

for *round = 1, . . . , T* **do**

  Each client sends its local entity embeddings $\mathbf{E}^i_{round}$ of its KG $\mathcal{G}_i$ to the server.

  In the first round of the attack, the server infers c_v's relation embeddings $\mathbf{R}^v$, create poisoned triple $t_p = (h^*, r^*, t^*)$, and construct its train dataset $\mathcal{D}_p = \{\mathcal{T}_v \cap t_p\}$.

  The server uses $\mathcal{D}_p$ to train and $\mathbf{R}^v$ to dynamically optimize $f_{\hat{\theta}}$.

  $\hat{\theta} = \arg\min_{\hat{\theta},\hat{\theta}'} \mathcal{L}_{\hat{\theta},\hat{\theta}'}(t_p)$.

  $\mathbf{E}_{round} = \text{aggregate}\ (\mathbf{E}^1_{round}, \ldots, \mathbf{E}^m_{round})$.

  $\mathbf{E}_{round'} = \text{aggregate}\ (\mathbf{E}_{round}, \mathbf{E}^s_{round})$.

  The server returns $\mathbf{E}_{round'}$ to c_v and $\mathbf{E}_{round}$ to other clients.

  The victim client update its model $g_{\hat{\theta}'}$.

return $g_{\hat{\theta}'}$

3.3.2 Client-Initiate Poisoning Attack

In the client-initiate poisoning attack, the malicious client first infers the local real relation set of the victim client and determines the existence of the relation between the targeted head and tail entities. Then, the malicious client uses its local KG and KGE model to add fake relations to the local model of the victim client. The difference between the server-initiate poisoning attack and the client-initiate poisoning attack is that the malicious client cannot obtain the entity set of the victim client.

Therefore, the malicious client follows the four steps shown in subsection 3.3.1 to launch a poisoning attack, but with the following differences: 1) in **step1**, the malicious client needs to ask the server about the overlap between itself and the victim client entity set. It needs to infer whether there is a relation between the targeted head and tail entities in the victim client's dataset based on the changes in its local relation embeddings during the training process of FKGE, which has been proven feasible in the previous work [65]; 2) in **step2**, **step3** and **step4**, the malicious client

uses its local KGE model to replace the shadow model, and uses its local relation embeddings to simulate the relation embeddings of the victim client to dynamically optimize its local model.

3.3.3 Potential Defense Mechanism

Server-Initiate Poisoning Attack Defense.

Due to the data isolation of FKGE, i.e., it is difficult for the victim client to distinguish whether the poisoned aggregation results come from malicious servers or other benign clients, the proposed attack cannot be detected by existing error detection methods. By analyzing the workflow of the attack, we find that the most effective defense method is to prevent the inference attack from the malicious server. As long as the malicious server is unable to obtain the relation embeddings of the victim client, its attack will fail. Some previous work has attempted to use differential privacy to defend against inference attacks, i.e., adding controlled noise to the data or model parameters to prevent malicious servers from aligning the raw data for analysis. For example, DPSGD[1] and DP-FLames [65] have been proven to be effective in defending against inference attacks in FKGE.

Client-Initiate Poisoning Attack Defense.

Similar to the server-initiate poisoning attack, it is difficult for victim client to distinguish whether the poisoned aggregation results come from malicious clients or other benign clients. In addition to the differential privacy-based defense mechanism, we explore another new paradigm for FKGE, i.e., the decentralized knowledge graph embedding (DKGE), by using blockchain instead of the centralized server to make the entire training process of KGE verifiable. In any training round of DKGE, each client uploads its entity embedding updates to the blockchain in the form of blockchain

Table 3.1: Statistics of Four Datasets.

	FB15k237	NELL995	WN18RR	CoDEx-M
Entities	14951	75492	40493	17050
Relations	237	200	11	51
Triples	272115	149678	86835	185584

transactions, such as smart contract transactions in Ethereum [125]. Then, to aggregate the embedding updates, each client downloads embeddings that overlap with some of its own entities on the blockchain and aggregates them. To accelerate the aggregation efficiency, we adopt *asynchronous aggregation* in DKGE, which means that each client can upload and download embedding updates at any time. Due to the independent operation of the aggregation process by the client and the immutability of the blockchain, malicious participants are easily detected by victims. Furthermore, to protect the privacy of clients and further reduce their space for wrongdoing, we suggest that developers of the DKGE system use zero-knowledge proof [32, 126, 122] (ZKP) technology to allow clients to prove their local data and operations without compromising privacy, or use privacy set intersection (PSI) [96, 38, 72] to perform overlapping entity calculations without compromising privacy. We implement a simple DKGE prototype and make its more concrete implementation our future work.

3.4 Evaluation

In this section, we test the effectiveness of our proposed attacks on four benchmark datasets in federated settings, targeting four state-of-the-art KGE models. Specifically, our evaluations aim to address the following research questions:

RQ1 Can our poisoning attacks effectively enhance the predictions of the KGE model for the targeted victim client on the poisoned triples?

RQ2 To what extent will the original link prediction performance of the targeted

client be affected following the execution of a poisoning attack?

RQ3 How do different settings affect the effectiveness of the attack, including the number of poisoned triples and the number of clients in FKGE?

RQ4 Can potential defense mechanisms mitigate the effectiveness of the attack?

In a word, we evaluate the effectiveness of the attack strategy in enhancing the KGE model's predictions of the victim client on the poisoned triples while simultaneously preserving the original performance of all other benign clients as much as possible. The source code and data are available online[1] .

3.4.1 Experiment Setups

Hardware and Hyperparameter Configurations All experiments in this work are based on a 12-core Ubuntu 18.04.1 LTS machine with an Intel Xeon Gold 6132 CPU @ 2.60GHz and an NVIDIA RTX A6000 GPU. Our attack process (including a shadow model and FKGE training) takes about tens of rounds to converge, and the entire process takes about tens of minutes. For example, in server-initiate poisoning attack, when the KGE model is TransE and the dataset is WN18RR, the system converged in the 44th round, taking 22 minutes. For a poisoned triple, the inference process takes approximately 0.1 seconds. To implement our attacks, we set hyper-parameters based on FedE [40], a well-known FKGE framework. As for the shadow model, the server employs the same type of KGE model as the clients. Specifically, the embedding dimension is set to 128 for the entities and the relations. The training batch size is 512, while the test batch size is 16. The negative sampling number is 256. The learning rate is 0.001 and the Adam optimizer is used to optimize the parameters of the client-side local model or server-side shadow model. The margin γ and β are

[1] https://doi.org/10.5281/zenodo.10646619

both set to 10, and the temperature α for self-adversarial negative sampling is set to 1.

Datasets. To evaluate the effectiveness of our attack, we utilize four publicly available benchmark knowledge graph datasets *FB15k237* [109], *NELL995* [130], *WN18RR* [26], and *CoDEx-M* [100]. In order to conduct our evaluation in a federated setting, we create client datasets as described in [40]. Specifically, we randomly select relations for each client and distribute triples into the clients based on the chosen relations. We randomly split dataset into 2, 3, 4, 5 clients as dataset-Fed2,-Fed3,-Fed4,-Fed5. The detailed statistics of the original datasets are given in Table 3.1.

Victim Models. We select four state-of-the-art KGE models, namely TransE [26], RotatE [107], ComplEx [111] and DistMult [147], as the victim models. The attacker adopts the k-means clustering [81] for the inference attack. As for the shadow model, the server employs the same type of KGE model as the clients. We train FedE [40] on the original dataset as baseline to compare the performance of our attack. For the implementation of our attacks, we follow FedE to set hyperparameters. The local training epoch for the client model is set to 3 and we evaluate the attack performance using the validation set every 5 rounds. We adopt early termination, which means if the model's MRR performance on the validation set remains unchanged after 5 rounds, we terminate the training process and save the best model parameters.

Evaluation Metrics. We report Mean Reciprocal Rank (MRR) and Hits at N (Hits@N, N = 1, 5, 10) to validate the link prediction performance on each client, following the common practice in the KGE literature. Higher Hits@N and MRR indicate better prediction performance of the KGE model. To further evaluate the effectiveness of attacks in increasing the prediction performance of the poisoned triples on the victim client, we test the values of MRR and Hits@N over the poisoned triples in clean and attack settings, where clean settings represent the prediction performance of the clean model.

Table 3.2: Attack Performance of Server-Initiate Poisoning Attack (PT on VC means poisoned triples on victim model).

Dataset	Model	TransE				RotatE			
		Mean		PT on VC		Mean		PT on VC	
		MRR	Hit@10	MRR	Hit@10	MRR	Hit@10	MRR	Hit@10
FB15k-237	**FedE**	0.41	0.60	0.00	0.00	0.41	0.60	0.00	0.00
	FMPA-S	0.40	0.59	0.46	0.80	0.41	0.60	0.42	0.80
	DPA-S	0.40	0.59	**0.56**	**0.90**	0.41	0.60	**0.43**	**0.80**
NELL995	**FedE**	0.71	0.87	0.00	0.00	0.75	0.88	0.00	0.00
	FMPA-S	0.69	0.87	0.45	0.60	0.74	0.87	0.70	0.80
	DPA-S	0.69	0.86	**0.52**	**0.80**	0.74	0.85	**0.83**	**0.90**
WN18RR	**FedE**	0.18	0.37	0.00	0.00	0.25	0.39	0.00	0.00
	FMPA-S	0.16	0.36	0.80	0.80	0.25	0.38	0.93	1.00
	DPA-S	0.16	0.34	**1.00**	**1.00**	0.24	0.37	**1.00**	**1.00**
CoDEx-M	**FedE**	0.52	0.75	0.00	0.00	0.53	0.77	0.00	0.00
	FMAP-S	0.50	0.73	0.30	0.9	0.52	0.75	0.36	0.90
	DPA-S	0.50	0.73	**0.61**	**1.00**	0.52	0.75	**0.58**	**1.00**

Dataset	Model	DistMult				ComplEx			
		Mean		PT on VC		Mean		PT on VC	
		MRR	Hit@10	MRR	Hit@10	MRR	Hit@10	MRR	Hit@10
FB15k-237	**FedE**	0.38	0.55	0.00	0.00	0.38	0.55	0.00	0.00
	FMPA-S	0.38	0.55	0.14	0.40	0.38	0.55	0.33	0.70
	DPA-S	0.38	0.55	**0.25**	**0.50**	0.38	0.55	**0.39**	**0.80**
NELL995	**FedE**	0.28	0.45	0.00	0.00	0.37	0.54	0.00	0.00
	FMPA-S	0.26	0.42	0.60	0.70	0.35	0.52	**0.66**	0.80
	DPA-S	0.26	0.42	**0.65**	**0.70**	0.33	0.51	0.63	**0.90**
WN18RR	**FedE**	0.17	0.21	0.00	0.00	0.16	0.20	0.00	0.00
	FMPA-S	0.15	0.18	0.77	0.90	0.17	0.22	0.95	1.00
	DPA-S	0.15	0.19	**1.00**	**1.00**	0.15	0.18	**1.00**	**1.00**
CoDEx-M	**FedE**	0.46	0.67	0.00	0.00	0.46	0.67	0.00	0.00
	FMAP-S	0.44	0.66	0.60	0.80	0.46	0.68	0.50	0.80
	DPA-S	0.44	0.65	**0.67**	**0.90**	0.46	0.68	**0.65**	**0.90**

3.4.2 Attack Evaluation

In this section, we demonstrate the effectiveness of the proposed poisoning attacks. First, we test the predictions of the KGE model for the targeted victim client on the poisoned triples (**RQ1**) and on the original link prediction task (**RQ2**). Second, we test the effectiveness of the attacks under different settings (**RQ3**).

Attack Performance (RQ1)

Attack performance of malicious server. We implement two malicious server attacks introduced in subsection 3.3.1, including the fixed model poisoning (FMPA-S) and dynamic poisoning (DPA-S). We randomly select a victim client for the fixed model poisoning attack. To ensure fair comparison, we consistently choose the same victim client when executing the dynamic poisoning attack. We select 10 head entities as the head entity of poisoned triples on the victim client and set the client number is 3. The MRR and Hit@N values on the original task and poisoned triples are reported. The results are presented in Table 3.2.

In Table 3.2, the poisoned triples on victim model (PT on VC) column clearly illustrates that our proposed attack methods, FMPA-S and DPA-S, significantly enhance the link prediction performance on poisoned triples compared to the original FedE. It is worth noting that in most cases, DPA-S outperforms FMPA-S in terms of link prediction accuracy on poisoned triples. When utilizing TransE, RotatE, ComplEx, and DistMult as KGE models, DPA-S achieves an average MRR of 0.67 and Hits@10 of 0.88 on the poisoned triples in dataset-Fed3 (FB15k237-Fed3, NELL995-Fed3, WN18RR-Fed3, and CoDEx-M-Fed3). Conversely, FMPA-S achieves an average MRR of 0.56 and Hits@10 of 0.79 on the poisoned triples in dataset-Fed3. Furthermore, all the MRR and Hit@N values for poisoned triples under FedE settings are found to be 0. For example, when using TransE as the KGE model, the Hit@10 exceeds 0.9 on the FB15k-237 dataset under the DPA-S attack, indicating that over 90% of the poisoned triples on the victim client are predicted within the top-10 of the ranking list. Additionally, we can conclude that the WN18RR dataset is more vulnerable to poisoning attacks compared to other datasets due to its sparser structure, i.e., it has fewer neighbors per triple. By injecting few triples, the poisoning attacks can achieve a high attack success rate.

Table 3.3: Attack Performance of Client-Initiate Poisoning Attack (PT on VC means poisoned triples on victim model).

Dataset	TransE				RotatE			
	Mean		PT on VC		Mean		PT on VC	
	MRR	Hit@10	MRR	Hit@10	MRR	Hit@10	MRR	Hit@10
FB15K-237	0.41	0.60	0.68	0.90	0.41	0.61	0.58	0.80
NELL995	0.85	0.85	0.90	0.76	0.88	0.77	0.90	0.47
WN18RR	0.17	0.38	0.66	0.80	0.27	0.39	0.53	0.90
CoDEx-M	0.51	0.75	0.52	0.90	0.52	0.77	0.71	0.80

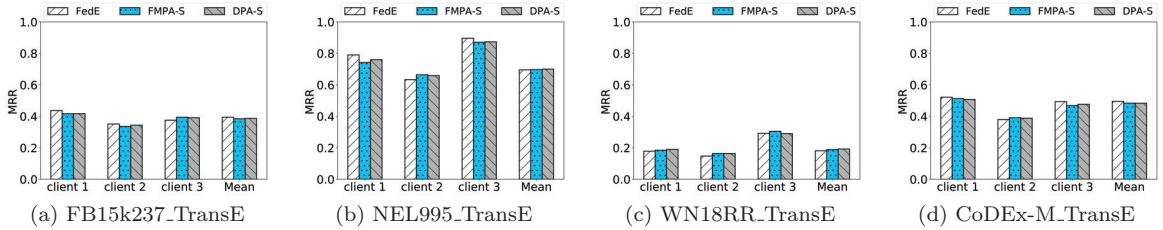
Dataset	ComplEx				DistMult			
	Mean		PT on VC		Mean		PT on VC	
	MRR	Hit@10	MRR	Hit@10	MRR	Hit@10	MRR	Hit@10
FB15K-237	0.37	0.55	0.36	0.70	0.38	0.56	0.41	0.60
NELL995	0.47	0.61	0.47	0.70	0.32	0.48	0.38	0.50
WN18RR	0.15	0.20	0.61	0.90	0.16	0.20	0.74	0.90
CoDEx-M	0.46	0.67	0.74	0.80	0.45	0.67	0.39	0.90

Attack performance of malicious client. We randomly select a malicious client and a victim client to evaluate the attack performance of the malicious client. We set the number of poisoned triples to be 10, and the link prediction results of the poisoned triples on the victim clients are presented in Table 3.3. Specifically, the malicious client trains its local KGE model using the poisoned dataset. The poisoned dataset consists of the original dataset of the malicious client and the poisoned triples inferred based on the changes in the local relation embeddings of the malicious client. As shown in Table 3.3, the client poisoning attack (CPA) achieves an average MRR of 0.59, Hits@10 of 0.81 on the poisoned triples in dataset-Fed3. In addition, we also observe that the client poisoning attack is more effective on the WN18RR dataset compared to other datasets, similar to what is observed in the poisoning attack on the server side. CPA achieves an average MRR of 0.64 and Hits@10 of 0.88 on the poisoned triples in WN18RR-Fed3.

In summary, these results demonstrate that our attack methods can efficiently elevate the ranks of poisoned triples, posing severe threats to knowledge graph embedding.

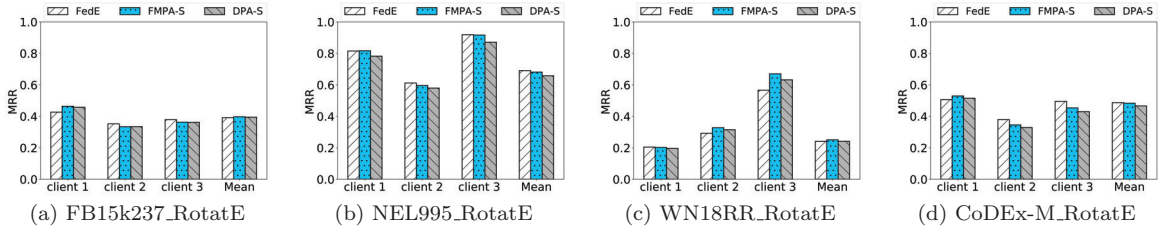
Clean Performance (Stealthiness) (RQ2)

We investigate the original test link prediction performance of different clients under our poisoning attack to validate the stealthiness of the attack. Our goal is to test how much the original link prediction performance remains unchanged. The MRR values of four dataset on TransE and RotatE are shown in Figure 3.3 and Figure 3.4. In the results, the performance differences between DPA-S, FMPA-S, and FedE are small across all local clients. This demonstrates that our attack can effectively balance the performance of the attack while maintaining the original link prediction performance as closely as attainable.



(a) FB15k237_TransE (b) NEL995_TransE (c) WN18RR_TransE (d) CoDEx-M_TransE

Figure 3.3: Clean Performance (Stealthiness) on TransE.



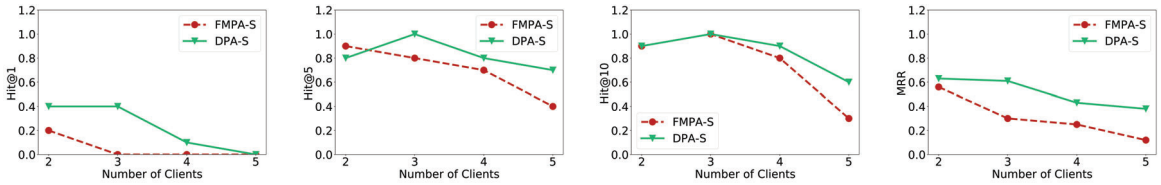
(a) FB15k237_RotatE (b) NEL995_RotatE (c) WN18RR_RotatE (d) CoDEx-M_RotatE

Figure 3.4: Clean Performance (Stealthiness) on RotatE.

Comparison of Different Settings (RQ3)

Impact of the number of clients. We explore the attack performance with different numbers of clients. Specifically, using TransE as the KGE model, we test the MRR and Hit@N values on the CoDEx-M dataset, varying the number of clients from 2

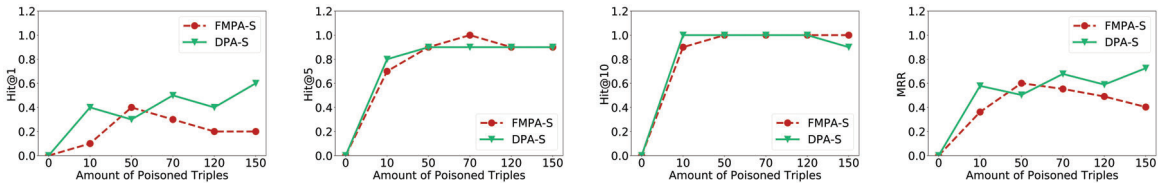
to 5. As shown in Figure 3.5, the metric values generally decrease as the number of clients increases. We speculate that the decreased attack effectiveness stems from the fact that the server's aggregated operations become diluted as the number of clients increases. We explore a potential solution to solve this problem and details are shown in subsection 3.4.4.



(a) CoDEx-M_TransE (b) CoDEx-M_TransE (c) CoDEx-M_TransE (d) CoDEx-M_TransE

Figure 3.5: Attack Performance of Different Numbers of Clients.

Impact of the number of poisoned triples. We use different poisoned datasets to investigate whether the effectiveness of the poisoning attack increases with the number of poisoned triples. The MRR and Hit@N results on CoDEx-M-Fed3 using the RotatE model as the KGE model are depicted in Figure 3.6. We vary the number of poisoned triples from 0 to 150. From the Figure 3.6, the metric values of on poisoned triples fluctuates within a certain range in CoDExM dataset. Therefore, under our settings, the number of poisoned triples does not have a significant impact on the attack success rate.



(a) CoDEX_M_RotatE (b) CoDEX_M_RotatE (c) CoDEX_M_RotatE (d) CoDEX_M_RotatE

Figure 3.6: Attack Performance of Different Numbers of Poisoned Triples.

Impact of server-client collusion. We conduct a comparison experiment to test the attack performance in collusion and non-collusion situations. If there is no collusion between the client and the server, the malicious client must compare its own uploaded embeddings and the global entity embeddings returned by the server to

Table 3.4: Attack Performance with or without Collusion

Collusion		**Non-collusion**	
MRR	**Hit@10**	**MRR**	**Hit@10**
0.68	0.90	0.28	0.40

identify the overlapping entities, and randomly select some overlapping entities as the target entities of the victim client. We perform a comparative experiment to test the attack performance in collusion and non-collusion situations. The following results are obtained (dataset: FB15K-237, KGE model: TransE) in Table 3.4. As shown in the results, the client-initiated poisoning attack can achieve better performance when the malicious client colludes with the server. This is because when they collude, the malicious client can obtain a more accurate set of victim entities to build a more accurate poisoned dataset.

3.4.3 Defense Evaluation (RQ4)

We finally test the effectiveness of the defense mechanisms introduced in subsection 3.3.3. For the server-initiate poisoning attack defense, we adopt two differential privacy-based methods, DPSGD and DP-Flames, to defend against attacker's membership inference attacks and weaken its poisoning attacks. We test the MRR and Hit@10 values of three schemes on the model TransE and datasets FB15k-237-Fed3 and CoDEx-M-Fed3. The results are shown in Table 3.5. From Table 3.5, we can see that after adopting defense mechanisms to the FB15k-237-Fed3 dataset, the MRR value decreases from 0.56 to 0.34 and 0.33, and the Hit@10 value decreases from 0.90 to 0.60 and 0.50. On the CoDex-M-Fed3, the defense effectiveness is better. These results demonstrate that the differential privacy-based defense mechanisms can achieve certain defensive effects, but there is still significant room for research.

For the client-initiate poisoning attack defense, we implement a prototype of DKGE by replacing the server with an Ethereum blockchain. 5 clients are installed with

Table 3.5: Attack Performance after Defense

Dataset	Schemes	MRR	Hit@10
FB15k-237	DPA-S	0.56	0.90
	DPSGD	0.34	0.60
	DP-Flames	0.33	0.50
CoDEx-M	DPA-S	0.61	1.00
	DPSGD	0.31	0.60
	DP-Flames	0.30	0.50

go-ethereum [49] nodes and trained through asynchronous aggregation for FKGE. Although the attack behavior in the system can be correctly detected and traced back to the attacker's identity, the convergence speed of the system is significantly slower than the previous FKGE. Therefore, further designs are needed to improve its availability.

Table 3.6: Attack Performance with or without Optimization

Number of Clients	**With optimization**		**Non-optimization**	
	Hit@10	**MRR**	**Hit@10**	**MRR**
2	1.000	0.4893	1.000	0.4372
3	0.900	0.4833	0.900	0.3832
4	0.900	0.4940	0.800	0.2133
5	0.900	0.4899	0.700	0.1333

3.4.4 Optimized Aggregation Mechanism

Figure 3.5 shows that the attack performance generally decreases as the number of benign clients increases. We investigate a method to address this issue and conduct additional experiments to test its effectiveness. The method involves aggregating only the embeddings of the targeted entities received from the victim client and the embeddings produced by the shadow model during the aggregation process, while excluding the embeddings of other benign clients, on the part of the malicious server. In this scenario, the poisoned embeddings of the targeted entities received by the victim client remain the same, despite an increase in the number of benign clients.

To test the attack performance following this optimization scheme, we conduct an experiment. The following results are obtained (dataset: CoDEx-M, KGE model: TransE) in Table 3.6.

Chapter 4

A Decentralized Knowledge Graph with Ownership-Preserving SPARQL Query

This chapter introduces Pistis, the first DKG with data ownership guarantees. Pistis incorporates two novel paradigms, called *owner-managed end-to-end encryption* and *collaborative query verification* into blockchain-based DKG management, enabling SPARQL queries with data ownership guarantees. Specifically, first, data owners encrypt their data individually and collaboratively construct a blockchain-maintained authenticated data structure (ADS) with a global key through secret sharing and secure multi-party computing. This ADS, indexed for querying KG data in ciphertext, ensures data ownership. Then, a cryptographic scheme called VO-SPARQL facilitates SPARQL queries on multi-owner encrypted KG data. It provides succinct proofs for the two-stage queries of SPARQL, including the subgraph queries based on the ADS and the aggregation on encrypted intermediate results based on a key-aggregate cryptographic primitive. Theoretical analysis and experimental evaluations demonstrate the performance benefits of Pistis with provable security.

4.1 Introduction

As one of the bold and innovative attempts in Web 3.0 [53, 80, 135], decentralized knowledge graph (DKG) is a new knowledge management technology to develop a global platform where everyone can share, manage, and exploit knowledge from diverse sources of data on the Web [92, 4, 6, 5]. Moreover, DKGs can provide high data availability and resilience, as the data is outsourced to and distributed across multiple nodes rather than centralized in a single location [6, 31, 59]. However, the decentralized nature of DKGs raises concerns about malicious threats, especially in the face of growing *Byzantine attacks* [35, 13, 133]. These malicious threats compromise the reliability of DKGs, necessitating robust defense strategies.

To ensure the reliability of DKGs, three key requirements must be addressed: *data integrity*, *query verifiability*, and *data ownership*, which are crucial in outsourced database scenarios [129, 87, 66]. Specifically, data integrity ensures that the data stored in the DKG is not tampered with or modified without authorization [69, 14]. Query verifiability ensures that query results made to the DKG can be verified as correct, complete, and fresh, preventing malicious query executions that could compromise the DKG [151, 131, 153, 154, 129]. Data ownership ensures that data owners have possession of complete control over the KG data, including the right to disclose raw data to others [118, 8]. Addressing these requirements is essential to ensure the reliability of DKGs and promote their wider adoption.

Emerging DKGs adopt blockchain to ensure data integrity [6, 92], the first requirement mentioned above. Because of their immutable nature, blockchains allow all nodes to collaborate on data updates and maintain a trusted historical record of all DKG data. Unfortunately, blockchain is not a silver bullet and does not inherently guarantee query verifiability and data ownership in DKGs. This is due to the following two reasons.

Data in Plaintext. Most blockchains store data in plaintext, which makes it acces-

sible to every DKG participant permanently. This renders raw data publicly available and data owners incapable to determine the data disclosure details, thus compromising the data ownership. An intuitive solution to ensure data ownership is to let data owners locally and independently encrypt their data before sharing it via blockchain. While this preserves data ownership, it hinders the ability of other participants to read the raw KG data and execute queries for multi-owner encrypted KG data; thus, damaging the DKG query verifiability.

Harsh-based Query Verification. Most blockchains only support hash-based verification for key-value query verifiability, where users use a given hash to validate the key-value query results. This hash-vased verification only works for simple key-value pair queries, while fails when dealing with other complex queries such as SPARQL, which are more common and thus requires more robust query verification strategies. A possible solution to achieving the query verifiability is to rely on the blockchain consensus and execute query operations on every node then agree on the same valid query result across all nodes. However, this approach requires all nodes to maintain the same global KG to execute local queries and verify the query results of the others, which leads to data owners to lose control of their data thus compromising ownership.

Contributions. In this work, we present Pistis, the first DKG that ensures data ownership without compromising data integrity and query verifiability. Our main idea is to incorporate two novel paradigms, called *owner-managed end-to-end encryption* and *collaborative query-verification*, into blockchain-based DKG management. Specifically, first, each data owner encrypts its data independently, and all owners collaboratively build a blockchain-maintained authenticated data structure (ADS) to share their ciphertext with the DKG. The ADS is a structured encrypted Merkle prefix tree managed by a global key via a new orchestration of secret sharing and secure multi-party computing. Using this ADS as an index, any participant can query KG data in ciphertext. The data owner can determine to whom to disclose the raw data in query results by managing its private key, thus ensuring data own-

ership. Then, to perform SPARQL queries (i.e., standard KG queries) on the multi-owner encrypted KG data with the guarantee of query verifiability, we design a new cryptographic scheme called VO-SPARQL (i.e., Verifiable and Ownership-preserving SPARQL). VO-SPARQL provides succinct verification proofs for the two-stage queries of SPARQL, including the subgraph queries based on the ADS and the aggregation on encrypted intermediate results based on a key-aggregate cryptographic primitive. Our contributions are summarized as follows.

- We develop Pistis, a blockchain-based DKG that ensures data ownership. Through a new cryptographic scheme VO-SPARQL with two novel paradigms, Pistis provides users with SPARQL query results and a succinct verification proof with data ownership guarantee throughout the query execution.

- We design an owner-managed end-to-end encryption scheme with a new authenticated data structure (ADS) called encrypted Merkle semantic trie (EMST). The scheme allows data owners to independently encrypt data to prevent loss of control, and the EMST achieves the functionality of subgraph querying for the data encrypted by different owners, ensuring data ownership in DKG.

- We design a collaborative query verification scheme called VO-SPARQL and implement it in a formalized way, which achieves verifiability of SPARQL queries in DKG through two steps while ensuring data ownership. VO-SPARQL includes EMST's workflow (e.g., initialization, update, and query) and a key-aggregate cryptographic primitive allowing one party to perform verifiable data aggregation on encrypted intermediate results queried from the EMST for final results.

- We conduct a comprehensive security analysis, develop a prototype for Pistis, and evaluate its performance under a widely-used benchmark largeRDFBench. The results show that Pistis achieves new functionalities and practical performance with a 45% reduction in index size compared to existing approaches.

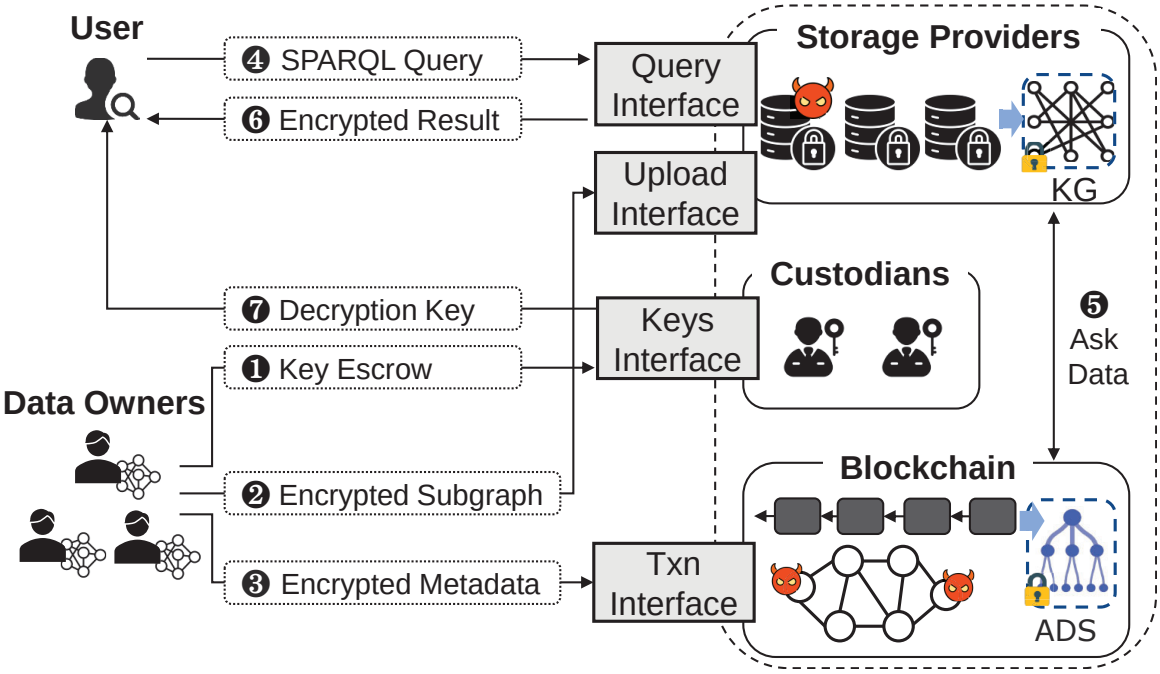


Figure 4.1: Architecture overview of Pistis.

4.2 The Pistis Model

4.2.1 System Model

As shown in Figure 4.1, Pistis provides KG data management services to two types of participants as follows.

- **Data owners** generate and share their RDF data as triple fragments. All triple fragments from different data owners make up a KG. Due to resource limitations, they expect to outsource the data to Pistis. However, they need to control their raw data and decide what to disclose and thus locally encrypt their respective data before outsourcing.
- **Users** access the KG by issuing the requests of SPARQL query to Pistis. The query involves the encrypted triple fragments from one or more data owners.

To meet the demand of data owners and users, Pistis is a DKG comprised of the

following three roles of nodes.

- **Blockchain** plays the role of a trust anchor in Pistis by building a public and immutable ledger via consensus. The ledger records data owners' RDF data metadata for a global index used to query and authenticate.
- **Storage providers** involve a distributed data storage protocol similar to IPFS [19]. They can provide efficient, usable, and cheap off-chain storage. Each storage provider stores a part of the data (i.e., a subgraph of a KG), and an RDF triple of a subgraph can be backed up to multiple storage providers with an address that can locate the relevant storage providers storing it. Storage providers can collaboratively execute a SPARQL query on multiple subgraphs.
- **Custodians** are only responsible for managing keys in Pistis. The number of custodians depends on the choice of MPC adopted in Pistis. In the following, we consider the case of 2PC, thus the system will set up two custodians. The custodian selection policy will be discussed in section 4.5.

4.2.2 Threat Model

In Pistis, the data owners and users are honest. The custodians are semi-honest, do not collude with each other and strictly follow the protocol's instructions. Blockchain nodes and storage providers can be malicious for various reasons, such as program glitches, security vulnerabilities, and commercial interests. The proportion of malicious blockchain nodes will not exceed the fault threshold of the blockchain (e.g., 1/2 in PoW or 1/3 in PBFT). Moreover, for each RDF triple, at least one storage node storing it is honest. Each participant does not maliciously communicate with the others in violation of the peer-to-peer network. Each adversary is computationally bounded and cannot break standard cryptographic primitives, e.g., finding hash collisions or forging digital signatures.

There are two types of adversarial attacks in Pistis: (*i*) ***Data breaches*** [118]: the semi-honest custodians, malicious blockchain nodes, or malicious storage providers may try to independently infer or learn sensitive information about data owners' data due to various interests. (*ii*) ***Data tampering*** [61, 142]: the malicious blockchain nodes or storage providers can launch data tampering attacks. They can behave arbitrarily, e.g., forge or tamper with their local data and query results, or provide outdated information. It is a stronger adversarial attack than data breaches.

4.2.3 Workflow

The workflow of Pistis relies on two key designs, including an encrypted Merkle semantic trie (EMST, refer to subsection 4.3.2), and a verifiable and ownership-preserving SPARQL query scheme (refer to subsection 4.3.3). As shown in Figure 4.1, Pistis consists of the following five phases.

Phase 1: ***Initialization.*** Each data owner generates a pair of public and private keys to encrypt and decrypt its own RDF data, respectively. After that, it uses secret sharing to share its private key with the two custodians through the keys interface, and the custodians collaboratively generate a global key used to build the EMST later (Figure 4.1-❶).

Phase 2: ***Data outsourcing.*** A data owner uses its private key to encrypt its data and then outsources the encrypted data to storage providers through the upload interface (Figure 4.1-❷).

Phase 3: ***Index updating.*** When a data owner outsources its encrypted data to the storage providers, it packs some metadata (e.g., the hash and the address) of the encrypted data into a blockchain transaction. The transaction will be submitted through the transaction (Txn) interface and will be committed to the blockchain (Figure 4.1-❸). The blockchain nodes then update the global index maintained in the blockchain according to the transaction via a consensus.

Phase 4: ***Query processing.*** To query the DKG, a user can send a SPARQL query request to any storage provider through the query interface (Figure 4.1-❹). Next, the storage provider asks blockchain nodes to search for the addresses of the relevant data by the on-chain global index, and aggregate the relevant data (Figure 4.1-❺). Finally, the storage provider sends the combination of the final query results and their verification proofs to the user (Figure 4.1-❻).

Phase 5: ***Verification.*** After receiving the encrypted query results and the corresponding proofs, the user decrypts the results (Figure 4.1-❼) with the aid of the two custodians and verifies the results based on the proof provided by the blockchain nodes and storage providers.

4.2.4 Design Goals

- **Data ownership.** PISTIS should prevent data breaches attacks without the data owner's authorization, i.e., the user knows nothing about the raw data except the query result decrypted by the owner. Likewise, blockchain and storage nodes can only conduct authorized operations on sensitive data without knowing its content.

- **Query verifiability.** PISTIS should prevent data tampering attacks, i.e., ensure the query results' *correctness* (i.e., none of the RDF triples returned as results have been tampered with), *completeness* (i.e., no valid result is missing from the query results), and *freshness* (i.e., the query results are based on the latest version of the DKG).

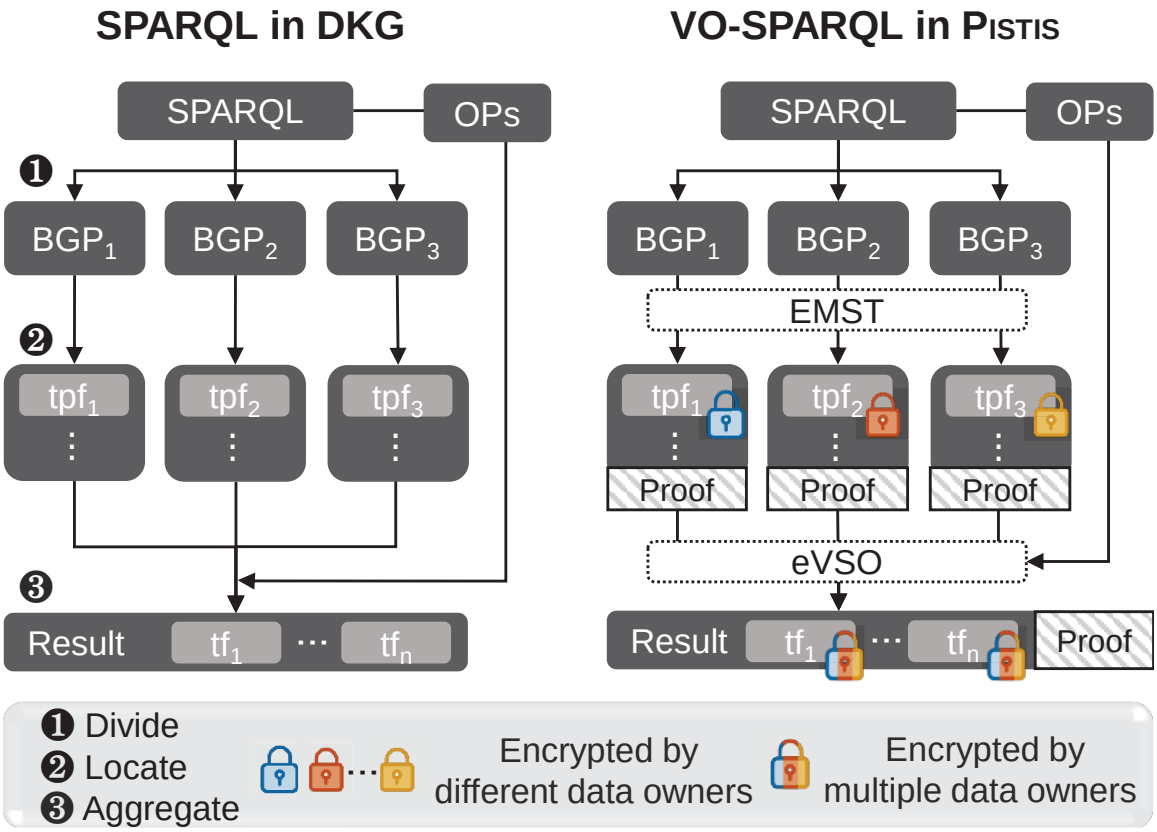


Figure 4.2: Comparison of SPARQL query in DKG and the verifiable and ownership-preserving SPARQL query in Pistis.

4.3 Methodology

4.3.1 Roadmap

As shown in Figure 4.2, to process a SPARQL query in a DKG, a common workflow [5, 6, 4] consists of three steps: 1) dividing the SPARQL query into multiple BGPs, 2) locating the relevant triple pattern fragments matching each BGP, and 3) aggregating all triple pattern fragments with set operators (OPs) to get the final results. Following this workflow, to implement a verifiable and ownership-preserving SPARQL query in DKG, we propose a cryptographic scheme called VO-SPARQL (i.e., verifiable and ownership-preserving SPARQL) in Pistis. In particular, in the locating process, Pistis adopts a structured encryption-based new Merkle tree variant, EMST,

and constructs it by blockchain consensus for verifiable and ownership-preserving triple pattern queries. In the aggregating process, Pistis designs an asymmetric key-aggregate-based new VSO algorithm, eVSO, for verifiable and ownership-preserving set operations between multiple triple pattern fragments encrypted by different data owners. In the following, we will introduce the design of EMST and eVSO and how VO-SPARQL integrates them into the SPARQL workflow in DKG with the participation of blockchain, storage providers, and custodians.

4.3.2 Encrypted Merkle Semantic Trie

Strawman

We first design a strawman ADS (i.e., a basic solution) with the ability of verifiable triple pattern queries based on a Merkle prefix tree (MPT). An MPT is an extension of a prefix tree (also known as a trie) by including a hash of the last prefix bit and value in each leaf, and the hash of the last bit and child hashes in internal nodes. It provides verifiable prefix-matching functionality that we depend on to implement verifiable triple pattern queries. In this strawman, we can construct an MPT with verifiable triple pattern queries, called Merkle Semantic Trie (MST). An MST of depth d contains three types of nodes: 1) one root node *root* storing the Merkle root hash is located at the top layer, 2) $d-2$ layers of branch nodes, each of which stores an individual character c and a hash value of the combination of its child nodes, and 3) one layer of leaf nodes, each of which stores an individual character c, a hash value of itself and some pointers. These pointers point to some addresses of RDF triples that belong to a triple fragment with the same prefix, i.e., each RDF triple in this fragment has at least one element matching the prefix on the path from the root node to that leaf node.

Figure 4.3 (a) shows an example of an MST. Particularly, f_1, f_2 and f_3 are three triple fragments with some RDF triples that consist of items aa, ab, and ba. For example,

$f_1 = \{\langle aa, aa, aa\rangle\}$, $f_2 = \{\langle ab, ab, ab\rangle\}$, and $f_3 = \{\langle ba, ba, ba\rangle\}$. To simplify the example, we set the three items in each triple to be the same. Given a triple fragment f_1' from a storage provider, to verify whether it corresponds to a triple pattern with a given item aa (i.e., f_1' is equal to f_1), a user who only holds the Merkle root $h6$ can recover a Merkle root ($h6'$) based on the hash of f_1' (i.e., $h1'$) and a Merkle proof ($h2$ and $h5$), and then verify whether $h6$ and $h6'$ are the same. If they are the same, it proves that the storage provider provides the correct triple pattern fragment.

However, the strawman ADS only supports verifiable triple pattern queries for plaintext KG data (even if the index only contains addresses of query results, plaintext indexing information and user queries can still compromise data ownership). A naive approach is to encrypt the content of each node independently without changing the structure of the MST, and the user can use the same key to encrypt the triple patterns of its queries and perform these triple pattern queries under ciphertext. This approach protects privacy to a certain extent, but some access and search patterns (refer to subsection 4.4.1) are still leaked, including the number of children of different nodes, the same ciphertext for the same characters on different layers, and the same increasing order of the characters of all layers. We also refer the readers to [36] for formal definitions of these leaks. A basic idea is to design a new structured encryption scheme to convert an MST to an EMST and protect the above-mentioned information from being leaked under the premise that the structure of the MST is also unchanged.

Challenge

Most of the structured encryption schemes [68, 36] break the correlation of messages and their ciphertext by inducing a random permutation between them to hide the part of access patterns. However, it is challenging to convert an MST to an EMST by this idea since inducing a random permutation for all the nodes in the whole MST will disrupt the tree structure and let it lose the prefix-matching functionality.

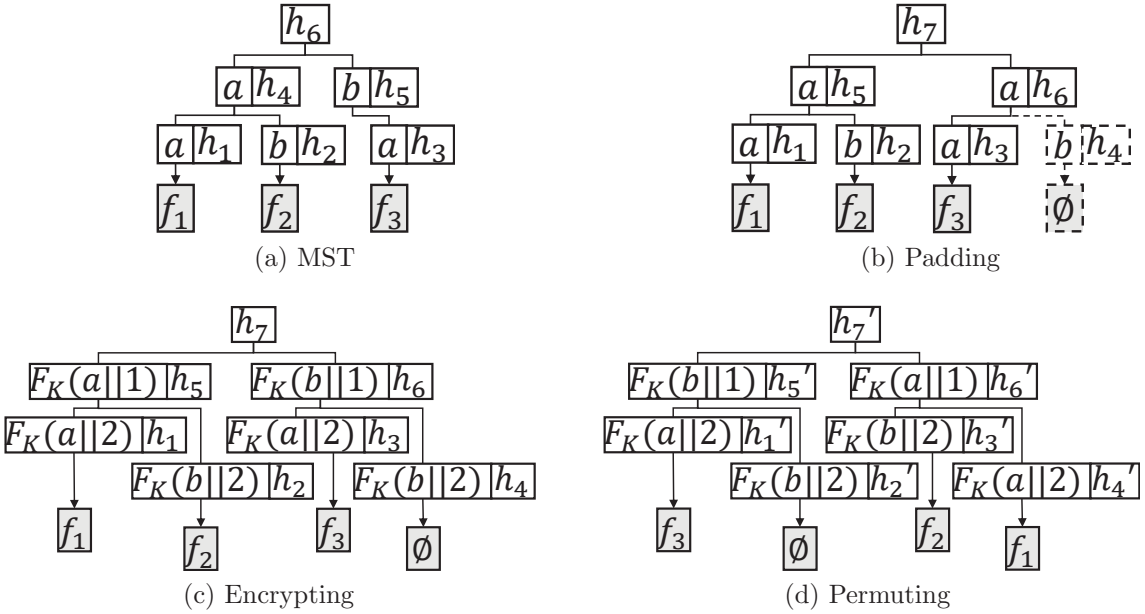


Figure 4.3: The process to convert an MST to an EMST (ϕ is an empty triple fragment, $F_K()$ is a pseudo-random functions with a private key K, and $\|$ is a cascading symbol).

Design

Therefore, to convert an MST to an EMST, we need a new random permutation for the MST with its prefix-matching functionality guarantee. The key of the challenge to achieve it is to keep the connection relationship between different layers in case the node position of MST is disturbed. To overcome this challenge, we design a subtree-based random permutation (STRP) algorithm to convert MST to EMST.

a) *STRP algorithm.* The encryption is done by (1) padding the child nodes of each non-leaf node in an MST to be of the same length; (2) For each non-leaf node in the MST, encrypting the character of it using the output of a PRF; (3) For each non-leaf node in the MST, randomly permuting the location of its child nodes using a PRP. The purpose of step (1) is to help us hide the number of data items of non-leaf nodes' child nodes, and the purpose of steps (2) and (3) is to prevent index information leakage and hide the part of access patterns. The formal description of STRP is shown in two protocols `AddToken` and `Add` of the pseudo-code of Figure 4.4.

An example of the conversion process of EMST is shown in Figure 4.3. In Figure 4.3 (b), the dashed box represents the padding part of the unbalanced MST shown in Figure 4.3 (a). After the padding process, each non-leaf node of the MST has the same number of child nodes. Figure 4.3 (c) shows the encryption process of MST. In this process, the character with its level cascaded of each non-root node of the MST is encrypted by a PRF. The final EMST with the permutation operation completed is shown in Figure 4.3 (d).

b) *Operations for EMST.* As mentioned in subsection 4.2.3, the EMST will be updated after a series of transactions submitted by data owners when the metadata of their RDF data are committed to the blockchain. The update process involves four different operations on the EMST, including `Insert`, `Change`, `Delete`, and `Query`. In structured encryption, the user needs to use a token (add token or query token, refer to section 2.4) to manipulate a structure. To simplify the steps, we omit the steps of generating tokens when introducing these operations. algorithm 2 shows the detailed processing of the `Insert` operation. In the `Insert` operation, each feature of the content of an RDF triple I (i.e., subject, predicate, or object) is individually encrypted with the private key K, and each encrypted character is recursively inserted into EMST. After that, the address Cid stored in the storage node of the triple is inserted into the leaf node of EMST. Operations `Change` and `Delete` are similar to the processing of operation `Insert`. The difference between these two operations and `Insert` operation is that `Insert` operation may create a new path on EMST if the keyword corresponding to the path has not appeared in the system before, while operations `Change` (or `Delete`) only need to find an existing path and update (or delete) the Cid stored on the leaf node.

Recall in subsection 4.2.3 the EMST can be used to find the relevant data (i.e., the intermediate results of a SPARQL query, which are some triple pattern fragments). Note that in subsection 4.3.3, the EMST can execute a set of triple patterns (i.e, a `BGP`) in batches. The pseudo-code of algorithm 3 shows the detailed processing of the

Algorithm 2: `Insert` operation of EMST

Function *Insert (EMST, I, K, Cid)*:
  Input : EMST root node *root*, the content of an RDF triple I, and a private key K
  Output: the root node *root*′ of a new state of EMST
  $node \leftarrow root$;
  foreach $item_i \in I$ **do**
    foreach c_j *in* $item_i$ **do**
      $c'_j \leftarrow F_K(c_j \| j)$;
      if $node.child[P_K(c'_j)] = null$ **then**
        $node.child[P_K(c'_j)] \leftarrow New(node)$;
      $node = node.child[P_K(c'_j)]$;
    add Cid to $node.CID$;
    $node.hash \leftarrow hash(node.c'_j || node.CID)$;
    if $node \; != root$ **then**
      $node.hash \leftarrow hash(node.c'_j || node.child)$;
      $node = node.parent$;
  return *node*;

`Query` operation. In the `Query` operation, each prefix in the triple pattern is encrypted by the private key K first, and then is used to match a path in the EMST. If a triple pattern has more than one variable, the results `R` will contain multiple triple pattern fragments. Regardless of whether the match is successful or not (i.e., null result), the proof corresponding to the result will be returned.

4.3.3 VO-SPARQL Scheme

In the section, we describe the design of the VO-SPARQL scheme in our Pistis system and its usage. The VO-SPARQL scheme Ω = (`InitGlobal, OffchainStore, AddToken, Add, QueryToken, Query, Aggregate, Verify`) consists of eight protocols which we describe at a high-level below. The detailed pseudo-code is shown in Figure 4.4 and Figure 4.5.

Parties. Pistis is designed to be executed among: a large (dynamic) number of data owners $\mathbf{DO}_1, \ldots, \mathbf{DO}_\theta$, a large (dynamic) number of storage providers $\mathbf{SP}_1, \ldots, \mathbf{SP}_\tau$,

Algorithm 3: `Query` operation of EMST

Function *Query (EMST, tp, K)*:
  Input : EMST root node *root*, a triple pattern *tp*, and a private key K
  Output: the query result `R` with a address set CID, the Merkle proof $Mproof$ of `R`
  $node \leftarrow root$;
  foreach $keyword_i \in tp$ **do**
    foreach $character_j$ *in* $keyword_i$ **do**
      $character'_j \leftarrow F_K(character_j \| j)$;
      add $node.hash$ to $mproof$;
      if $node.child[P_K(character'_j)] = null$ **then**
        add ϕ to `R`;
        add the Merkle proof $mproof$ to $Mproof$; Break;
      $node = node.child[P_K(c'_j)]$;
    add CID to `R`;
    add the Merkle proof $mproof$ to $Mproof$;
  return *R*, $Mproof$;

two custodians $\mathbf{C}_1$ and $\mathbf{C}_2$, a large (dynamic) number of blockchain nodes $\mathbf{BN}_1, \ldots,$ $\mathbf{BN}_\rho$, and a user $\mathbf{Q}$.

Initializing a global index. To initialize the system, the two custodians $\mathbf{C}_1$ and $\mathbf{C}_2$ execute Ω.`InitGlobal` to generate an empty EMST on the blockchain which we call the global index and provides each custodian with a share of the global key. In detail, $\mathbf{C}_1$ and $\mathbf{C}_2$ execute a 2PC function to 1) generate a global key K, 2) generate an empty MST and use K to encrypt MST to EMST , and 3) use a Shamir secret sharing scheme to distribute K to K_1 and K_2 and share them to $\mathbf{C}_1$ and $\mathbf{C}_2$ respectively. And then they send EMST to the blockchain. The global index can support prefix-based triple pattern queries for RDF triples and return their addresses in the storage providers. With the addresses, the storage providers can get the ciphertext of relevant RDF triples and execute the aggregation processing to provide final query results to the user.

Adding a new data item. A new record with a triple item I is added by a data owner $\mathbf{DO}_i$ by executing the Ω.`OffchainStore` with the storage nodes **SP**,

- $\texttt{InitGlobal}_{\mathbf{C}_1,\mathbf{C}_2}(1^k, 1^k)$:

  1) $\mathbf{C}_1$ randomly samples $r_1 \xleftarrow{\$} \{0,1\}^k$ and $\mathbf{C}_2$ randomly samples $r_2 \xleftarrow{\$} \{0,1\}^k$;

  2) $\mathbf{C}_1$ and $\mathbf{C}_2$ execute $(K_1, K_2, \mathsf{EMST}) \leftarrow \mathcal{F}^f_{2PC}(r_1, r_2)$
  where $f(r_1, r_2)$:
  a) $K = \Pi.Gen(1^k, r_1 \oplus r_2)$
  b) generate an empty MST and use K to encrypt MST, output EMST to $\mathbf{C}_1$
  c) $(K_1, K_2) \leftarrow SS.share(K, 2, 2)$
  d) output K_1 to $\mathbf{C}_1$ and output K_2 to $\mathbf{C}_2$

  3) $\mathbf{C}_1$ sends EMST to blockchain.

- $\texttt{OffchainStore}_{\mathbf{DO}_i}(t)$:

  1) $\mathbf{DO}_i$ executes `RSA.Gen()` to generate a key-pair (Pk_i, Sk_i), broadcasts Pk_i, splits Sk_i by $(Sk_1, Sk_2) \leftarrow SS.share(Sk_i, 2, 2)$, and sends Sk_1 to $\mathbf{C}_1$ and Sk_2 to $\mathbf{C}_2$.

  2) When a data owner $\mathbf{DO}_i$ wants to outsource an item I with an RDF triple (s, p, o) to the storage providers, it uses Pk_i to encrypt all the three elements of the triple separately, and sends $hash(I)$ and {`RSA.Enc(`Pk_i`,` $I.s$`), RSA.Enc(`Pk_i`,` $I.p$`), RSA.Enc(`Pk_i`,` $I.o$`)`} to the storage providers.

  3) The storage providers return the storage address Cid to $\mathbf{DO}_i$.

- $\texttt{AddToken}_{\mathbf{C}_1,\mathbf{C}_2,\mathbf{DO}_i}(K1, K2, I)$:

  1) $\mathbf{DO}_i$ parses I as $(I.s, I.p, I.o)$;

  2) for each member $I.m$ in I, do:
  a) $\mathbf{DO}_i$ computes $(p_1, p_2) \leftarrow \texttt{SS.Share}(I.m, 2, 2)$ and sends p_1 to $\mathbf{C}_1$ and send p_2 to $\mathbf{C}_2$;
  b) $\mathbf{C}_1$ and $\mathbf{C}_2$ execute $\texttt{ATK} \leftarrow \mathcal{F}^f_{2PC}(K1, K2, p_1, p_2)$
  where $f(K1, K2, p_1, p_2)$:
  $(I.m) \leftarrow \texttt{SS.Recover}(p_1, p_2)$;
  for each character c_j in $I.m$, do:
  $c'_j \leftarrow F_K(c_j \| j)$ and add $c'_j \| i \| P_K(c_j)$ to atk_i
  c) add atk_i to $\texttt{ATK}$

  3) add Cid to $\texttt{ATK}$ and sent $\texttt{ATK}$ to $\mathbf{DO}_i$.

- $\texttt{Add}_{\mathbf{BN}_l}(\mathsf{EMST}, \texttt{ATK})$:

  1) $\mathbf{BN}_l$ parse $\texttt{ATK}$ as $atk_1, atk_2, \ldots, atk_i$;

  2) Let $node = \mathsf{EMST}.root$. For each atk_i in $\texttt{ATK}$, do:
  a) let $j = 1$, $kw = c'_j$ in atk_i. $\mathbf{BN}_l$ do:
  a) if $node.child[P_K(c_j)] = null$,
  b) $node = node.child[P_K(c_j)]$, and $i++$
  b) $node.add(\texttt{ATK}.Cid)$

  3) $\mathbf{BN}_l$ broadcasts EMST to other blockchain nodes;

Figure 4.4: VO-SPARQL query scheme in Pistis (part 1).

- $\texttt{QueryToken}_{\mathbf{C}_1,\mathbf{C}_2,\mathbf{Q}}(K1, K2, \texttt{BGP})$:
  1) $\mathbf{Q}$ parses BGP as $\{tp_1, tp_2, \ldots, tp_\alpha\}$
  2) For each tp_i in BGP, do:
  a) compute $(q_1, q_2) \leftarrow \texttt{SS.Share}(tp_i, 2, 2)$ and sends q_1 to $\mathbf{C}_1$ and send q_2 to $\mathbf{C}_2$;
  b) $\mathbf{C}_1$ and $\mathbf{C}_2$ execute $\texttt{QTK} \leftarrow \mathcal{F}^f_{2PC}(K1, K2, q_1, q_2)$
  where $f(K1, K2, q_1, q_2)$:
  $tp_i \leftarrow \texttt{SS.Recover}(q_1, q_2)$;
  for each character c_i in $tp_i.keyword$, do: \\ assume that there is one given keyword in tp_i
  $c'_i \leftarrow F_K(c_i \| i)$ and add $c'_i \| i \| P_K(c_i)$ to qtk_i
  c) add qtk_i to QTK
  3) sent QTK to $\mathbf{Q}$.

- $\texttt{Query}_{\mathbf{SP}_l}(\mathsf{EMST}, \texttt{QTK})$:
  1) $\mathbf{SP}_l$ parse QTK as $qtk_1, qtk_2, \ldots, qtk_i$;
  2) Let $node = \mathsf{EMST}.root$. For each qtk in QTK, do:
  a) let $j = 1$, $kw = c'_j$ in qtk_i. $\mathbf{SP}_i$ do:
  a) if $node.child[P_K(c_j)] = null$, break;
  b) $node = node.child[P_K(c_j)]$, and $i++$
  b) add $node.CID$ and $mproof$ to R and $Mproof$

- $\texttt{Aggregate}_{\mathbf{SP}_l}(\texttt{R})$:
  1) $\mathbf{SP}_j$ parses $R.tpf$ as $\{tpf_1, tpf_2 \ldots, tpf_\gamma\}$ and extracts all public keys $\{Pk_1, Pk_2, \ldots, Pk_j\}$ from R;
  2) For each tpf_i in R, $\mathbf{SP}_j$ caculates $E(tpf_i) \leftarrow \texttt{RSA.Enc}(Pk_1, Pk_2, \ldots, Pk_\varphi, tpf_i)$ and add $E(tpf_i)$ to $ETPF$;
  3) $S \leftarrow$ aggregate(ETPF);
  4) $\pi \leftarrow$ prove(ETPF, $Pk_{\mathbf{SP}_j}$);
  5) $\mathbf{SP}_j$ sends $\{S, \pi, \texttt{R}, Mproof\}$ to $\mathbf{Q}$

- $\texttt{Verify}_{\mathbf{C}_1,\mathbf{C}_2,\mathbf{Q}}(S, \pi, \texttt{R}, Mproof)$:
  1) $\mathbf{Q}$ verifies R by its Merkle proof $Mproof$.
  2) For each tpf_i in R, $\mathbf{Q}$ calculates $acc(tpf_i)$ and adds it to acc.
  3) $\mathbf{Q}$ computes $vr \leftarrow VerifyProof(acc, \pi)$ and verifies vr;
  4) $\mathbf{Q}$ asks $\mathbf{C}_1$ and $\mathbf{C}_2$ the decrypt keys to decrypt the results. $\mathbf{C}_1$ and $\mathbf{C}_2$ recover these keys and send them to $\mathbf{Q}$.

Figure 4.5: VO-SPARQL query scheme in Pistis (part 2).

Ω.AddToken protocol with two custodians $\mathbf{C}_1$ and $\mathbf{C}_2$, and Ω.Add with the blockchain nodes $\mathbf{BN}$. At a high level, these protocols work as follows. First, $\mathbf{DO}_i$ execute Ω.OffchainStore to generate a pair of RSA keys, Pk_i and Sk_i, shares Sk_i to $\mathbf{C}_1$ and $\mathbf{C}_2$, broadcasts Pk_i, and uses Pk_i to encrypt each element (i.e., subject, predicate,

or object) of the triple item I separately. After that, $\mathbf{DO}_i$ sends the ciphertext of I to **SP** and gets its address Cid. Second, $\mathbf{DO}_i$ splits the index information (i.e., prefix) of I into two shares p_1 and p_2 and sends them to $\mathbf{C}_1$ and $\mathbf{C}_2$ separately and asks them to execute Ω.`AddToken` to generate an add token. In detail, $\mathbf{C}_1$ and $\mathbf{C}_2$ use 2PC to securely compute a function that: 1) recovers the global key K from their key shares; 2) recovers the index information from their pair shares p_1 and p_2; and 3) uses PRF and PRP to encrypt and permute all characters in the keywords, and add them to an add token `ATK`. Third, after receiving the add token, all blockchain nodes **BN** execute the Ω.`AddToken` protocol and update the on-chain global index EMST through consensus.

Querying a global index. When a user **Q** wants to query the DKG with a `BGP` $= \{tp_1, tp_2, \ldots, tp_\alpha\}$, it first executes Ω.`QueryToken` in conjunction with $\mathbf{C}_1$ and $\mathbf{C}_2$ to generate a query token. In detail, **Q** splits the BGP into two shares q_1 and q_2 and sends them to $\mathbf{C}_1$ and $\mathbf{C}_2$ separately. $\mathbf{C}_1$ and $\mathbf{C}_2$ use 2PC to securely compute a function that: 1) recovers the global key K from their key shares; 2) recovers the `BGP` from their pair shares q_1 and q_2; and 3) for each tp_i in `BGP`, uses PRF and PRP to encrypt and permute all characters of its prefixes, and add them to a query token `QTK`. Then **Q** sends `QTK` to a storage provider $\mathbf{SP}_l$ and asks it to execute Ω.`Query` protocol to search for some relevant triples through the on-chain global index. In detail, $\mathbf{SP}_l$ first parses `QTK` as $\{qtk_1, qtk_2, \ldots, qtk_\beta\}$, and then for each qtk in `QTK`, $\mathbf{SP}_l$ searches in EMST through any blockchain node by matching all characters of qtk with all layers of EMST. Finally, if the match is processed successfully, $\mathbf{SP}_l$ adds the addresses of query results of qtk and their relevant Merkle proofs into `R` and $Mproof$ separately. `R` and $Mproof$ will be sent to **Q** after the entire query process is over.

Aggregating intermediate results by eVSO. To get the final query results, $\mathbf{SP}_l$ executes the Ω.`Aggregate` protocol to aggregate the intermediate query results from the global index. This protocol contains an asymmetric key-aggregate-based VSO algorithm `eVSO`. In the process of `eVSO`, first, $\mathbf{SP}_l$ fetches and parses `R` as different

triple pattern fragments $\{tpf_1, tpf_2, \ldots, tpf_\gamma\}$, and extracts a public key set $Pks = \{Pk_1, Pk_2, \ldots, Pk_\varphi\}$ from `R`. Then, $\mathbf{SP}_l$ uses Pks to re-encrypt all elements of all relevant triples in different triple pattern fragments. Next, $\mathbf{SP}_l$ aggregates these encrypted triple pattern fragments (ETPF) by executing some set operations for them and gets the final query results S, and generates a verification proof of S through VSO. Finally, after the aggregation process is completed, $\mathbf{SP}_l$ sends S, `R`, and the verification proofs to $\mathbf{Q}$.

Verifying query results. After receiving the final results, the user $\mathbf{Q}$ needs to execute the Ω.`Verify` protocol to verify the result in three steps. First, it uses the CID in `R` and their Merkle proofs $Mproof$ to verify whether the storage provider $\mathbf{SP}_l$ is correctly asking the blockchain nodes to perform the triple pattern query through the on-chain global index. Second, it uses the accumulated values and the proof of eVSO to verify whether the $\mathbf{SP}_j$ is correctly performing the aggregation processing on the intermediate query results. Third, $\mathbf{Q}$ asks $\mathbf{C}_1$ and $\mathbf{C}_2$ the decrypt keys to decrypt the results. $\mathbf{C}_1$ and $\mathbf{C}_2$ recover these keys and send them to $\mathbf{Q}$.

Cost Analysis. Here we give the time and space complexity of each function involved in VO-SPARQL. Ω.`InitGlobal` has $O(k)$ time and space complexity, where k is the length of the global key. Ω.`OffchainStore` has $O(1)$ time complexity and $O(k_R)$ space complexity, where k_R is the length of the RSA key. Ω.`AddToken` has $O(l)$ time and space complexity, where l is the length of characters in the triple. Ω.`Add` has $O(l)$ time complexity and $O(1)$ space complexity. Ω.`QueryToken` has $O(\alpha \times l_q)$ time and space complexity, where α is the number of triple patterns and l_q is the length of characters in the triple pattern. Ω.`Query` has $O(\alpha \times l_q)$ time complexity and $O(\alpha)$ space complexity. Ω.`Aggregate` has $O(\alpha \times \varphi)$ time complexity and $O(\alpha + \varphi)$ space complexity, where φ is the number of relevant owners of the query. Ω.`Verify` has $O(\alpha + \varphi)$ time complexity and $O(\alpha)$ space complexity.

4.4 Analysis

In this section, we first formalize the security of our design in the ideal/real-world paradigm and give the proof of the security, and then give a verifiability analysis of queries in Pistis.

4.4.1 Ideal/real-world Paradigm

The goal of the ideal/real world paradigm is to prove that the scheme in Pistis can protect against adaptive adversaries from potential information leakage during interaction with the scheme. Intuitively, we require that the view of an adversary (i.e., the encrypted data structure, the sequence of ciphertexts, and the sequence of tokens) generated from any adaptive query strategy be simulatable given the leakage information. We first define a leakage function $\mathcal{L}$ for Pistis, which describes the information revealed in the verifiable SPARQL query process. The input of the query protocol is a KG $\mathcal{G}$ and a SPARQL query with a `BGP`. $\mathcal{L}(\mathcal{G}, \texttt{BGP})$ is defined as follows:

Definition 5 ($\mathcal{L}(\mathcal{G}, \texttt{BGP})$)**.** *The leakage function $\mathcal{L}$ involves access pattern and search pattern.*

- ***Access pattern****. The access pattern is a mapping relation between the submitted token and the corresponding encrypted RDF triples.*
- ***Search pattern****. The search pattern is the difference between two input tokens. Namely, it indicates whether a token has been added or searched.*

$\mathcal{L}$ is always considered as default leaked information in searchable symmetric encryption [68], and the Adaptive $\mathcal{L} - security$ can be defined as follows.

Definition 6 (Adaptive $\mathcal{L} - security$)**.** *Let Ω = (`InitGlobal`, `Off-chainStore`, `AddToken`, `Add`, `QueryToken`, `Query`, `Aggregate`, `Verify`) be a VO-SPARQL scheme.*

Let $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_q)$ *and* $\mathcal{S} = (\mathcal{S}_0, \ldots, \mathcal{S}_q)$ *be an adversary and a simulator, respectively, where* $q \in \mathbb{G}$. *We define the* $\boldsymbol{Real}^{\mathcal{A}}_{\Omega}(1^k)$ *experiment and the* $\boldsymbol{Ideal}^{\mathcal{A}}_{\mathcal{L},\mathcal{S}}(1^k)$ *experiment as follows.*

$\mathbf{Real}^{\mathcal{A}}_{\Omega}(1^k)$: In the real-world execution every party has access to ideal $\mathcal{F}_{2PC}$ functionalities. At round 0, $\mathbf{C}_1$ and $\mathbf{C}_2$ execute Ω.`InitGlobal` to generate an encrypted index EMST and send it to $\mathcal{A}$, and each data owner $\mathbf{DO}_i$ executes Ω.`OffchainStore` with $\mathbf{SP}$. Then, $\mathcal{A}$ adaptively chooses a polynomial number of commands $(comm_1, \ldots, comm_q)$ of the form $comm_r = (\mathbf{DO}_i, op_r)$, where op_r is either an add operation (`AddToken`, `Add`) or a query operation (`QueryToken`, `Query`, `Aggregate`, `BGP`). At round r $(1 \leq r \leq q)$, $\mathcal{A}_r$ executes op_r by Ω. After q round interactions, $\mathcal{A}$ produces a b bit as the output.

$\mathbf{Ideal}^{\mathcal{A}}_{\mathcal{L},\mathcal{S}}(1^k)$: In the ideal-world execution every party has access to ideal $\mathcal{F}_{2PC}$ functionalities. At round 0, $\mathcal{S}_0$ randomly generates an index EMST^* and an encrypted KG $\mathcal{G}^*$ by utilizing $\mathcal{L}(\mathcal{G}, \mathtt{BGP})$, and sends EMST^* to $\mathcal{A}$. Then, $\mathcal{A}$ adaptively chooses a polynomial number of commands $(comm_1, \ldots, comm_q)$ of the above form. At round r $(1 \leq r \leq q)$, $\mathcal{A}_r$ reviews the previous requests and generates f_r adaptively. If op_j is an add, with $\mathcal{L}(\mathcal{G}, \mathtt{BGP})$, $\mathcal{S}_r$ generates an add token $\mathtt{ATK}^{r*}$ with $\mathbf{C}_1$ and $\mathbf{C}_2$ and sends $\mathtt{ATK}^{r*}$ to $\mathcal{A}_r$. After that, $\mathcal{A}_r$ updates EMST^* by utilizing $\mathtt{ATK}^{r*}$. If op_j is a query, with $\mathcal{L}(\mathcal{G}, \mathtt{BGP})$, $\mathcal{S}_r$ generates an appropriate query token $\mathtt{QTK}^{r*}$ with $\mathbf{C}_1$ and $\mathbf{C}_2$ and sends $\mathtt{QTK}^{r*}$ to $\mathcal{A}_r$. After that, $\mathcal{A}_r$ searches EMST^* by utilizing $\mathtt{QTK}^{r*}$. After q round interactions, $\mathcal{A}$ produces a b bit as the output.

We say that Ω is adaptively $\mathcal{L}$-secure if for all probabilistic polynomial-time (PPT) semi-honest adversaries $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_q)$, there exists a simulator $\mathcal{S} = (\mathcal{S}_0, \ldots, \mathcal{S}_q)$ and a negligible function $negl(k)$ such that

$$\left| Pr\left[\boldsymbol{Real}^{\mathcal{A}}_{\Omega}(1^k) = 1\right] - Pr\left[\boldsymbol{Ideal}^{\mathcal{A}}_{\mathcal{L},\mathcal{S}}(1^k) = 1\right] \right| \leq negl(k).$$

Theorem 1. *If* $\mathcal{SS}$ *is secure and if* F *and* P *are pseudo-random, then* Ω *is adaptively* $\mathcal{L}-$*security.*

Proof. We create a simulator $\mathcal{S} = (\mathcal{S}_0, \ldots, \mathcal{S}_q)$ such that for an adversary $\mathcal{A} = (\mathcal{A}_0, \ldots, \mathcal{A}_q)$, and the outputs of $\mathbf{Real}^{\mathcal{A}}_{\Omega}(1^k)$ and $\mathbf{Ideal}^{\mathcal{A}}_{\mathcal{L},\mathcal{S}}(1^k)$ are computationally indistinguishable. The simulator and adversary work as follow.

$\mathcal{S}_0$: $\mathcal{S}_0$ simulates $(K_1, K_2) \leftarrow SS.share(0^k, 2, 2)$ and $\mathcal{F}_{2PC}$, and sends K_1 to $\mathbf{C}_1$ and K_2 to $\mathbf{C}_2$, respectively. It then generates an empty EMST.

$\mathcal{S}_r$: For $1 \leq r \leq q$, if $comm_r$ is an add, $\mathcal{S}_0$ simulates $\mathcal{F}_{2PC}$ to generates an add token $\mathtt{ATK}^{r*}$. For each element in $\mathtt{ATK}^{r*}$, $\mathcal{S}_0$ sets its value as a random string $\{0,1\}^*$, whose length is the same as the output of F, and permutes their positions by P. Then $\mathcal{S}_0$ sends $\mathtt{ATK}^{r*}$ to $\mathcal{A}$. If $comm_r$ is a query, $\mathcal{S}_0$ first checks whether the query $\mathtt{BGP}_r$ has appeared before with the search pattern in $\mathcal{L}(\mathcal{G}, \mathtt{BGP})$. If it has appeared before, $\mathcal{S}_0$ searches the access pattern in $\mathcal{L}(\mathcal{G}, \mathtt{BGP})$ and gets the same $\mathtt{QTK}^{r*}$ that has used before. If is has not has appeared before, $\mathcal{S}_0$ simulates $\mathcal{F}_{2PC}$ to generate a query token $\mathtt{QTK}^{r*}$ in the same way as generating atk. Then $\mathcal{S}_0$ sends $\mathtt{QTK}^{r*}$ to $\mathcal{A}$. With all but negligible probability, $\mathcal{A}$ cannot recover K_1 and K_2, thus $\mathcal{A}$ cannot distinguish the values in $\mathtt{ATK}^{r*}$ and $\mathtt{QTK}^{r*}$ from that in $\mathtt{ATK}^{r}$ and $\mathtt{QTK}^{r}$, respectively. This is because the $\mathtt{SS}$ is secure, and assuming that both F and P are pseudo-random.

Therefore, $\mathcal{A}$ cannot distinguish the output of $\mathbf{Ideal}^{\mathcal{A}}_{\mathcal{L},\mathcal{S}}(1^k)$ from $\mathbf{Real}^{\mathcal{A}}_{\Omega}(1^k)$.

□

4.4.2 Verifiability Analysis

Definition 7 (Query verifiability)**.** *We say a SPARQL query is verifiable if the success probability of any polynomial-time adversary $\mathcal{A}$ is negligible in the following experiment:*

- *$\mathcal{A}$ selects a set of RDF triples $\mathcal{T}$;*

- *The* EMST *generate algorithm constructs an* EMST *and its digest* EMST_{root} *based on* $\mathcal{T}$*;*

- $\mathcal{A}$ *produces result* $\mathtt{R}$ *and* VO_t *for the SPARQL query* Q*;*

- $\mathcal{A}$ *succeeds if one of the following results is true: 1)* $\mathtt{R}$ *includes an RDF triple which does not satisfy* Q *(**correctness**); 2) There exist an RDF triple which is not in* $\mathtt{R}$ *but satisfies* Q *(**completeness**); 3)* $\mathtt{R}$ *includes an RDF triple not from the latest DKG (**freshness**).*

Theorem 2. PISTIS *is verifiable with respect to **Definition 7** if the cryptographic hash function is a pseudo-random function, the cryptographic accumulator is secure under the q-SBDH assumption, and the computing power of malicious nodes is less than 51% of the blockchain network.*

Proof. We intuitively prove Theorem 2 by three cases, which represent proofs of soundness, completeness, and freshness.

Case 1: This case means a tampered or fake RDF triple t is returned, which does not satisfy the BGPs of Q. In this case, once t passed the verification of the user under the soundness in **Definition 7**, it means that the adversary can get two different triple pattern fragments with the same digest EMST_{root} of the ADS or the adversary can get two different set operation results with the same accumulator proof π. Case 2: This case means an RDF triple t that satisfies the BGPs of Q is missing from $\mathtt{R}$. In this case, if the returned result $\mathtt{R}$ can pass the verification of the user under the completeness in **Definition 7**, it means that the adversary can get a triple pattern fragment that does not contain some matching triples and has the same digest EMST_{root} of the ADS with the genuine fragment or the adversary can get an incomplete set operation result with the same accumulator proof π of the genuine set operation result. Case 3: This case means the result R involves an old RDF triple t that satisfies q but is not from the latest DKG. In this case, once t passed the verification of the user under the

freshness in **Definition 7**, it means that the adversary can get two different triple pattern fragments (i.e., a new and an old) with the same digest EMST_{root} of the ADS or the adversary can get two different set operation results with the same accumulator proof π.

However, all these three cases contradict two assumptions. The first is that the digest of the on-chain ADS EMST_{root} is generated by the cryptographic hash function, with all but negligible probability, the adversary can forge another fragment with the same hash value as the genuine fragment. The second assumption is the unforgeability for VSO, which has been proved to be held under the q-SBDH assumption [25]. □

4.5 Discussion

Custodians. The choice of custodians is an important consideration for our design. The security of Pistis relies on the custodians not colluding so they should be selected carefully. A certification authority (CA) like IdenTrust and DigiCert usually acts as a trusted third party to help users manage keys in web environment. For example, IdenTrust and DigiCert under the assumption that data owners would trust that they would not collude with each other in order to subvert the system and recover the private information of data owners. Also, we note that the number of custodians can be easily increased to any number by using MPC to replace 2PC.

Operations over BGPs. The above content describes how Pistis implements verifiable and ownership-preserving BGP-based SPARQL queries, while SPARQL also has some operations over BGPs, including property paths[1], named graphs[2], restrictions in the `FILTER` pattern, and most of the solution sequence modifiers[3] (e.g., `ORDER BY`, `OFFSET`, `DISTINCT`, `LIMIT`). Due to space limitations, we discuss the solutions and

[1]https://www.w3.org/TR/sparql11-property-paths/
[2]https://www.w3.org/2009/07/NamedGraph.html
[3]https://www.w3.org/TR/sparql11-query/#convertSolMod

limitations here to achieve their verifiability and take other advanced operations as our future work.

- For named graphs and property paths, a simple method is to concatenate them as a prefix to the RDF triple items and process them in the usual manner. However, this may cause the RDF triples to be too long and increase the storage overhead of EMST.
- For restrictions in the `FILTER` pattern, the storage provider can use a hash function, a partial path of EMST, VSO, or generate a general zero-knowledge proof to prove that the result satisfies its constraints. Specifically, a matching or no-matching constraint of two words can be verified by their hash values, a regular expression constraint can be verified by VSO and a Merkle proof provided by a partial path of EMST, and a range proof of whether the query result satisfies a certain size range can be generated through bulletproof [30]. However, in cases where data is encrypted, it is difficult for the storage provider to generate a range proof of the encrypted result. In this case, order-preserving encryption [7] is a potential solution.
- For the solution sequence modifiers, since the sorting criteria are given by the clients, they can verify the query results themselves without proof. A `GROUP BY` clause is used to group query results based on one or more variables, and can also be checked by clients themselves.

Data abuse and audit. Although Pistis provides protection against data breaches and tampering attacks, the primary concern resides in the abuse of data collection within DKG. Some authorized users regularly access DKG to obtain a significant amount of raw data, subsequently misusing it for purposes such as data mining and recommendation systems. To address this data misuse problem, we recommend that the DKG community limit the frequency of user access (e.g., by an owner-defined and dynamically adjusted policy), formulate rules through the consensus of participants

and use technical means (e.g., smart contract or attribute-based encryption (ABE) [20]) to implement it. Another real problem is that some data owners may encrypt some illegal data (e.g., drug dealing or pornography) and shares it in DKG, and others cannot judge its legitimacy from the ciphertext. Some works [73, 37] use data deduplication [139, 17] to solve this problem.

4.6 Evaluation

4.6.1 Experimental Setup

Hardware configuration. We run 8 data owner nodes, 16 blockchain nodes and 16 storage providers nodes on 16 64-bit Linux servers (Ubuntu 20.04) with Intel i9-11th CPU and 64GB memory. We set the bandwidth of connections between them to 20Mbps.

Implementation environment. A prototype of Pistis is implemented in Java, C++, Go and JavaScript. The blockchain module is implemented based on Go-Ethereum[4] and the storage module is implemented based on IPFS [19]. The prototype has a user-server architecture that is implemented based on Spring Boot framework[5] and the blockchain interfaces and requests are in the form of web3.js[6].

Cryptographic primitives. For all 2PCs, our prototype uses the ABY framework [44]. For MPC, our prototype uses JIFF library [9]. For secret sharing, our prototype instantiates a threshold secret sharing with Shamir secret sharing [102].

Datasets and benchmark. We evaluate the query performance of Pistis using the datasets and queries from largeRDFBench [101] benchmark, which is widely used by the DKG community. LargeRDFBench consists of 13 datasets and more than 1

[4]https://github.com/ethereum/go-ethereum
[5]https://spring.io/projects/spring-boot
[6]https://web3js.readthedocs.io/en/v1.5.2/

billion triples in total. The largeRDFBench queries in our evaluation include simple (S), complex (C), and large data (L) categories.

Metrics. We measure the following metrics of Pistis:

- Storage Cost (SC): The storage space size of the index.
- Token Generation Time (TGT): The amount of time it takes to generate an add token or query token.
- Item Add Time (IAT): The amount of time it takes to add a new item to Pistis, including adding to the storage network and blockchain.
- Query Execution Time (QET): The amount of time it takes to receive the full query results.
- Proof Generation Time (PGT): The amount of time it takes to generate the verification proof of query results.
- Verification Time and Verification Object Size (VT & VO): The time to verify query results and the proof size.

4.6.2 Experimental Results

Overall Comparison

A high-level overall comparison between our Pistis and other state-of-the-art DKG systems is shown in Table 4.1. In these three DKGs, Pistis is the only one that implements SPARQL queries with data integrity, query integrity and data ownership. Through the VO-SPARQL scheme, Pistis can give data owners control over data that is outsourced to a decentralized storage system. Moreover, Pistis also achieves double verifiability of raw data and query results in a decentralized byzantine environment. For the storage cost of the index on the DBPedia-Subset dataset with 42849609

Table 4.1: Overall comparison of three different systems

Schemes	Data integrity	Query verifiability	Data ownership	Storage cost of index (KB)
PIQNIC with PPBF [4]	✗	✗	✗	128425
Colchain [6]	✓	✗	✗	128425
Pistis	✓	✓	✓	70912
Pistis in plaintext	✓	✓	✗	13296

triples, the size of the index in **PIQNIC with PPBF** and **Colchain** is 128425 KBs, while on Pistis is 70912 KBs. The reason is that EMST compresses the index size by combining the same prefixes of keywords and Pistis needs a Merkle characteristic and prefix encryption to guarantee the query verifiability and data ownership. When data ownership is not considered, that is, only MST is used as the ADS of the system, the size of the index is 13296KB.

Performance Evaluation

Token generation time. We evaluate the performance of the `AddToken` and `QueryToken` protocols by generating a series of add tokens from the dataset and some different query tokens of different types of queries in the benchmark, and testing their token generation time (TGT) respectively. In Figure 4.6, we vary the size from 10 up to 1 million RDF triples and then test the add token generation time of them. The results demonstrate that generating an add token of a new triple is independent of the size of requests and takes about 24 milliseconds per triple. Besides, the results show that in the token generating process, the time required for the 2PC computation dominates the other tasks. Figure 4.7 shows the query token generation time of different types of KG query (i.e., S, C, and L queries in the benchmark, and setting 1000 items for each query). The results show that different types of KG queries have different query token generation time. Generating a simple (S) query token and a large data (L) query token takes about 96ms and 144ms respectively, and a complex (C) query

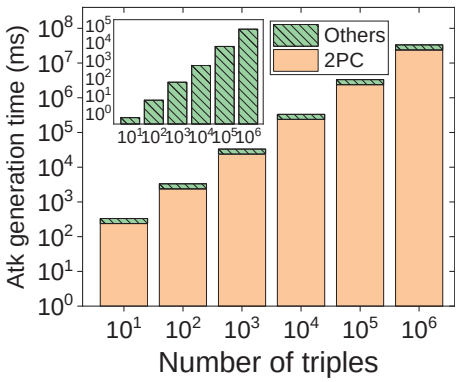


Figure 4.6: Add token generation time (x and y-axis in log scale.)

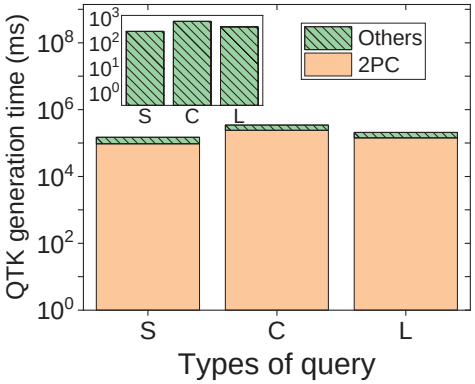


Figure 4.7: Query token generation time (y-axis in log scale.)

token needs about 240ms because it contains the largest number of triple patterns. Likewise, the 2PC computation also dominates the other tasks in the query token generation process.

Add time. We evaluate the performance of the `Add` protocol by storing a series of RDF triples to the storage providers and submitting their add tokens to the blockchain network. We choose the DBPedia-Subset dataset with 42849609 triples, store each of them to the IPFS network, and pack each into a transaction and submit it to Ethereum. The item add time (IAT) has two parts: 1) the synchronization time of IPFS nodes and 2) the confirmation time of Ethereum blocks. The time of storing one triple in the IPFS is 0.0187s (in **PIQNIC**, it is 0.0031s) and the block confirmation time is 6.803s (in **Colchain**, it is 5.44s). Obviously, the block confirmation time dominates the storing time and Pistis has a small insert overhead with respect to systems without encryption. We also conduct a set of comparative experiments to compare the block confirmation efficiency of the original Ethereum and Pistis, and their transaction throughput is 15.6 transactions/s for the original Ethereum and 14.9 transactions/s for Pistis. Therefore, the reason for the long add time is the low throughput of Ethereum itself and not due to our encryption algorithms.

Query execution time. To evaluate the performance of the `Query` and `Aggregate` protocols of Pistis, we compare the query execution time (QET) of Pistis with

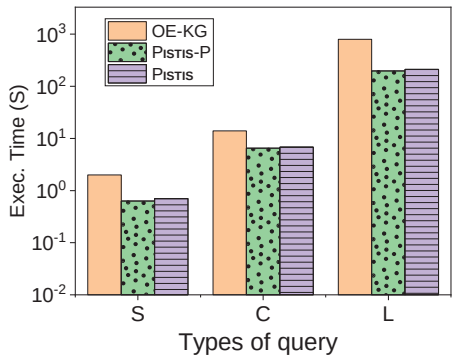


Figure 4.8: Query execution time of Pistis and original Ethereum-based KG (OE-KG) (y-axis in log scale).

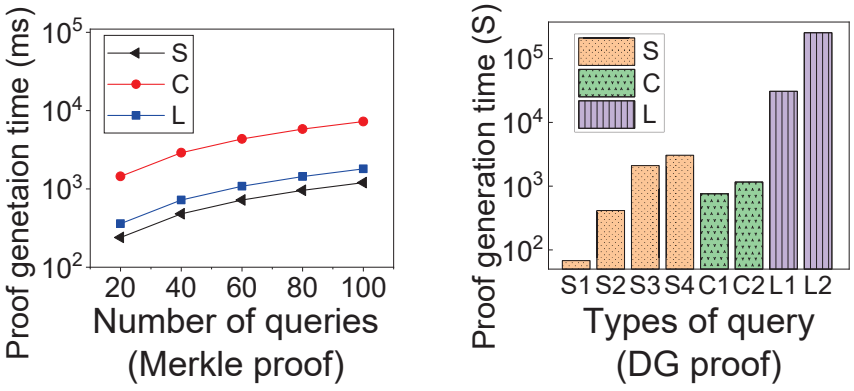


Figure 4.9: Proof generation time (y-axis in log scale).

two baseline systems, including an original Ethereum-based KG (OE-KG) that stores all encrypted RDF triples without any index and a variant Pistis system with a plaintext MPT (Pistis-P). Figure 4.8 shows the QET of them for different queries in the S, C and L query categories. For Pistis, the average QET for these three types of queries is 0.7s, 6.8s, and 212s, respectively. For Pistis-P, the average QET for these three types of queries is 0.63s, 6.5s, and 198s, respectively. For OE-KG, the average QET for these three types of queries is 2.1s, 14.2s, and 815s, respectively. By comparing the query performance of these three systems, we can find that Pistis and Pistis-P have better performance than OE-KG because both of them have indexes. Besides, compared with Pistis-P, Pistis achieves efficient query under ciphertext with small performance loss.

In addition to comparing with blockchain-based baselines, we also do the performance comparison of our Pistis with the state-of-the-art DKGs including **PIQNIC** and **Colchain** in terms of triple pattern query execution time on the DBPedia-Subset dataset, and the clients send 2000 triple pattern query requests for each system. As for the triple pattern query time, PIQNIC and Colchain need 5.081s and 6.773s to get the query results because they have the same index and Colchain has to find the triples in the blockchain transactions. Due to the performance overhead on the verification and encryption schemes, Pistis require more time to get the query results. The results show that Pistis introduces an additional execution time of 0.038s and 0.037s for each query compared with PIQNIC and ColChain due to more verification and encryption operations. Therefore, although Pistis requires more time to get the triple pattern query results than the existing systems, it can guarantee the data ownership and verifiability of query results and the extra time is tiny in real-world applications.

Verification cost. All the query results with their verification proofs are generated on the blockchain and storage providers, and need to be verified on the user side. We evaluate the performance of the `Verify` protocol by executing a series of SPARQL queries and testing their Proof Generation Time (PGT), Verification Object Size (VO) and time (VT). The experimental results are shown in Figure 4.9, Figure 4.10, and Figure 4.11. First, for the PGT, we test the Merkle proof generation time on the blockchain side with three different types of SPARQL queries (S, C and L), and test the aggregation proof generation time of different types of queries (i.e., 4 S queries, two C queries, and two L queries, setting 1000 items for each query). From Figure 4.9, we can see that the complex queries have the longest PGT for Merkle proof because they have the highest number of query-related fragments and each fragment has a Merkle proof. Also in Figure 4.9, the S and C queries take less time to generate aggregation proofs while the L queries have a longer PGT of data aggregation (DG). The reason is that the L queries have a higher number of intermediate results. Second, Figure 4.10

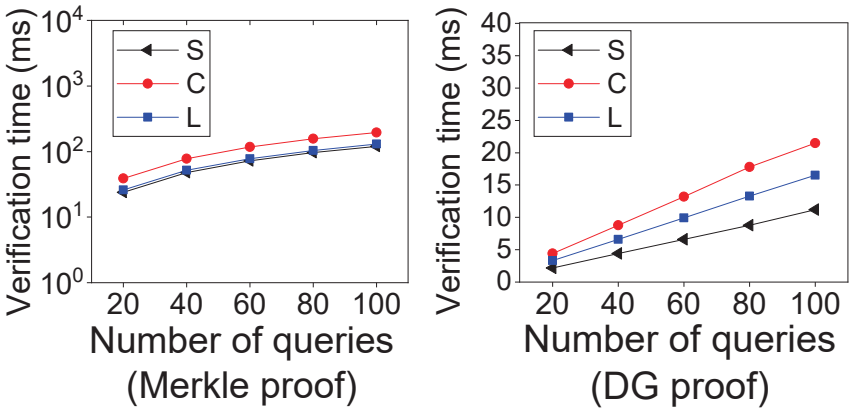


Figure 4.10: Verification time (left: y-axis in log scale).

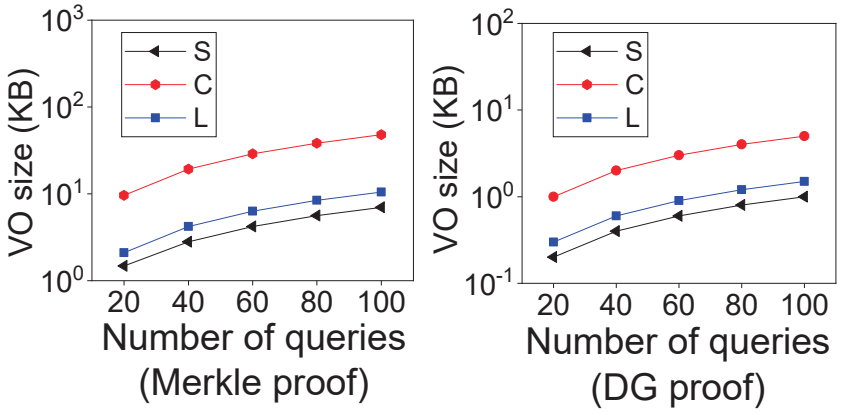


Figure 4.11: Verification object size (y-axis in log scale).

shows the verification time (VT) for Merkle proofs and aggregation proofs of different queries, which are the same, and C queries have slightly longer VT. For VT of Merkle proofs, the reason is that all queries need to calculate the same Merkle root hash while the complex queries need to calculate more fragments' hashes. From the results of VT for data aggregation proof, we can see that the verification is very fast and only related to the number of query-related fragments. Third, Figure 4.11 shows the size of VO of Merkle proofs and aggregation proofs on different queries. From these two figures, we can see that for both Merkle proof and data aggregation proof, the complex queries have the largest VO due to their highest number of fragments.

Chapter 5

VeriDKG: A Verifiable SPARQL Query Engine on Decentralized Knowledge Graph

The ability to decentralize knowledge graphs (KG) is important to exploit the full potential of the Semantic Web and realize the Web 3.0 vision. However, decentralization also renders KGs more prone to attacks with adverse effects on *data integrity* and *query verifiability*. While existing studies focus on ensuring data integrity, how to ensure query verifiability - thus guarding against incorrect, incomplete, or outdated query results - remains unsolved.

This chapter introduces VeriDKG, the first SPARQL query engine for decentralized knowledge graphs (DKG) that offers both data integrity and query verifiability guarantees. The core of VeriDKG is the RGB-Trie, a new blockchain-maintained authenticated data structure (ADS) facilitating correctness proofs for SPARQL query results. VeriDKG enables verifiability of subqueries by gathering global index information on subgraphs using the RGB-Trie, which is implemented as a new variant of the Merkle prefix tree with an RGB color model. To enable verifiability of the

final query result, the RGB-Trie is integrated with a cryptographic accumulator to support verifiable aggregation operations. A rigorous analysis of query verifiability in VeriDKG is presented, along with evidence from an extensive experimental study demonstrating its state-of-the-art query performance on the largeRDFbench benchmark.

5.1 Introduction

The Web 3.0 [43, 80, 53] envisions a future Web where the Semantic Web and the Web of Data play increasingly important roles [117, 103, 135, 71]. For the Semantic Web to meet the expectations, it is desirable, or necessary, to be able to support a decentralized knowledge graph (DKG) [3, 143, 5, 16]. For example, DBpedia[1] and Wikidata[2] provide free knowledge base with 9500 and 100 million linked data items contributed by communities of volunteers, empowering diverse applications from research to recommendation systems. As illustrated in Figure 5.1, a DKG is stored, managed, and queried using a decentralized infrastructure that facilitates the storage of subgraphs at multiple storage nodes. This infrastructure enables data owners to share their linked data as subgraphs of the global KG stored by the infrastructure through the public SPARQL endpoints [52, 29] or dereferenceable URIs it provides.

However, decentralized systems are more vulnerable to attacks and faults (e.g., *Byzantine fault* [35]) and therefore need means of ensuring *data integrity* and *query verifiability* in order to facilitate trustworthiness. In the context of a commercial peer-to-peer database, compromised nodes can manipulate transactions and forge data, posing risks to financial transactions and compromising business data. Data integrity ensures the storage nodes cannot tamper with their data [69, 27, 14], while query verifiability ensures that query results are complete, sound, and fresh [151, 153, 154, 129].

[1] https://www.dbpedia.org/
[2] https://www.wikidata.org/

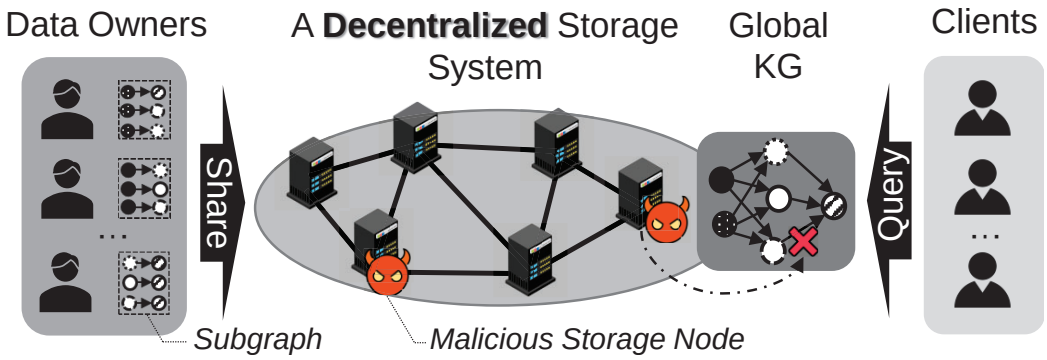


Figure 5.1: Illustration of a DKG system.

Without any measures taken, a malicious storage node in DKG can modify its data or return incorrect, incomplete, or outdated results that do not match user requests, violating data integrity and query verifiability. Taking Figure 5.1 as an example, a malicious storage node (comparable to a travel agency) can intentionally conceal relationships within the KG (suitable or low-cost options) and selectively provide incomplete results to clients (to prioritize its travel packages).

Unfortunately, most of the existing DKG infrastructures tend to prioritize query efficiency over trustworthiness [31, 4, 5, 16], although recent studies exist that focus on data integrity in DKGs [6, 105, 92]. For example, ColChain [6] enables data integrity in Byzantine environments by establishing storage nodes that maintain duplicate, immutable copies of subgraphs through blockchain consensus. However, these studies still assume that the subquery and communication in DKGs are trustworthy, which may not hold in real-world settings. Therefore, it is highly relevant to provide mechanisms that make it possible to verify that the subquery and the aggregation result are correct.

The standard approach to enable the verification of query results is to maintain an authenticated data structure (ADS) [108] that enables the detection of incorrect query results computed by an untrusted party on an outsourced database. While ADSs can be extended to decentralized infrastructures, existing ADSs are not designed for the querying of DKGs and cannot be applied readily to this setting, for two main reasons.

- **Decentralized data storage.** Due to the decentralized storage of DKG data, DKG query processing is done in two steps: executing subqueries to obtain intermediate results from individual storage nodes and aggregating the intermediate results to get final query results [6]. In contrast, existing ADS schemes assume that all data is stored in a single node when building and maintaining an ADS.

- **Semantic richness.** Because of its semantic richness, KG data that is both diverse and exhibits complex relationships is challenging for existing ADSs to capture. Therefore, existing ADSs can support neither verification of local query processing nor verification of global query processing where local results are aggregated.

Therefore, in this work, we design a new ADS-based SPARQL [97, 18] query verification scheme and enable the use of blockchain for its generation and management to ensure data integrity and query verifiability. To contend with the decentralized data storage, our main idea is to design an ADS that can accommodate essential metadata of subgraphs stored on different nodes to achieve a global index, thus relaxing the requirement for a node to hold all data to build an ADS. To contend with the semantic richness, we ensure that it is possible to embed the semantic information required for KG query into the ADS. Specifically, we use keyword prefixes and an RGB color model to represent all semantic information with minimal cost. With the resulting ADS, DKG can be verified in a divide-and-conquer manner and the data integrity can also be ensured by blockchain. Specifically, for the step of finding local subgraphs, we implement the global index as a new variant of the Merkle tree that can provide the location of any subgraph and give verification proof. For the step of aggregation, we provide a cryptographic accumulator [91, 33] and combine it with the ADS, thus allowing nodes to perform verifiable aggregation operations on intermediate results from different nodes.

Our contributions can be summarized as follows.

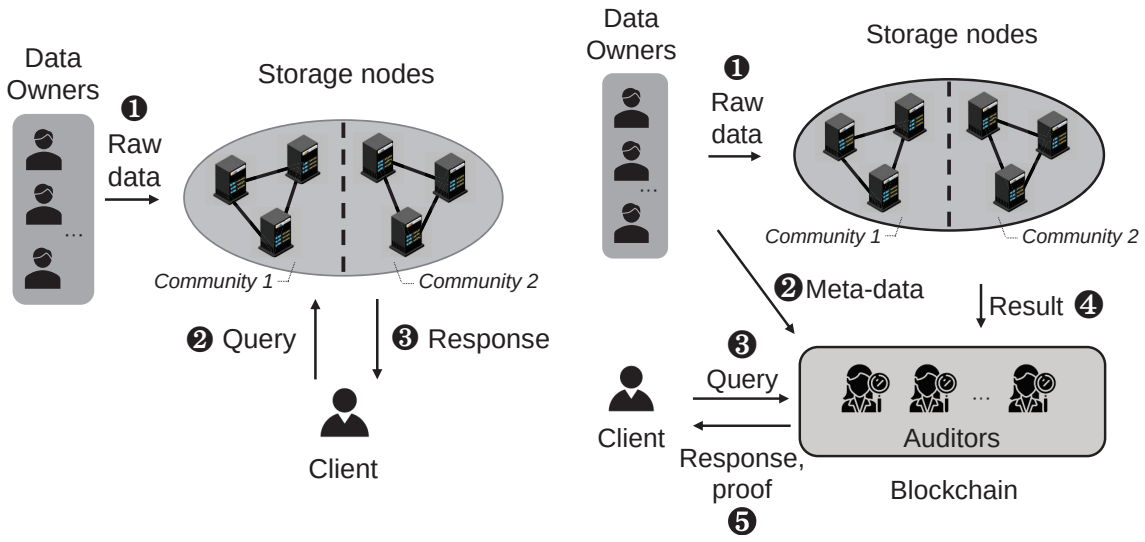
- We propose VeriDKG, a novel DKG system that enables SPARQL query verifiability. To the best of our knowledge, this is the first of its kind.
- We propose a verification framework for DKGs that relies on a novel ADS, the RGB-Trie, to process SPARQL queries in a divide-and-conquer manner with correctness proofs.
- We implement the RGB-Trie as a Merkle prefix tree with an RGB color model to enable the capture of the necessary semantic information and combine it with a cryptographic accumulator technique to support verifiable aggregation on intermediate results from multiple storage nodes.
- We provide a rigorous security analysis and report on experiments with a prototype of VeriDKG. The results demonstrate that the system can achieve state-of-the-art query performance on the largeRDFbench benchmark while supporting data integrity and query verifiability.

5.2 The VeriDKG Model

5.2.1 System Model & Threat Model

Basic Model. In a DKG system, there are three types of participants.

Data owners generate raw RDF data as triple fragments and share their data in the DKG. Due to limited resources, the data owners outsource their data to storage nodes. **Storage nodes** offer DKG storage and data management services, divided into KG communities, each responsible for specific triple fragments. They can collaborate across communities when executing SPARQL queries and facilitate addressing



(a) A typical DKG (b) Our VeriDKG

Figure 5.2: Comparison between the existing DKG and our VeriDKG.

relevant nodes for a requested RDF triple. **Clients** query the global KG by issuing SPARQL requests to the storage nodes.

As shown in Figure 5.2a, data owners outsource their data to the storage nodes (❶), and a client can send its query request to any storage node (❷) and get the query result (❸). Most DKG systems have a data replication mechanism [5, 4] to ensure query services normally run when some storage nodes encounter failure.

Threat Model. Different from existing DKGs [5, 4] that assume that storage nodes are trustworthy, in VeriDKG, the storage nodes are not trusted. Each storage node may forge or tamper with query results or return outdated information for various reasons, such as program glitches, security vulnerabilities, and commercial interests. We assume that the data owners and clients are trusted. (A discussion for malicious data owners is provided in subsection 5.4.2.) A similar threat model can be found in outsourced databases [87, 127, 154].

System Model. Our VeriDKG introduces a new role of participants to build a verifiable SPARQL query engine on DKG as follows.

Auditors compose a blockchain system as a trust anchor in the DKG. The blockchain guarantees that the auditors can collectively build a public and immutable ledger via consensus. We assume that the proportion of malicious auditors will not exceed the fault threshold of blockchains (e.g., 1/2 in PoW or 1/3 in PBFT).

5.2.2 System Workflow

As illustrated in Figure 5.2b, the data owners, storage nodes, auditors and clients interact in VeriDKG as follows.

Phase 1: ***Data Outsourcing.*** Each data owner outsources its RDF data to the storage nodes (❶). At the same time, the data owner also calculates the hash of each RDF triple, and proposes a transaction containing the metadata (i.e., the hash, index information, and address of the triple in the storage network) for each triple.

Phase 2: ***ADS Generation.*** The data owner then submits the proposed transaction with metadata to the blockchain (❷). To commit the transaction, all the auditors run a consensus to build an ADS with an index function for all RDF data stored on the storage node network. The ADS maintained in the blockchain will be updated based on each newly committed transaction. The details will be described in subsection 5.3.2.

Phase 3: ***Query Processing.*** A client can issue a SPARQL query to any auditor (❸). The auditor converts the query to a set of triple patterns, uses the ADS to search for the triple pattern fragments of each triple pattern (namely triple pattern queries), and gets the triple pattern fragments from storage nodes (❹). All the triple pattern fragments are aggregated on the auditor for the final results. The auditor also generates proofs using the ADS, discussed in subsection 5.3.3.

Phase 4: ***Query Verification.*** The final query results and their corresponding proofs are sent to the client from the auditor, and the client verifies the query results

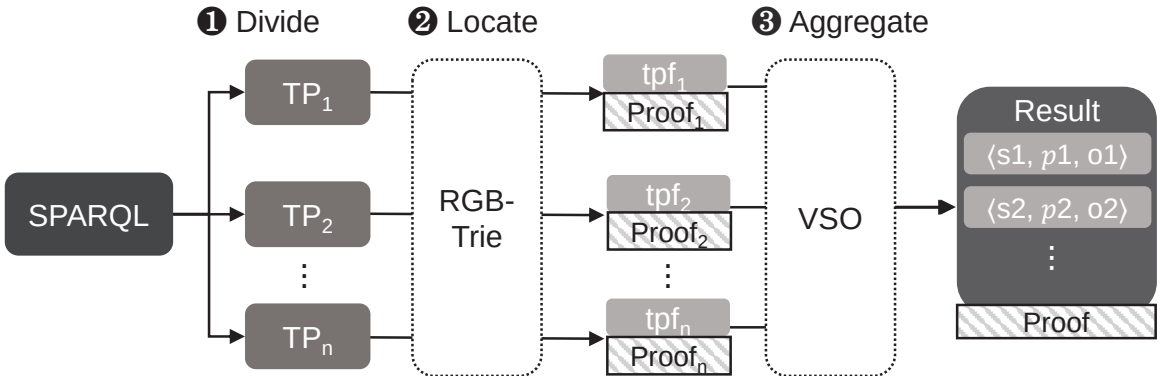


Figure 5.3: The process of verifiable SPARQL query execution in VeriDKG.

(❺), which is discussed in subsection 5.3.3.

5.2.3 Goals

The query verification in **Phase 4** should guarantee that the query results returned from the storage nodes satisfy three security criteria: 1) **Soundness**: None of the RDF triples returned as results have been tampered with and all of them satisfy the SPARQL query conditions; 2) **Completeness**: No valid result (e.g., RDF triples and their members) is missing from the query results; 3) **Freshness**: The query results are based on the latest version of the DKG.

5.2.4 Roadmap

As mentioned before, executing a SPARQL query in DKGs involves three steps: 1) dividing the SPARQL query into multiple triple patterns, 2) locating the relevant triple pattern fragments matching each triple pattern, and 3) aggregating these fragments for final results. To enable verifiable SPARQL queries in VeriDKG, we introduce a novel ADS for verifiable triple pattern queries and utilize a verifiable set operation-based approach for aggregation. In particular, as shown in Figure 5.3, in the locating process, VeriDKG employs the RGB-Trie, a Merkle tree variant, to find relevant

intermediate results (triple pattern fragments) for all triple patterns alongside their proofs. In the aggregating process, most aggregation operators are converted to some set operations, and a storage node in VeriDKG can perform verifiable set operations on the intermediate results, and returns the final query results with a verification proof to clients. In the following, we will introduce the design of RGB-Trie in Section subsection 5.3.2 and the entire query process in Section subsection 5.3.3.

5.3 Methodology

5.3.1 Strawman

To support verifiable triple pattern queries, we first describe a strawman ADS for **Phase 2** only based on a Merkle prefix tree (MPT) like Merkle Patricia Tree [125], an ADS for verifiable prefix-based keyword queries adopted by many blockchain systems. In the strawman, the auditors can build an MPT on the blockchain, which treats the value given in the triple pattern as a keyword. The internal nodes of the MPT stores the index information (i.e., keywords) of RDF triples, and the MPT's leaf nodes point triple fragments, each containing some RDF triples with the same keyword. For example, for KG $\mathcal{G}$ in Table 2.1, we let $p_1 = aa$, $p_2 = ab$, and $p_3 = ba$, the MPT to query $\mathcal{G}$ can be shown in Figure 5.4. For three triple patterns $\langle ?s, p_1, ?o\rangle$, $\langle ?s, p_2, ?o\rangle$, and $\langle ?s, p_3, ?o\rangle$, f_1, f_2 and f_3 are their corresponding triple pattern fragments through the MPT. Given a triple pattern fragment f_1' from a storage node, to verify whether it corresponds to triple pattern $\langle ?s, p_1, ?o\rangle$, a client can recover a Merkle root ($h6'$) based on the hash of f_1' ($h1'$) and a Merkle proof ($h2$ and $h5$), and then verify whether $h6$ and $h6'$ are the same.

Unfortunately, the strawman falls short of enabling verifiable triple pattern queries due to two limitations as follows.

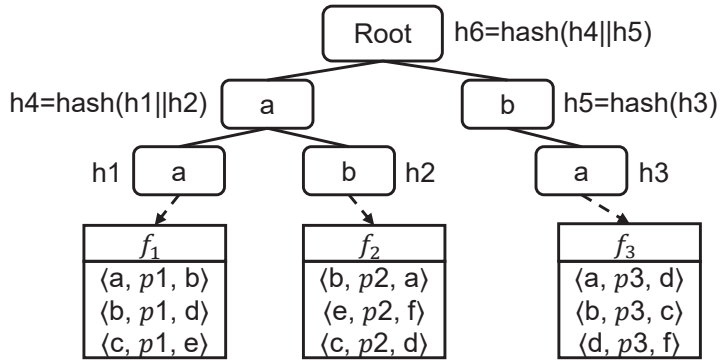


Figure 5.4: An example of an MPT in the strawman system.

First, it only supports verifiable queries for limited triple patterns, i.e., those where only one of the three features in triples is given. For example, the MPT in Figure 5.4 supports verifiable queries for triple patterns with a given predicate. It cannot support verifiable queries for triple patterns with a given subject or object.

Second, it cannot support verifiable aggregation of intermediate results, but a SPARQL query requires performing aggregation operations (e.g., UNION or JOIN) on triplet pattern fragments according to section 2.1. For example, if a SPARQL query requires the join of f_1 and f_3 on the subject, the MPT in Figure 5.4 cannot provide any proof.

Therefore, we propose a new ADS called *RGB-Trie*, a variant of MPT with two new characteristics. 1) It includes an RGB color model on MPT nodes for verifiable queries for any triple patterns, and 2) it integrates an accumulated value design with the MPT for verifiable set operations for aggregation on triple patterns.

5.3.2 RGB-Trie: ADS for VeriDKG

Trie Structure

As shown in Figure 5.5, RGB-Trie comprises four types of nodes, i.e., a root node, branch nodes, extension nodes, and leaf nodes, which are described as follows. In the figure, an RDF dataset with three RDF triples is inserted into an RGB-Trie, and the

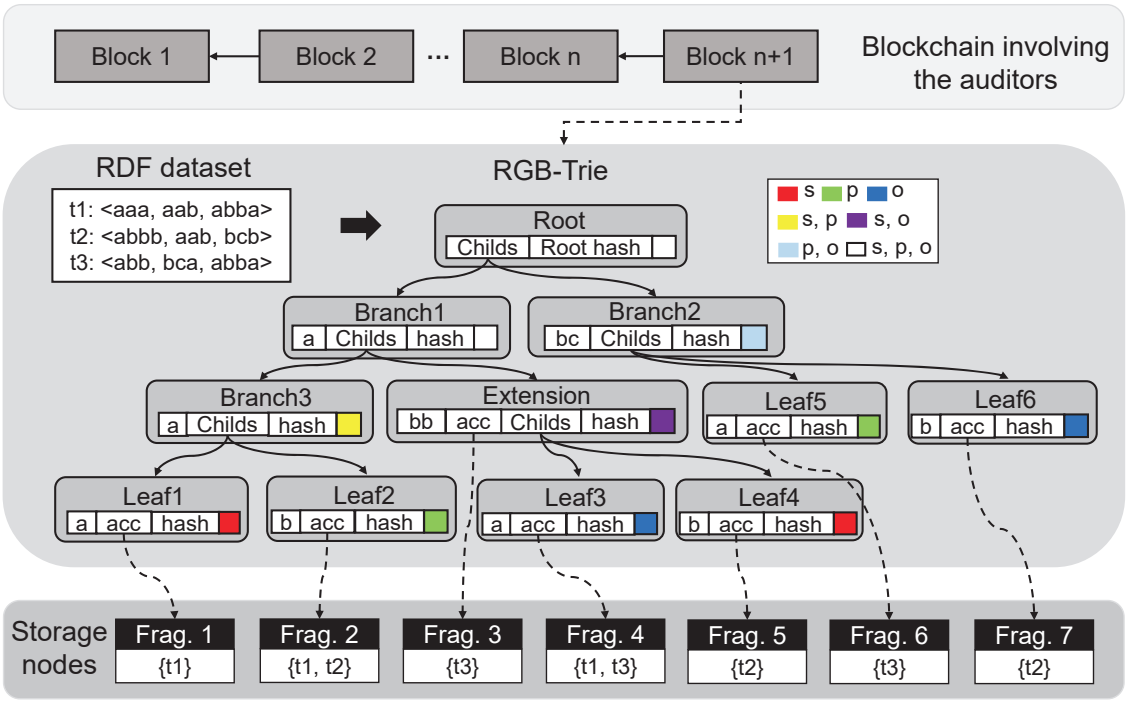


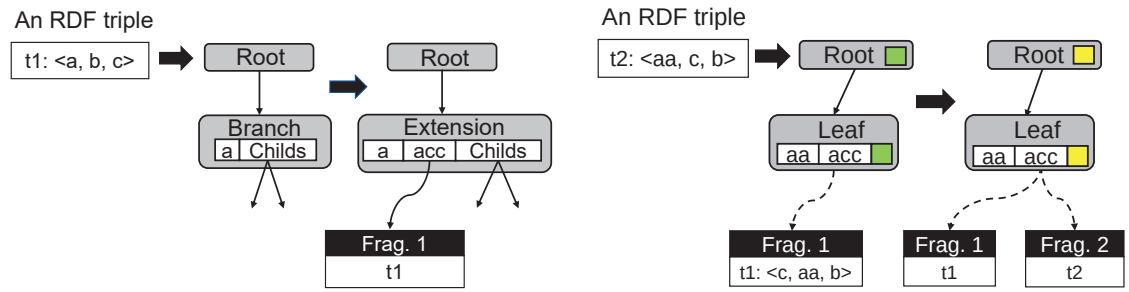
Figure 5.5: The structure of RGB-Trie.

features in the triples may have all or part of the same prefix. It's worth noting that the prefix for each entity in an RDF triple can encompass various elements such as property paths[3], named graphs[4], and other components found within the KG.

Root node. The top layer of RGB-Trie is a root node, which is a Merkle root maintaining a consistent snapshot of global KG. Based on the characteristics of MPT, the hash value inside a root node changes when any node in RGB-Trie is modified.

Branch/Extension node. The middle layer of RGB-Trie includes branch nodes and extension nodes. Each branch node connects its predecessor (parent node) and successors (child node) through its own value properties. It stores the common prefix of its child nodes and has at least two children. An extension node is a special branch node and can represent the termination of a query path. It has some pointers to point to triple pattern fragments. A branch node can be transformed into an extension node

[3]https://www.w3.org/TR/sparql11-property-paths/
[4]https://www.w3.org/2009/07/NamedGraph.html



(a) The process of converting a branch node to an extension node.

(b) The process of changing the node color.

Figure 5.6: Two cases of RGB-Trie node operation (Frag. means triple fragment.

by inserting some RDF triples with a keyword whose matching path ends at this node to RGB-Trie. We illustrate the process of converting a branch node to an extension node in 5.6a. When an RDF triple with a subject a is inserted into the RGB-Trie, a branch node with the common prefix a is transformed to an extension node that has a pointer to point the inserted RDF triple.

Leaf node. RGB-Trie's bottom layer comprises leaf nodes, each with pointers to address sets for triplet pattern fragments and their accumulated values. Additionally, each leaf node holds hash values for the referenced fragments. These attributes are pivotal in enabling RGB-Trie to furnish verification proofs for SPARQL queries (refer to subsection 5.3.3).

The design above creates an index for triple fragments linked to an input word. However, in KG, each triple has three features (subject, predicate, object), and each index fragment corresponds to just one feature. This necessitates three separate tries, tripling the storage and computation overhead as all three must update simultaneously with each transaction. To address this, we merge these tries into one using an RGB color model as follows.

Definition 8 (RGB Color Model)**.** *Given a node n in RGB-Trie, its color field is defined as red, green or blue if its search target feature is subject s, predicate p or*

object o, respectively. If n points to two different features, e.g., $s \wedge p$, $s \wedge o$, or $p \wedge o$, its color field is defined as yellow, magenta, or cyan, respectively. The color field is set to white when n has simultaneous access to the three features.

Example. *An example of node color change is shown in 5.6b. The color of a leaf node is green because the fragment it points to is a triple pattern fragment for the triple pattern $\langle ?s, aa, ?o \rangle$ and its index value is **aa**. If a new RDF triple with a subject **aa** is inserted into this leaf node, its color is changed from green to yellow because yellow is a mixed color of green (for predicate) and red (for subject). After the node color change process is over, the leaf node will store two different pointers, two hash values, and two accumulated values.*

After adding the RGB color model to MPT, the RGB-Trie can convert any form of triple pattern query into keyword queries with different color combinations, which is described in subsection 5.3.3. We also give a cardinal Rule of *color mixing* for the RGB color model in RGB-Trie to improve its query performance. Details are shown in section 5.3.2.

Operations of RGB-Trie

At the beginning of VeriDKG, an empty RGB-Trie is stored in the genesis block (the first blockchain block). Along with each new RDF data outsourced to the storage nodes (Figure 5.2b-❶), the auditors change the RGB-Trie according to the newly committed transaction about the new RDF data (Figure 5.2b-❷). There are three kinds of operations, i.e., *insert*, *update*, and *delete*, introduced as follows.

Insert Operation. After receiving a transaction with an address points to an RDF triple $\langle s, p, o \rangle$, the auditors add three items $[s, red]$, $[p, green]$, $[o, blue]$ to an item list L. For each item $item_i$ in L, the auditors search for an insertion point $node_p$ of the root node of RGB-Trie by traversing its child nodes.

For each item $item_i$, if the auditors find $node_p$, it inserts $item_i$ to $node_p$ through a recursive insertion algorithm. If they cannot find $node_p$, auditors create a new child node $node_{new}$ of the root node of RGB-Trie and insert $item_i$ to $node_{new}$. In the insert processing, if at any time the next character to be matched in the item does not match the characters stored in all children of an inserted node, the RGB-Trie will create a new child node of the inserted node to store the remaining unmatched part of the item. If $node_p$ has more than one character (i.e., slices of character, which is a variable-length character array), and the unmatched part of the item is only partly the same as the slices, $node_p$ will split (i.e., keep the matched part of the slices in the node and create two new child nodes to store the remaining part of $item_i$ and the remaining part of the slices).

Example. *Consider Figure 5.5 as an example. Suppose* $t3 = \langle abb, bca, abba \rangle$ *is a new triple. When it is inserted into the tree, the right child node of node* $Branch1$ *is converted to an extension node with a new triple fragment* $Frag.3$*. The node* $Branch2$ *will only keep prefix* **bc** *and split with two new child nodes, including a leaf node* $Leaf6$ *with* $Frag.7$ *and a leaf node* $Leaf5$ *with prefix* **a** *and a new triple fragment* $Frag.6$*. And* $t3$ *will also be inserted into* $Frag.4$*.*

Finally, after the insertion process, the auditors update RGB-Trie's hashes and colors and change it to a new state. Different nodes have different methods to update their hash value. Let $||$ be the concatenation operator of multiple values, $hash(\cdot)$ is the cryptographic hash function, and the hash update processing of different types of nodes in RGB-Trie are as follows:

For a leaf node n_l, the hashing process is:

- f_n = the triple pattern fragment(s) that n_l points;
- c_n = the keyword related index content in n_l, i.e., a segment of the keyword.
- $h_n = hash(c_n||hash(f_n))$, it is the hash value of n_l.

For an extension node n_e, denotes its child nodes is $\{cn_1, cn_2 \dots \ cn_k\}$, the hashing process is:

- f_n = the triple pattern fragment(s) that n_e points;
- c_n = the keyword related index content in n_e, i.e., a segment of the keyword.
- $ch_n = hash(cn_1||cn_2 \dots ||cn_k)$
- $h_n = hash(c_n||hash(f_n)||ch_n)$, it is the hash value of n_e.

For a branch node n_b, denotes its child nodes is $\{cn_1, cn_2 \dots cn_k\}$, the hashing process is:

- c_n = the keyword related index content in n_b, i.e., a segment of the keyword.
- $ch_n = hash(cn_1||cn_2 \dots ||cn_k)$
- $h_n = hash(c_n||ch_n)$, it is the hash value of n_b.

Besides, RGB-Trie updates the color of all nodes on this path at the same time. It follows the RGB additive color mixing rule. The color of a node is determined by a mixture of the index field in which it stores its content and the color of its children, and the specific descriptions are shown as follows:

Basic color mixing rule. Under this rule, each node in the RGB-Trie has a triple (R, G, B), and each element of the triple is a binary value. We use 1 to indicate that the color exists, and 0 to indicate that the color does not exist. For example, in Figure 5.7, the color of the root node is white and represented as $(1, 1, 1)$, because all child nodes of the root node contain all items of the RDF triple. To update the color of a non-leaf node on the RGB-Trie, the node needs to OR the color triples of all its child nodes.

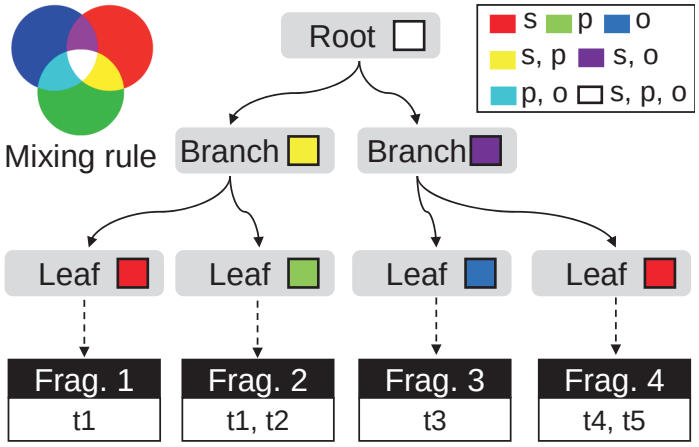


Figure 5.7: The color mixing rule in RGB-Trie.

Gradient color mixing rule. Under this rule, each node in the RGB-Trie has a triple (R, G, B), and each element of the triple is an 8 bits value. We use $(255, 0, 0)$ to represent color red, $(0, 255, 0)$ to represent color green and $(0, 0, 255)$ to represent color blue. To update the color of a non-leaf node on the RGB-Trie, it needs to calculate the proportion of different items of RDF triples (i.e., s, p, and o) in its child nodes. For example, in Figure 5.7, the color triple of the root node can be calculated as $(255 * (3/5), 255 * (2/5), 255 * (1/5))$, because there are 5 RDF triples in the whole KG, including 3 pointed by subject, 2 pointed by predicate and 1 pointed by object.

The two color mixing rules offer distinct advantages and drawbacks. The basic rule simplifies color calculations and ensures precise and comprehensive query results. In contrast, the gradient color mixing rule empowers clients to decide query termination based on color proportions, enhancing efficiency but potentially reducing recall. Importantly, the recall reduction doesn't compromise query verifiability, as it aligns with client preferences. We will assess the query performance of these rules in section 5.5.

Update & Delete. When a data owner wants to update or delete an RDF triple stored in storage nodes, it asks the storage nodes to update or delete that triple, and sends a new transaction consisting of the triple and a field marked for update or deletion to the auditors. When the auditors receive the transaction, they will

update or delete the triple item and update the nodes in the RGB-Trie to a new state. Under these two operations, the update process of the RGB-Trie is similar to the insert operation.

Cost Analysis

Here, we provide the time and space complexities associated with the RGB-Trie. **Time complexity:** When inserting an item with a word of length L and a color, the auditor traverses the RGB-Trie recursively level by level, which incurs L comparisons in the worst case. Updating RGB-Trie's hashes and colors upon an insertion involves reversing the order of traversal and L executions. Thus, insertion has a time complexity $O(L)$, with L being the length of the word to be inserted. Similarly, update, deletion and querying also have time complexity $O(L)$.

Space complexity: The space consumption of an RGB-Trie depends on its total number of nodes, which is related to its depth and number of leaf and non-leaf nodes. An RGB-Trie is constructed from a dataset of RDF triples. We denote the number of distinct items among all RDF triples in the input dataset by N, and we denote the cardinality of the character set of the input dataset by D. We can then determine an upper bound on the number of children of each non-leaf node. We start by observing that in an RGB-Trie, each item is stored either in a non-leaf node or in a leaf node. Further, each non-leaf node has at least 2 and at most D child nodes (assuming $D \geq 2$). Next, in the worst case, an RGB-Trie is a fully balanced binary tree with all items being stored in leaf nodes. Thus, the leaf layer has N nodes, each non-leaf layer L has 2^L nodes, and the RGB-Trie has $\lceil log_2 N \rceil$ layers. The maximum total number of nodes that an RGB-Trie can contain is therefore $1+2+2^2+\ldots+2^{log_2 N-1}+N = 2N-1$. Consequently, the space complexity of the RGB-Trie is $O(N)$.

5.3.3 Query Processing and Verification

We achieve a verifiable SPARQL query process in two steps, i.e., *verifiable triple pattern query* and *verifiable fragments aggregation*, as follows.

Verifiable Triple Pattern Query. A triple pattern query aims to search for a triple pattern fragment that matches a given triple pattern. algorithm 4 describes the verifiable triple pattern query process through RGB-Trie. First, the auditor searches in RGB-Trie to find the triple pattern fragments that match the input triple pattern (Lines 1-8). Each given value of the triple pattern, i.e., $item_i$, visits the RGB-Trie to query related triple fragments with their colors independently. RGB-Trie uses the depth-first search method to find a path in RGB-Trie that matches $item_i$, and the path ends at a leaf node or an extension node. Then, if $item_i$ matches successfully, RGB-Trie will add the fragment pointed by the end-point node and its Merkle proof to a verification object (VO) (Lines 9-16). Otherwise, if $item_i$ matches failed, RGB-Trie will add a set of nodes with their Merkle proofs, including the nodes in the partially matched path and all child nodes of the last node in this path, to the VO (Lines 17-18). Finally, the VO is returned to the client.

After receiving the results and proof, the client can verify them by comparing the recovered root node's hash that it calculates with the root hash of RGB-Trie which is kept in the current block header. If they are equal, the verification succeeds.

Example. *To search a triple pattern* $tp_i = \langle aaa, ?p, aab \rangle$ *in 5.6a, an auditor should extract two items* ***aaa*** *and* ***aab*** *with color red and blue, and later search the RGB-Trie to find the related triple pattern fragments. The auditor will get the results* $R_{aaa} = \{t_1\}$*, and* $R_{aab} = \emptyset$*, because no nodes in RGB-Trie have both a prefix content of* ***aab*** *and possess the blue color. Finally, the auditor takes the intersection of two intermediate results* R_{aaa} *and* R_{aab} *and gets the final result* $\emptyset$*. Thus, the auditor will return the final result of* tp_i *(i.e.,* $\emptyset$*), the intermediate result* $\{t_1\}$ *with its Merkle proof, and the non-existing proof of* R_{aab} *to the client.*

Algorithm 4: Verifiable triple pattern query

```
1  Function Search for fragments (r,tp_i):
       Input  : RGB-Trie root node r, a triple patten tp_i
       Output: Triple pattern fragments f, matching path p
2      foreach item_i in tp_i do
3          f_i, p_i = r.nodeSearch(item_i)
4          if f_i != null then
5              add f_i to f
6          else
7              add p_i to p
8      return f, p;
9  Function Get proof (node with f_i,f_i ∈ f, p):
10     foreach node with f_i do
11         add hash(f_i) to VO;
12         while node.parent != null do
13             add nodeInfo to VO
14             foreach node_i ∈ node.parent.child do
15                 add node_i.hash to VO
16             node ← node.parent
17     if p != null then
18         add p, Merkle proof of p to VO
19     return VO
```

Verifiable Fragments Aggregation. After the triple pattern query, a data aggregation process needs to be executed on an auditor that collects the intermediate results (i.e., triple pattern fragments) to get the final results. A SPARQL begins with an operation `SELECT` followed by a subset of variables in the triple patterns, and then a keyword `WHERE` followed by a set of triple patterns (i.e., a BGP) connected by some graph patterns (e.g., `JOIN`, `LEFT JOIN`, `FILTER`) and operations (e.g., `OPTIONAL`, `UNION`, `ORDER BY`). It is worth noting that most graph patterns and operations can be converted to set operations between subgraphs. For example, for a `JOIN` graph pattern involving the same features of two triple pattern fragments, an auditor treats the features of these two fragments as two sets and constructs a proof for their intersection through VSO. In the following, we will describe the detailed execution process of the verifiable fragments aggregation on the graph patterns and operations that can

Algorithm 5: Verifiable data aggregation

```
1  Function Get result and proof (f, tp, pk):
       Input  : Triple fragments f, triple pattern list tp, RGB-Trie R, public key pk
       Output: Query result S, verification proof π
2      foreach f_i ∈ f do
3          foreach tp_j ∈ tp do
4              tpf_ij ← R.search (f_i, tp_i) and add tpf_ij into tpf_j
5          add tpf_j into tpf
6      S ← aggregate (all tpf_i ∈ tpf)
7      π ← prove (tpf_i, pk)
8      return {S, π}
9  Function Verify proof (R, π, X*):
       Input  : Triple pattern fragments tpf, query result S, verification proof
                {X*, π}
       Output: verification result
10     foreach tpf_i ∈ tpf do
11         add acc(tpf_i) into ACC
12     result ← VerifyProof(ACC, π)
13     return result
```

be converted to set operations and give the solutions for other operations.

In VeriDKG, the auditor is responsible for aggregating the intermediate results from the triple pattern query transfers the aggregation operations in SPARQL query request into accumulator set operations (e.g., intersection, union, complement, and difference) and uses the following algorithm to generate verification proofs of the aggregating operation.

algorithm 5 describes the process of verifiable fragments aggregation. First, for each triple pattern, an auditor uses the RGB-Trie to search for its related triple pattern fragments (Lines 2-5). After all triple pattern fragments are found, the auditor aggregates them to get the final query result (Line 6). To generate the verification proof of the final result, the auditor uses the accumulated values of all triple pattern fragments and generates a verification proof π (Line 7). The client that receives the query result uses the proof π and the accumulated values to verify the query result locally (Lines 9-13).

Example. *Figure 5.8 gives an example of fragments aggregation, the SPARQL query selects s fields from two fragments through two triple patterns including a p-fixed triple pattern and an o-fixed triple pattern. The generation proof is* $< acc(X^1), acc(X^2), \pi, [1, 1, 0], [1, 0, 1] >$.

To calculate the accumulated value of each triple pattern fragment, all the auditors collaboratively generate a pair of keys (i.e., a public key and a private key), share the public key in a public way, and share the private key in a private way (e.g., secret sharing [102]). All the accumulated values are stored on the extension nodes and leaf nodes of RGB-Trie.

Verification of Non-set Operations. For other operations that cannot be converted to set operations, the auditor can let clients verify the results themselves or use other cryptographic tools to generate proofs for their results. These operations include some restrictions in the `FILTER` pattern and most of the solution sequence modifiers (e.g., `ORDER BY`, `OFFSET`, `DISTINCT`, `LIMIT`). For restrictions in the `FILTER` pattern, the auditor can use a hash function, a partial path of RGB-trie, VSO, or generate a general zero-knowledge proof to prove that the result satisfies its constraints. Specifically, a matching or no-matching constraint of two words can be verified by their hash values, a regular expression constraint can be verified by VSO and a Merkle proof provided by a partial path of RGB-Trie, and a range proof of whether the query result satisfies a certain size range can be generated through bulletproof [30]. For the solution sequence modifiers, since the sorting criteria are given by the clients, they can verify the query results themselves without proof. A `GROUP BY` clause is used to group query results based on one or more variables, and can also be checked by clients themselves.

Time-window Query. It is straightforward to extend the query algorithm to support time-window multi-version queries because the state of RGB-Trie at each moment is recorded by its root hash which is stored in the block header with a timestamp. Therefore, to query historical data in DKG, a client only needs to send a SPARQL

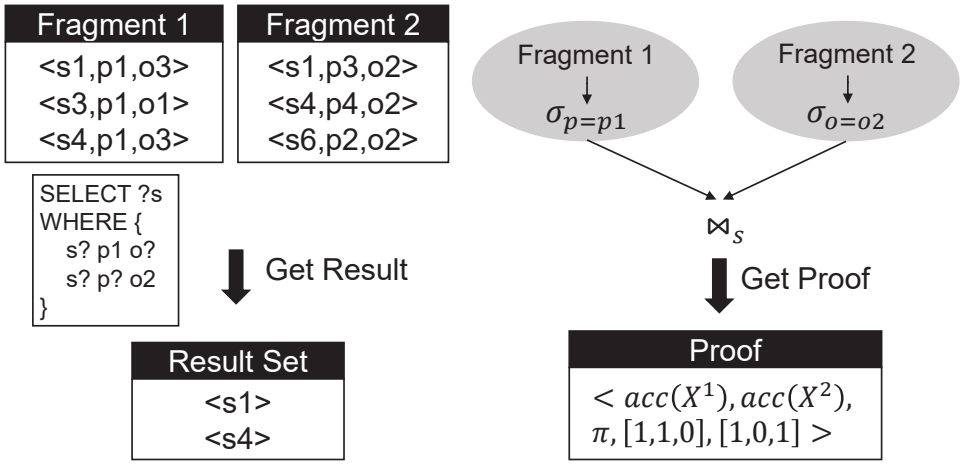


Figure 5.8: Example for proof generation of two triple fragments 1 and 2, respectively. (σ is an operator to select triples from a triple pattern and $\bowtie$ is an operator to join fragments based on a specified column.

query request with a given time period to an auditor, and the auditor uses the previous version RGB-Trie for the time-window queries. For verification, the client can use the Merkle root corresponding to the time window stored locally to verify the query result.

5.4 Analysis

5.4.1 Verifiability Analysis

VeriDKG enables verifiable SPARQL query in DKGs, which ensures soundness, completeness, and freshness as defined in subsection 5.2.3. Next, similar to the common security definition of verifiable query [154, 153], we describe the formal definition of our SPARQL query's security as follows.

Definition 9 (Query verifiability)**.** *A SPARQL query is verifiable if the success probability of any polynomial-time adversary $\mathcal{A}$ is negligible in the following experiment:*

For a SPARQL query Q, $\mathcal{A}$ is picked as the auditor for executing the triple pattern

*queries and fragments aggregation, and $\mathcal{A}$ produces result R and proof VO_t for Q. $\mathcal{A}$ succeeds if one of the following results is true: 1) R includes an RDF triple which does not satisfy Q (**correctness**); 2) There exist an RDF triple which is not in R but satisfies Q (**completeness**); 3) R includes an RDF triple not from the latest DKG (**freshness**).*

Theorem 3. VeriDKG *is verifiable with respect to **Definition 9** if the hash function is a pseudo-random function, the accumulator is secure under the q-SBDH assumption, and the proportion of malicious auditors will not exceed the fault threshold of blockchains.*

Proof. We intuitively prove Theorem 3 by three cases, which represent proofs of soundness, completeness, and freshness.

Case 1: This case means a tampered or fake RDF triple t is returned, which does not satisfy the BGPs (i.e., a set of triple patterns) of Q. In this case, if t passes client verification following the soundness in **Definition 9**, it implies that the auditor can obtain two distinct triple pattern fragments sharing the same digest RGB_{root} in the on-chain ADS or two distinct set operation results with the same accumulator proof π. Case 2: This case means an RDF triple t that satisfies the BGPs of Q is missing from R. In this case, if the returned result R verifies with the client under the completeness criterion in **Definition 9**, it suggests that the auditor can acquire a triple pattern fragment lacking some matching triples but sharing the same digest RGB_{root} as the genuine fragment or an incomplete set operation result with the same accumulator proof π as the genuine result. Case 3: This case means the result R involves an old RDF triple t that satisfies q but is not from the latest DKG. In this case, once t passed the verification of the client under the freshness in **Definition 9**, it means that the auditor can get two different triple pattern fragments (i.e., a new and an old) with the same digest RGB_{root} of the on-chain ADS or the auditor can get two different set operation results with the same accumulator proof π.

However, all these three cases contradict two assumptions. First, the on-chain ADS digest RGB_{root} is generated by a cryptographic hash function, making it nearly impossible for the auditor to forge another fragment with the same hash value as the genuine one. Second, the unforgeability of verifiable set operations, proven to hold under the q-SBDH assumption [25]. A special case occurs when the auditor returns a null result to the client while the system indeed has the matched query result. This case also contradicts the first assumption as the auditor must provide non-existence proof.

□

5.4.2 Discussion

Storage Optimization of RGB-Trie. Since RGB-Trie needs to store the index information of all triples, the size of RGB-Trie increases fast as the KG data is continuously added. Storing the latest RGB-Trie in every block will be expensive. Thus, in VeriDKG, auditors only need to update part of the RGB-Trie in a new block with its transactions, and the internal nodes of the two RGB-Tries in two blocks are mostly the same. Based on the node pointers design in [93], we use a node pointer structure to link the same path between multiple RGB-Tries in multiple blocks.

Limitations and Potential Workaround. When deploying VeriDKG in real-world environments, some limitations and potential workarounds must be considered. First, while VeriDKG effectively protects against malicious behavior between storage nodes, it does not address the possibility of malicious data owners introducing fake or junk data-a challenge seen in widely used AI applications. To mitigate this concern, we assume the integration of a content moderation mechanism (e.g., as proposed in existing studies [67, 113, 56]) to detect malicious data owners. Second, there are several resource-intensive tasks in VeriDKG, such as data storage, blockchain consensus, and query execution. However, incentivizing participants to effectively

contribute resources can be complicated and requires careful consideration of the interests of different participants and ways of ensuring fairness. Third, VeriDKG introduces auditors that are responsible for maintaining trust anchors in the DKG. They require higher hardware specifications compared to existing DKG nodes. For example, the Ethereum backend requires a full node with at least 16GB of RAM and a 1TB SSD. Hardware requirements for blockchain node deployment vary widely depending on factors such as network, participation level, and use case. Careful selection of settings by application deployers is critical.

Real-world Deployment. The deployment of VeriDKG involves several main steps. Blockchain nodes, acting as auditors, are initially set up with robust hardware configurations, including substantial RAM and storage capacity. Users access the DKG through responsive interfaces from a variety of devices. Storage nodes, equipped with suitable hardware and data distribution software, host and share the KG data. Additionally, scalability, maintenance strategies, and regulatory compliance are considered to ensure seamless real-world operation. VeriDKG is particularly suited to high-trust data scenarios, such as those found in healthcare and finance, and offers trusted data to improve AI fidelity.

Graph Model Extension of VeriDKG. In addition to RDF triple-based KGs, some recent studies focus on graph model-based knowledge graph management. VeriDKG can be extended to such graph-based DKGs by managing RDF data in a graph database. This involves two steps: storing RDF triples as a graph, where entities are represented as nodes and relations are represented as edges, and converting SPARQL queries into graph patterns, such as paths or trees. Taking Neo4j[5] as an example, we can use tools such as rdf2neo [28] to convert RDF triples into a Neo4j graph and to convert SPARQL queries into Cypher queries.

[5]https://neo4j.com/

5.5 Evaluation

5.5.1 Implementation

A prototype of VeriDKG is implemented in Java, Go and JavaScript. In the prototype, the blockchain involving the auditors is implemented based on Go-Ethereum [49] and the decentralized storage system is implemented based on IPFS [19]. The prototype has a user-server architecture that is implemented based on Spring Boot framework [114] and the blockchain interfaces and requests are in the form of web3.js [50]. For the proposed ADS, the level of the tree is set to 32, and the cryptographic hash function is SHA-256. The triple fragments stored on storage nodes are separate HDT files [51], which allows the storage nodes to efficiently execute triple patterns. For VSO, our prototype uses library named ate-pairing [62].

5.5.2 Experimental Setup

Hardware Configuration. We run 16 VeriDKG auditor nodes and 16 storage nodes on 32 64-bit Linux servers (Ubuntu 20.04) with Intel i9-11th CPU and 64GB memory. All nodes are run on separate machines, and we set the bandwidth of connections to 20Mbps.

Baseline. Three state-of-the-art DKGs are considered as three baselines. (1) A DKG with a locational index in every peer [5] (**P2P-LI**). (2) VeriDKG without the blockchain architecture and Merkle tree characteristic, which is a DKG with an untrusted RGB-Trie (**P2P-RGB**). (3) A sharding blockchain-based DKG in ColChain (**Colchain**) [6]. **Colchain** adopt the same consensus of Ethereum and hardware configuration. Besides, the maximum number of shards in **Colchain** is 16, and each shard has 8 nodes which are put into the docker containers.

Datasets and Benchmark. We evaluate the query performance of VeriDKG using

six real-world datasets that represent different scenarios and using queries from largeRDFBench [101] benchmark, which is used widely in the Semantic Web community. The datasets include DBPedia-Subset (42,849,609 triples), GeoNames (107,950,085 triples), Jamendo (1,049,647 triples), Linked MDB (6,147,996 triples), DrugBank (517,023 triples), and Semantic Web Dog Food (103,595 triples). The largeRDFBench queries in our evaluation include simple (S), complex (C), and large data (L) categories. Moreover, we develop a variant of simple queries named time-window simple queries to study the time-window query in VeriDKG.

Metrics. We measure the following metrics of VeriDKG: 1) *On-chain Storage Cost (OSC)*: the storage space size of transactions in the blockchain, 2) *Transaction Throughput (TT)*: the number of committed transactions per second, 3) *Triple Pattern Query Time (TPQT)*: the amount of time to receive the triple pattern query results, 4) *Query Execution Time (QET)*: the amount of time to receive the full query results, 5) *Number of Exchanged Messages and Transfer Bytes (NEM & NTB)*: the number of messages exchanged and transferred bytes between nodes, 6) *Proof Generation Time (PGT)*: the amount of time to generate the verification proof of query results, and 7) *Verification Time and Object Size (VT & VOS)*: the amount of time to verify query results and the proof size.

5.5.3 Experimental Results

Overall Comparison

Table 5.1 provides an overview of the performance of VeriDKG, comparing its query verifiability, index storage cost, and triple pattern query execution time with those of three baseline systems on the six datasets. Here, the clients send 2,000 triple pattern query requests to each system. As shown in Table 5.1, only VeriDKG enables verifiable SPARQL query results, ensuring that the malicious storage nodes cannot tamper with query results. Considering the index storage cost, the index size of **P2P-**

Table 5.1: Overall comparison of four different systems

Schemes	Query Verifiability	Storage cost of index (KB)	Triple pattern query time (s/ms)
P2P-LI [3]	✗	317982	7.392/3.696
P2P-RGB	✗	5647	36.990/18.495
Colchain [4]	✗	317982	8.374/4.187
VeriDKG	✓	9193	51.116/25.558

LI and **Colchain** is 317,982 KBs, while **P2P-RGB** and VeriDKG only need 5,647 KBs and 9,193 KBs, respectively. The reasons are that the RGB-Trie compresses the index by combining the same prefixes of keywords and that VeriDKG needs a Merkle characteristic to enable query verifiability. As for the triple pattern query time, **P2P-LI** and **BC-SC** need 7.392s (3.696ms per query) and 8.374s (4.187ms per query) to get the query results because they have the same index, and **Colchain** has to find the triples in the blockchain transactions. **P2P-RGB** and VeriDKG respectively require 36.990s (18.495ms per query) and 51.116s (25.558ms per query) to get the query results because they have the same index, and VeriDKG need more time to generate the verification proofs.

In summary, VeriDKG is the only one DKG that achieved verifiable SPARQL query and has a smaller index size than the locational index-based systems. Although it requires more time to get the triple pattern query results than the existing systems, it can guarantee the verifiability of query results and the extra time is tiny in real-world applications.

On-chain Cost

To alleviate the on-chain storage pressure, VeriDKG stores the raw data on storage nodes and only keep the metadata on blockchain. We randomly sample triples from the 6 datasets to determine the storage cost (i.e., OSC) of VeriDKG and the other two blockchain-based baseline systems. For **Colchain**, it has 16 shards and each

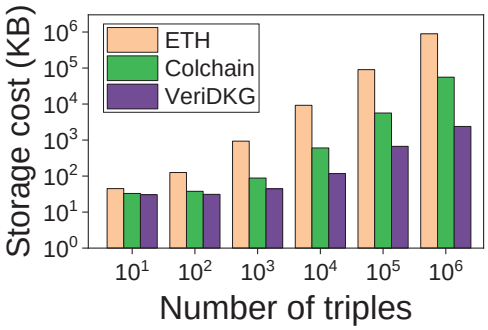


Figure 5.9: On-chain storage cost (y-axis in log scale.)

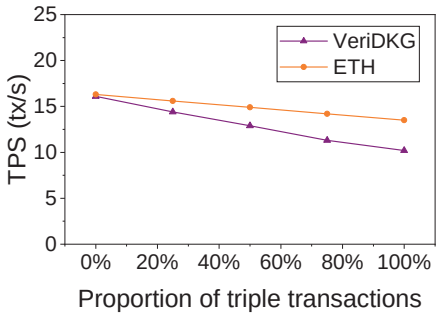
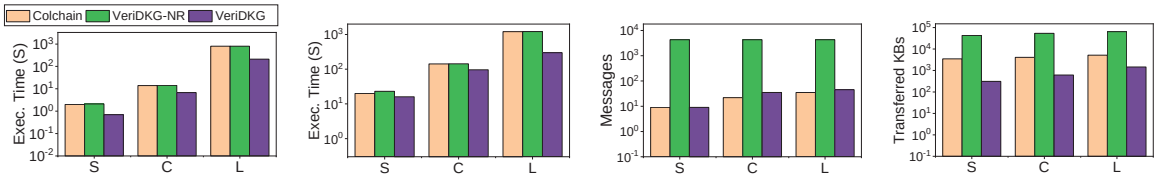


Figure 5.10: Transaction throughput.

node only stores one shard. Figure 5.9 shows the average storage size of each node in VeriDKG and the baselines.



(a) QET. (b) QET (time - window). (c) NEM. (d) NTB.

Figure 5.11: Query performance (y-asix in log scale).

As shown in Figure 5.9, by comparing to the original Ethereum blockchain scheme **ETH**, **Colchain** reduces the on-chain storage of each node by about 80%, and maintains this ratio as the amount of data grows. Compared to the two baselines, the storage savings in VeriDKG can increase to over 99% as the amount of data increases. It is because our design only keeps a succinct ADS on-chain, and most of the growing data is stored off-chain. The ADS only needs to update some hash values of the index information. We also compare experimentally the storage consumption of the RGB-Trie with and without optimization. For the VeriDKG system without storage optimization, the total size of the RGB-Trie is 13,296MB. For the VeriDKG system with storage optimization, the total size of the RGB-Trie is 2,241MB. Thus, after optimizing the storage of the RGB-Trie, the storage cost can be reduced by more than 80%.

Moreover, we evaluate the transaction throughput (i.e., TT) of **ETH** and VeriDKG with different proportions of triple transactions, and the results are shown in Figure 5.10. The result shows a downtrend of TT in both systems when the proportion of triple transactions in all transactions increases. In particular, VeriDKG has a bigger trend in decline, which means that the update of the RGB-Trie has a negative impact on TT. However, VeriDKG still maintains more than 10 transactions per second in the worst case, which is acceptable.

Verifiable Query Performance

Figure 5.11a shows the query execution time (QET) for different SPARQL queries in **Colchain**, in VeriDKG without the RGB-Trie (**VeriDKG-NR**), and in VeriDKG. Note that there is no global index in **Colchain**. Similar query performance is observed of **VeriDKG-NR** and **Colchain**, and VeriDKG is fastest for all three queries because RGB-Trie can efficiently find query-relevant fragments, which significantly reduces the search space. Figure 5.11b shows the QET for time-window queries. VeriDKG has the shortest QET among the three systems because, apart from the above reasons, in real-time SPARQL queries, the auditors in VeriDKG only need to backtrack all block headers to search for a certain previous RGB-Trie state by a given timestamp, instead of tracing back the historical records of all query results. Figure 5.11c shows the average number of exchanged messages (NEM) between nodes in the three systems. VeriDKG has the same number of messages as **Colchain** because both two only need to transfer a fixed number of fragments between nodes. Further, **VeriDKG-NR** needs to transmit larger messages to get the result because it lacks an RGB-Trie and needs to download all fragments from all storage nodes. Figure 5.11d shows the number of transferred KBs (NTB) in the three systems. The nodes in VeriDKG transmit the least amount of data for each query because VeriDKG only needs to transfer the query related triple pattern fragments to the auditor, which are more compact than the entire RDF dataset.

Table 5.2: Performance of RGB-Trie under different rules. It shows triple pattern query time (TPQT) and recall per 1000 queries for basic and gradient color mixing rule with different proportion of colors required for early termination.

Rule	TPQT	Recall
Basic	51.2s	1
Gradient (5%)	47.5s	0.952
Gradient (10%)	40.7s	0.816
Gradient (20%)	27.8s	0.621
Gradient (30%)	22.3s	0.366

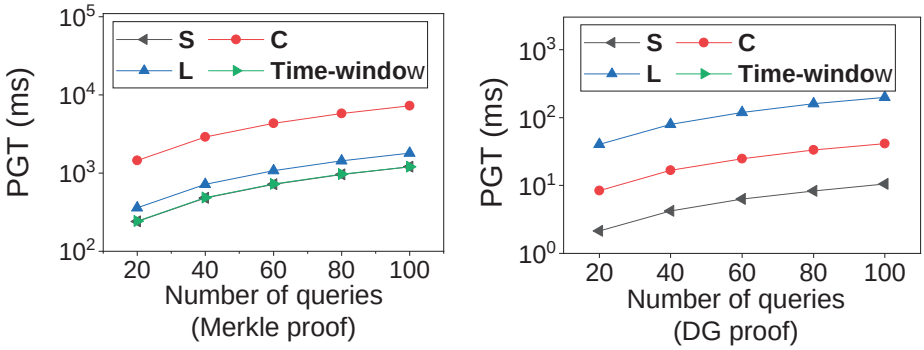


Figure 5.12: Proof generation time (y-axis in log scale).

Comparison of different color mixing rules. As we mentioned in section 5.3.2, the gradient color mixing rule can improve the query performance of RGB-Trie while decrease its recall (i.e., the fraction of relevant RDF triples that are returned). Thus we test the triple pattern query time (TPQT) and recall of 1000 triple pattern query requests in VeriDKG under different color mixing rules. Table 5.2 shows the results. The results imply that the gradient color mixing rule can reduce the time of triple pattern query VeriDKG, while relaxing the early termination conditions results in higher query efficiency at the expense of lower recall rates.

Verification Cost

All the query results with their verification proofs are generated on an auditor and need to be verified on the client side. Therefore, the proof generation time on the auditor, the size of proof, and the verification time on the client are very important

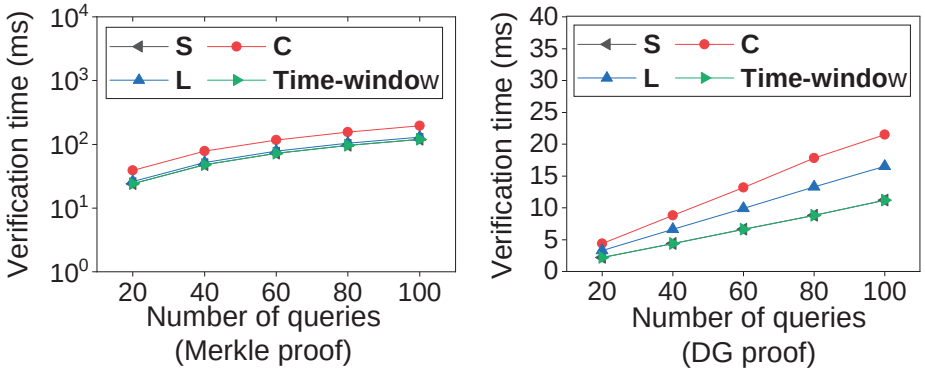


Figure 5.13: Verification time (left: y-axis in log scale).

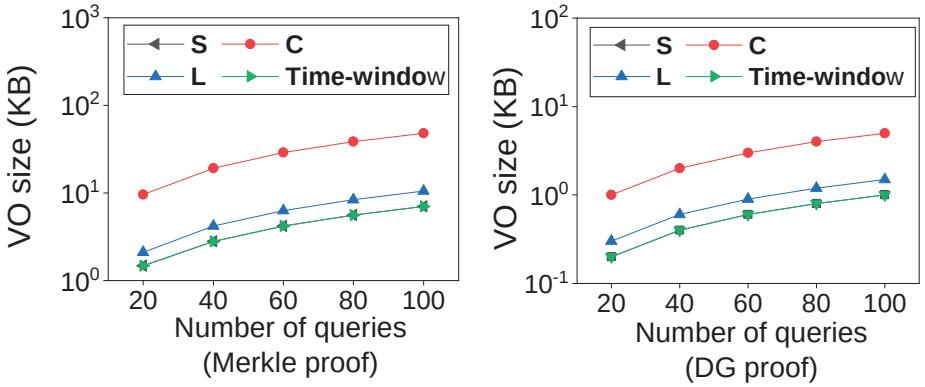


Figure 5.14: Verification object size (y-axis in log scale).

for the availability of VeriDKG. A proof consists of two parts: a Merkle proof and a data aggregation (DG) proof. We evaluate the verification costs of these two proofs as follows.

Figure 5.12 shows the proof generation time (PGT) of the auditor for four SPARQL query types (S, C, L, and time-window S query). It shows that the complex queries have the longest PGT because they have the highest number of query-related fragments, each of which needs a Merkle proof. On the other hand, the large data queries have the longest PGT for data aggregation proof. Because most of their query-related fragments exceed those of the other queries, they need the most time to generate the accumulated values.

Figure 5.13 shows the verification time (VT) for proofs of different SPARQL queries,

which are the same in Merkle proof, and complex queries have slightly longer Merkle proof VT. This is because all queries need to calculate the same Merkle root hash, while the complex queries need to calculate hashes of more fragments. From the VT results for the data aggregation proof, we can see that the verification is fast and is related only to the number of query-related fragments. Figure 5.14 shows the VO size for the different query types. It shows that for both Merkle and data aggregation proofs, the complex queries have the largest VOS because they have the highest number of fragments.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Decentralized knowledge graphs (DKGs) represent a paradigm shift in how data is created, shared and queried across the web, leveraging the principles of decentralization to enhance interoperability, scalability, and availability. At the core, these graphs utilize semantic technologies to create a web of interconnected data, where relationships and meanings are explicitly defined, enabling machines to understand and process the information much like humans do. This semantic web approach facilitates more intelligent data integration, search, and analysis, promising to transform various domains by making information more accessible and useful. However, this decentralization and semantic richness introduce significant security risks. The open and distributed nature of these graphs can expose data to unauthorized access, manipulation, and misuse, challenging traditional security models. Ensuring data integrity, privacy, and access control in such an environment requires innovative security mechanisms tailored to the unique characteristics of decentralized, semantically-rich data networks.

To address these challenges, this report is mainly composed of three following parts.

- We have comprehensively investigated the poisoning attacks of FKGE. Our attacks can accurately inject fake relations into the victim's model, even if their local KG data and some model parameters are unknown. We demonstrate the effectiveness and practicality of four real-world datasets and four KGE models. The dynamic poisoning attack achieves an average MRR of 0.67 and Hits@10 of 0.88 on the poisoned triples in four datasets on four different KGE models. In particular, the success rate of the attack is 100% on the WN18RR dataset. Furthermore, the experimental results also demonstrate that the FKGE's original task performance is not significantly affected by our attacks. To mitigate this attack, we explore two potential defense mechanisms that shed light on improving the current practice of FKGE and point to several promising research directions, such as decentralized and verifiable KGE.

- We present Pistis, an end-to-end encrypted and collaboratively query-verifiable DKG with a new cryptographic scheme. The scheme relies on a novel ADS and a key-aggregate cryptographic primitive to query the multi-owner KG data in a verifiable and ownership-preserving manner. Security analysis with an idea/real-world paradigm and experimental evaluations prove the security and availability of our system. In particular, Pistis achieves new functionalities at an overhead of microsecond-level computation time, and kilobyte-level communication costs for a SPARQL query.

- We present VeriDKG, which supports verifiable SPARQL query in Web 3.0. We design a new ADS called RGB-Trie for verifiable subgraph locating and combined the tree with cryptographic accumulators for verifiable aggregation for intermediate results. We also do extensive experiments to test the performance of VeriDKG, and the results show that VeriDKG implements verifiable SPARQL with a competitive query performance compared with the state-of-the-art. Compared with the leading-edge DKGs, VeriDKG reduces the index storage overhead by 97%.

6.2 Future Work

Decentralized knowledge graphs (DKGs) is a promising way to sharing knowledge in Web 3.0, and blockchain-based methods effectively solve the security problems in it. However, since the definition of web 3.0 is not very clear, the future web 3.0 may have various scenarios other than DKGs. For example, with the gradual landing of the metaverse, web 3.0 may become one of the network infrastructures of the virtual world. In this environment, the way of knowledge sharing will also become different. To achieve a more secure and efficient decentralized knowledge sharing in the future web 3.0, there are at least the following three directions that I can study in the future:

First, in the metaverse based on web 3.0 in the future, there may be more than one blockchain, and different blockchain has different architecture, consensus, and security assumption. Designing a decentralized knowledge sharing system in this multi-chain scenario may have some challenges. For example, the interoperability between different blockchains can make it difficult to share and access knowledge across different networks. This requires the development of standards and protocols that enable seamless communication and data exchange between disparate blockchains.

Second, investigating the integration of decentralized knowledge graphs with large language models (LLMs) presents a promising research direction. This integration can leverage the semantic understanding and generative capabilities of LLMs to enhance the accessibility and usability of DKGs, facilitating more intuitive query mechanisms and knowledge extraction methods. Exploring techniques to mitigate potential biases and misinformation within LLM-generated content, while ensuring the scalability and privacy of such integrations, would be pivotal.

Third, decentralized identity (DID) is an important concept in Web 3.0-based metaverses, as it enables users to have greater control over their personal data and online identity. DID refers to the use of decentralized technologies such as blockchain to create and manage digital identities that are owned and controlled by the user rather

than a centralized authority. However, there are several technical challenges associated with implementing decentralized identity (DID) in Web 3.0-based metaverses, such as the balance of privacy and supervision and the integration of virtual and real identities.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proc. of the 2016 ACM SIGSAC conference on computer and communications security*, pages 308–318, 2016.

[2] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. A survey and experimental comparison of distributed sparql engines for very large rdf data. *Proceedings of the VLDB Endowment*, 10(13):2049–2060, 2017.

[3] Ibrahim Abdelaziz, Essam Mansour, Mourad Ouzzani, Ashraf Aboulnaga, and Panos Kalnis. Query optimizations over decentralized rdf graphs. In *Proc. of 2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 139–142, 2017.

[4] Christian Aebeloe, Gabriela Montoya, and Katja Hose. A decentralized architecture for sharing and querying semantic data. In *Proc. of the European Semantic Web Conference (ESWC)*, pages 3–18, 2019.

[5] Christian Aebeloe, Gabriela Montoya, and Katja Hose. Decentralized indexing over a network of rdf peers. In *Proc. of the International Semantic Web Conference (ISWC)*, pages 3–20, 2019.

[6] Christian Aebeloe, Gabriela Montoya, and Katja Hose. Colchain: Collaborative linked data networks. In *Proc. of the Web Conference (WWW)*, pages 1385–1396, 2021.

[7] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 563–574, 2004.

[8] Ali M Al-Khouri et al. Data ownership: who owns "my data". *International Journal of Management & Information Technology*, 2(1):1–8, 2012.

[9] Kinan Dak Albab, Rawane Issa, Andrei Lapets, Peter Flockhart, Lucy Qin, and Ira Globus-Harris. Tutorial: Deploying secure multi-party computation on the web using jiff. *2019 IEEE Cybersecurity Development (SecDev)*, pages 3–3, 2019.

[10] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. A survey of rdf stores & sparql engines for querying knowledge graphs. *The VLDB Journal*, pages 1–26, 2022.

[11] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proc. of the EuroSys Conference*, EuroSys, 2018.

[12] S Josephin Arulmozhi, K Praveenkumar, and G Vinayagamoorthi. BITCOIN IN INDIA: A DEEP DOWN SUMMARY. *Advance and Innovative Research*, page 28, 2019.

[13] Balaji Arun and Binoy Ravindran. Scalable byzantine fault tolerance via partial decentralization. *Proc. of the VLDB Endowment*, 15(9):1739–1752, 2022.

[14] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proc. of ACM conference on Computer and communications security (CCS)*, pages 598–609, 2007.

[15] Farah Atif, Ola El Khatib, and Djellel Difallah. Beamqa: Multi-hop knowledge graph question answering with sequence-to-sequence prediction and beam search. In *Proc. of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 781–790, 2023.

[16] Amr Azzam, Christian Aebeloe, Gabriela Montoya, Ilkcan Keles, Axel Polleres, and Katja Hose. Wisekg: Balanced access to web knowledge graphs. In *Proc. of the Web Conference (WWW)*, pages 1422–1434, 2021.

[17] Andrei Bacs, Saidgani Musaev, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. {DUPEFS}: Leaking data over the network with filesystem deduplication side channels. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 281–296, 2022.

[18] Debayan Banerjee, Pranav Ajit Nair, Jivat Neet Kaur, Ricardo Usbeck, and Chris Biemann. Modern baselines for sparql semantic parsing. In *Proc. of ACM SIGIR*, pages 2260–2265, 2022.

[19] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[20] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *2007 IEEE symposium on security and privacy (SP'07)*, pages 321–334. IEEE, 2007.

[21] Peru Bhardwaj, John Kelleher, Luca Costabello, and Declan O'Sullivan. Adversarial attacks on knowledge graph embeddings via instance attribution methods. In *Proc. of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8225–8239, 2021.

[22] Peru Bhardwaj, John Kelleher, Luca Costabello, and Declan O'Sullivan. Poisoning knowledge graph embeddings via relation inference patterns. In *Proc. of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, pages 1875–1888, 2021.

[23] Franziska Boenisch, Adam Dziedzic, Roei Schuster, Ali Shahin Shamsabadi, Ilia Shumailov, and Nicolas Papernot. Reconstructing individual data points in federated learning hardened with differential privacy and secure aggregation. In *Proc. of the 2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 241–257. IEEE, 2023.

[24] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proc. of ACM Special Interest Group on Management of Data (SIGMOD)*, pages 1247–1250, 2008.

[25] Dan Boneh and Xavier Boyen. Short signatures without random oracles and the sdh assumption in bilinear groups. *J. Cryptol.*, 21(2):149–177, feb 2008.

[26] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Proc. of the Advances in neural information processing systems*, 2013.

[27] Kevin D Bowers, Ari Juels, and Alina Oprea. Hail: A high-availability and integrity layer for cloud storage. In *Proc. of ACM conference on Computer and communications security (CCS)*, pages 187–198, 2009.

[28] Marco Brandizi, Ajit Singh, and Keywan Hassani-Pak. Getting the best of linked data and property graphs: rdf2neo and the knetminer use case. In *SWAT4LS*, 2018.

[29] Carlos Buil-Aranda, Aidan Hogan, Jürgen Umbrich, and Pierre-Yves Vandenbussche. Sparql web-querying infrastructure: Ready for action? In *Proc. of the international Semantic Web Conference (ISWC)*, pages 277–293. Springer, 2013.

[30] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proc. of 2018 IEEE symposium on security and privacy (SP)*, pages 315–334, 2018.

[31] Min Cai and Martin Frank. Rdfpeers: a scalable distributed rdf repository based on a structured peer-to-peer network. In *Proc. of the Web Conference (WWW)*, pages 650–657, 2004.

[32] Matteo Campanelli, Dario Fiore, and Anaïs Querol. Legosnark: Modular design and composition of succinct zero-knowledge proofs. In *Proc. of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2075–2092, 2019.

[33] Ran Canetti, Omer Paneth, Dimitrios Papadopoulos, and Nikos Triandopoulos. Verifiable set operations over outsourced databases. In Hugo Krawczyk, editor, *Public-Key Cryptography – PKC 2014*, pages 113–130, 2014.

[34] Xiaoyu Cao, Jinyuan Jia, Zaixi Zhang, and Neil Zhenqiang Gong. Fedrecover: Recovering from poisoning attacks in federated learning using historical information. In *Proc. of the 2023 IEEE Symposium on Security and Privacy (SP)*, pages 1366–1383. IEEE, 2023.

[35] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *Proc. of OSDI*, volume 99, pages 173–186, 1999.

[36] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *Proc. of the International conference on the theory and application of cryptology and information security*, pages 577–594. Springer, 2010.

[37] Fei Chen, Fengming Meng, Tao Xiang, Hua Dai, Jianqiang Li, and Jing Qin. Towards usable cloud storage auditing. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2605–2617, 2020.

[38] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1243–1255, 2017.

[39] Mingyang Chen, Wen Zhang, Zhen Yao, Xiangnan Chen, Mengxiao Ding, Fei Huang, and Huajun Chen. Meta-learning based knowledge extrapolation for knowledge graphs in the federated setting. In *Proc. of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)*, 2022.

[40] Mingyang Chen, Wen Zhang, Zonggang Yuan, Yantao Jia, and Huajun Chen. Fede: Embedding knowledge graphs in federated setting. In *Proc. of the 10th International Joint Conference on Knowledge Graphs*, pages 80–88, 2021.

[41] Mingyang Chen, Wen Zhang, Zonggang Yuan, Yantao Jia, and Huajun Chen. Federated knowledge graph completion via embedding-contrastive learning. *Knowledge-Based Systems*, 252:109459, 2022.

[42] Mingyang Chen, Wen Zhang, Yushan Zhu, Hongting Zhou, Zonggang Yuan, Changliang Xu, and Huajun Chen. Meta-knowledge transfer for inductive knowledge graph embedding. In *Proc. of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 927–937, 2022.

[43] Usman W Chohan. Web 3.0: The future architecture of the internet? *Available at SSRN*, 2022.

[44] Daniel Demmler, Thomas Schneider, and Michael Zohner. Aby-a framework for efficient mixed-protocol secure two-party computation. In *Proc. of the NDSS*, 2015.

[45] Khoa Doan, Yingjie Lao, Weijie Zhao, and Ping Li. Lira: Learnable, imperceptible and robust backdoor attacks. In *Proc. of the IEEE/CVF international conference on computer vision*, pages 11966–11976, 2021.

[46] Junnan Dong, Qinggang Zhang, Xiao Huang, Keyu Duan, Qiaoyu Tan, and Zhimeng Jiang. Hierarchy-aware multi-hop question answering over knowledge graphs. In *Proc. of the ACM Web Conference 2023*, pages 2519–2527, 2023.

[47] Jacob Eberhardt and Jonathan Heiss. Off-chaining models and approaches to off-chain computations. In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, pages 7–12, 2018.

[48] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. Blockchaindb: A shared database on blockchains. volume 12, pages 1597–1609, 2019.

[49] Ethereum. Go ethereum, 2013.

[50] Ethereum. web3.js - ethereum javascript api, 2016.

[51] Javier D Fernández, Miguel A Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. Binary rdf representation for publication and exchange (hdt). *Journal of Web Semantics*, 19:22–41, 2013.

[52] Sébastien Ferré. Expressive and scalable query-based faceted search over sparql endpoints. In *Proc. of ISWC*, pages 438–453.

[53] Wensheng Gan, Zhenqiang Ye, Shicheng Wan, and Philip S Yu. Web 3.0: The future of internet. In *Proc. of the Web Conference (WWW)*, pages 1266–1275, 2023.

[54] Zhipeng Gao, Hongli Li, Kaile Xiao, and Qian Wang. Cross-chain oracle based data migration mechanism in heterogeneous blockchains. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1263–1268, 2020.

[55] Shijie Geng, Zuohui Fu, Juntao Tan, Yingqiang Ge, Gerard De Melo, and Yongfeng Zhang. Path language modeling over knowledge graphsfor explainable recommendation. In *Proc. of the ACM Web Conference 2022*, pages 946–955, 2022.

[56] Mitchell L Gordon, Michelle S Lam, Joon Sung Park, Kayur Patel, Jeff Hancock, Tatsunori Hashimoto, and Michael S Bernstein. Jury learning: Integrating dissenting voices into machine learning models. In *Proc. of CHI Conference on Human Factors in Computing Systems*, pages 1–19, 2022.

[57] Tao Guo, Song Guo, and Junxiao Wang. pfedprompt: Learning personalized prompt for vision-language models in federated learning. In *Proc. of the ACM Web Conference 2023*, pages 1364–1374, 2023.

[58] Tao Guo, Song Guo, Junxiao Wang, Xueyang Tang, and Wenchao Xu. Promptfl: Let federated participants cooperatively learn prompts instead of models-federated learning in age of foundation model. *IEEE Transactions on Mobile Computing*, 2023.

[59] Lars Heling and Maribel Acosta. Federated sparql query processing over heterogeneous linked data fragments. In *Proc. of the Web Conference (WWW)*, pages 1047–1057, 2022.

[60] Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, page 245–254, New York, NY, USA, 2018. Association for Computing Machinery.

[61] Maurice Herlihy, Barbara Liskov, and Liuba Shrira. Cross-chain deals and adversarial commerce. *The VLDB journal*, 31(6):1291–1309, 2022.

[62] Herumi. High-speed software implementation of the optimal ate pairing over barreto-naehrig curves, 2020.

[63] Zicong Hong, Song Guo, Peng Li, and Wuhui Chen. Pyramid: A layered sharding blockchain system. In *Proc. of IEEE INFOCOM*, 2021.

[64] Zicong Hong, Song Guo, Rui Zhang, Peng Li, Yufen Zhan, and Wuhui Chen. Cycle: Sustainable off-chain payment channel network with asynchronous rebalancing. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 41–53. IEEE, 2022.

[65] Yuke Hu, Wei Liang, Ruofan Wu, Kai Xiao, Weiqiang Wang, Xiaochen Li, Jinfei Liu, and Zhan Qin. Quantifying and defending against privacy threats on federated knowledge graph embedding. In *Proc. of the ACM Web Conference 2023*, pages 2306–2317, 2023.

[66] Junbeom Hur, Dongyoung Koo, Youngjoo Shin, and Kyungtae Kang. Secure data deduplication with dynamic ownership management in cloud storage. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 69–70. IEEE, 2017.

[67] Shagun Jhaver, Sucheta Ghoshal, Amy Bruckman, and Eric Gilbert. Online harassment and content moderation: The case of blocklists. *Journal of ACM Transactions on Computer-Human Interaction (TOCHI)*, 25(2):1–33, 2018.

[68] Seny Kamara, Tarik Moataz, Andrew Park, and Lucy Qin. A decentralized and encrypted national gun registry. In *Proc. of the 2021 IEEE Symposium on Security and Privacy (S&P '21)*, pages 1520–1537. IEEE, 2021.

[69] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-end integrity protection for web applications. In *Proc. of 2016 IEEE Symposium on Security and Privacy (SP)*, pages 895–913, 2016.

[70] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1575–1590, 2020.

[71] Lukas Klic. Linked open images: Visual similarity for the semantic web. *Journal of Semantic Web*, 14(2):197–208, 2023.

[72] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. In *Proc. of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1257–1272, 2017.

[73] Jingwei Li, Jin Li, Dongqing Xie, and Zhang Cai. Secure auditing and deduplicating data in cloud. *IEEE Transactions on Computers*, 65(8):2386–2396, 2015.

[74] Yang Li, Kai Zheng, Ying Yan, Qi Liu, and Xiaofang Zhou. EtherQL: a query layer for blockchain system. In *International Conference on Database Systems for Advanced Applications*, pages 556–567. Springer, 2017.

[75] Jinwen Liang, Zheng Qin, Sheng Xiao, Lu Ou, and Xiaodong Lin. Efficient and secure decision tree classification for cloud-assisted online diagnosis services. *Journal of IEEE Transactions on Dependable and Secure Computing*, 18(4):1632–1644, 2021.

[76] Jinwen Liang, Zheng Qin, Liang Xue, Xiaodong Lin, and Xuemin Shen. Verifiable and secure SVM classification for cloud-based health monitoring services. *Journal of IEEE Internet of Things Journal*, 8(23):17029–17042, 2021.

[77] Ganlin Liu, Xiaowei Huang, and Xinping Yi. Adversarial label poisoning attack on graph neural networks via label propagation. In *Proc. of the European Conference on Computer Vision*, pages 227–243. Springer, 2022.

[78] Hua-Xuan Liu, Bik-Chu Chow, Wei Liang, Holger Hassel, YaJun Wendy Huang, et al. Measuring a broad spectrum of ehealth skills in the web 3.0 context using an ehealth literacy scale: Development and validation study. *Journal of Medical Internet Research*, 23(9):e31627, 2021.

[79] Lihui Liu, Yuzhong Chen, Mahashweta Das, Hao Yang, and Hanghang Tong. Knowledge graph question answering with ambiguous query. In *Proc. of the ACM Web Conference 2023*, pages 2477–2486, 2023.

[80] Zhuotao Liu, Yangxi Xiang, Jian Shi, Peng Gao, Haoyu Wang, Xusheng Xiao, Bihan Wen, Qi Li, and Yih-Chun Hu. Make web3. 0 connected. *Journal of IEEE Transactions on Dependable and Secure Computing*, 2021.

[81] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proc. of the fifth Berkeley symposium on mathematical statistics and probability*, 1967.

[82] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. Bigchaindb: a scalable blockchain database. *white paper, BigChainDB*, 2016.

[83] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from

decentralized data. In *Proc. of the Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[84] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Proc. of the conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[85] Ken Miyachi and Timothy Mackey. hocbs: A privacy-preserving blockchain framework for healthcare data leveraging an on-chain and off-chain system design. *Inf. Process. Manag.*, 58(3):102535, 2021.

[86] Hamid Mozaffari, Virat Shejwalkar, and Amir Houmansadr. Every vote counts: Ranking-based training of federated learning to resist poisoning attacks. In *Proc. of the 32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[87] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. *Journal of ACM Transactions on Storage (TOS)*, 2(2):107–138, 2006.

[88] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[89] Abdelghny Orogat and Ahmed El-Roby. Smartbench: demonstrating automatic generation of comprehensive benchmarks for question answering over knowledge graphs. *Proceedings of the VLDB Endowment*, 15(12):3662–3665, 2022.

[90] Xudong Pan, Mi Zhang, Beina Sheng, Jiaming Zhu, and Min Yang. Hidden trigger backdoor attack on {NLP} models via linguistic style manipulation. In *Proc. of the 31st USENIX Security Symposium (USENIX Security 22)*, pages 3611–3628, 2022.

[91] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal verification of operations on dynamic sets. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 91–110, 2011.

[92] OriginTrail Parachain. Origintrail ecosystem white paper 2.0, 2022.

[93] Qingqi Pei, Enyuan Zhou, Yang Xiao, Deyu Zhang, and Dongxiao Zhao. An efficient query scheme for hybrid storage blockchains based on merkle semantic trie. In *Proc. of the International Symposium on Reliable Distributed Systems (SRDS)*, pages 51–60. IEEE, 2020.

[94] Hao Peng, Haoran Li, Yangqiu Song, Vincent Zheng, and Jianxin Li. Differentially private federated knowledge graphs embedding. In *Proc. of the 30th ACM International Conference on Information & Knowledge Management*, pages 1416–1425, 2021.

[95] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. Falcondb: Blockchain-based collaborative database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 637–652, New York, NY, USA, 2020.

[96] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on {OT} extension. In *Proc. of the 23rd USENIX Security Symposium (USENIX Security 14)*, pages 797–812, 2014.

[97] Eric Prudhommeaux. Sparql query language for rdf, 2008.

[98] Mayank Rathee, Conghao Shen, Sameer Wagh, and Raluca Ada Popa. Elsa: Secure aggregation for federated learning with malicious actors. In *Proc. of the 2023 IEEE Symposium on Security and Privacy (SP)*, pages 1961–1979. IEEE, 2023.

[99] Amrita Roy Chowdhury, Chuan Guo, Somesh Jha, and Laurens van der Maaten. Eiffel: Ensuring integrity for federated learning. In *Proc. of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2535–2549, 2022.

[100] Tara Safavi and Danai Koutra. Codex: A comprehensive knowledge graph completion benchmark. In *Proc. of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

[101] Muhammad Saleem, Ali Hasnain, and Axel-Cyrille Ngonga Ngomo. Largerdfbench: a billion triples benchmark for sparql endpoint federation. *Journal of Web Semantics*, 48:85–125, 2018.

[102] Adi Shamir. How to share a secret. *Journal of Communications of the ACM*, 22(11):612–613, 1979.

[103] Dan Sheridan, James Harris, Frank Wear, Jerry Cowell Jr, Easton Wong, and Abbas Yazdinejad. Web3 challenges and opportunities for the market. *arXiv preprint arXiv:2209.02446*, 2022.

[104] Amit Singhal et al. Introducing the knowledge graph: things, not strings. *Official google blog*, 5:16, 2012.

[105] Mirek Sopek, Przemyslaw Gradzki, Witold Kosowski, Dominik Kuziski, Rafa Trójczak, and Robert Trypuz. Graphchain: a distributed database with explicit semantics and chained rdf graphs. In *Proc. of the Web Conference (WWW)*, pages 1171–1178, 2018.

[106] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proc. of the Web Conference (WWW)*, pages 697–706, 2007.

[107] Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. Rotate: Knowledge graph embedding by relational rotation in complex space. In *Proc. of the International Conference on Learning Representations*, 2018.

[108] Roberto Tamassia. Authenticated data structures. In *Proc. of European symposium on algorithms*, pages 2–5, 2003.

[109] Kristina Toutanova, Danqi Chen, Patrick Pantel, Hoifung Poon, Pallavi Choudhury, and Michael Gamon. Representing text for joint embedding of text and knowledge bases. In *Proc. of the 2015 conference on empirical methods in natural language processing*, pages 1499–1509, 2015.

[110] Tamilla Triantoro and Nicole Jackson. From classroom to metaverse. towards methodology for upskilling and reskilling in the age of web 3.0. In *Proc. of PACIS*, 2022.

[111] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction. In *Proc. of the International conference on machine learning*, pages 2071–2080. PMLR, 2016.

[112] Abeba N. Turi. *Web 3.0: The Distributed Information Network Economy*, pages 87–121. 2020.

[113] Kristen Vaccaro, Ziang Xiao, Kevin Hamilton, and Karrie Karahalios. Contestability for content moderation. *Proc. of the ACM on Human-Computer Interaction*, 5(CSCW2):1–28, 2021.

[114] VMvare. Spring boot., 2023.

[115] Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, sep 2014.

[116] Aidmar Wainakh, Alejandro Sanchez Guinea, Tim Grube, and Max Mühlhäuser. Enhancing privacy via hierarchical federated learning. In *Proc. of the 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 344–347. IEEE, 2020.

[117] Shicheng Wan, Hong Lin, Wensheng Gan, Jiahui Chen, and Philip S Yu. Web3: The next internet revolution. *arXiv preprint arXiv:2304.06111*, 2023.

[118] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, et al. Operon: An encrypted database for ownership-preserving data management. *Proceedings of the VLDB Endowment*, 15(12):3332–3345, 2022.

[119] Xiang Wang, Tinglin Huang, Dingxian Wang, Yancheng Yuan, Zhenguang Liu, Xiangnan He, and Tat-Seng Chua. Learning intents behind interactions with knowledge graph for recommendation. In *Proceedings of the Web Conference 2021*, WWW '21, page 878–887, New York, NY, USA, 2021. Association for Computing Machinery.

[120] Xiting Wang, Kunpeng Liu, Dongjie Wang, Le Wu, Yanjie Fu, and Xing Xie. Multi-level recommendation reasoning over knowledge graphs with reinforcement learning. In *Proc. of the ACM Web Conference 2022*, pages 2098–2108, 2022.

[121] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. In *Proc. of the AAAI conference on artificial intelligence*, volume 28, 2014.

[122] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for {Zero-Knowledge} proofs with applications to machine learning. In *Proc. of the 30th USENIX Security Symposium (USENIX Security 21)*, pages 501–518, 2021.

[123] Wikidata. Wikidata: a free and open knowledge base that can be read and edited by both humans and machines, 2023.

[124] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

[125] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[126] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. {DIZK}: A distributed zero knowledge proof system. In *Proc. of the 27th USENIX Security Symposium (USENIX Security 18)*, pages 675–692, 2018.

[127] Songrui Wu, Qi Li, Guoliang Li, Dong Yuan, Xingliang Yuan, and Cong Wang. Servedb: Secure, verifiable, and efficient range queries on outsourced database. In *Proc. of 2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 626–637, 2019.

[128] Zhihui Wu, Yang Xiao, Enyuan Zhou, Qingqi Pei, and Quan Wang. A solution to data accessibility across heterogeneous blockchains. In *26th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2020, Hong Kong, December 2-4, 2020*, pages 414–421. IEEE, 2020.

[129] Min Xie, Haixun Wang, Jian Yin, and Xiaofeng Meng. Integrity auditing of outsourced data. In *Proc. of the VLDB Endowment*, volume 7, pages 782–793, 2007.

[130] Wenhan Xiong, Thien Hoang, and William Yang Wang. Deeppath: A reinforcement learning method for knowledge graph reasoning. In *Proc. of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 564–573, 2017.

[131] Cheng Xu, Ce Zhang, and Jianliang Xu. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proc. of ACM Special Interest Group on Management of Data (SIGMOD)*, pages 141–158, 2019.

[132] Yuqin Xu, Shangli Zhao, Lanju Kong, Yongqing Zheng, Shidong Zhang, and Qingzhong Li. Ecbc: A high performance educational certificate blockchain with efficient query. In *International Colloquium on Theoretical Aspects of Computing*, pages 288–304. Springer, 2017.

[133] Hiroyuki Yamada and Jun Nemoto. Scalar dl: scalable and practical byzantine fault detection for transactional database systems. *Proc. of the VLDB Endowment*, 15(7):1324–1336, 2022.

[134] Bishan Yang, Scott Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. In *Proc. of the International Conference on Learning Representations (ICLR)*, 2015.

[135] Sean Yang and Max Li. Web3. 0 data infrastructure: Challenges and opportunities. *Journal of IEEE Network*, 37(1):4–5, 2023.

[136] Wenkai Yang, Yankai Lin, Peng Li, Jie Zhou, and Xu Sun. Rethinking stealthiness of backdoor attack against nlp models. In *Proc. of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5543–5557, 2021.

[137] Yuhao Yang, Chao Huang, Lianghao Xia, and Chenliang Li. Knowledge graph contrastive learning for recommendation. In *Proc. of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1434–1443, 2022.

[138] Zhanpeng Yang, Yuanming Shi, Yong Zhou, Zixin Wang, and Kai Yang. Trustworthy federated learning via blockchain. *IEEE Internet of Things Journal*, 10(1):92–109, 2022.

[139] Zuoru Yang, Jingwei Li, and Patrick PC Lee. Secure and lightweight deduplicated storage via shielded {Deduplication-Before-Encryption}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 37–52, 2022.

[140] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proc. of the 27th Annual Symposium on Foundations of Computer Science (SFCS '86)*, pages 162–167. IEEE, 1986.

[141] Xiaoyu You, Beina Sheng, Daizong Ding, Mi Zhang, Xudong Pan, Min Yang, and Fuli Feng. Mass: Model-agnostic, semantic and stealthy data poisoning attack on knowledge graph embedding. In *Proc. of the ACM Web Conference 2023*, pages 2000–2010, 2023.

[142] Cong Yue, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, Sheng Wang, and Xiaokui Xiao. Analysis of indexing structures for immutable data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 925–935, 2020.

[143] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale rdf data. volume 6, pages 265–276, 2013.

[144] Ce Zhang, Cheng Xu, Haixin Wang, Jianliang Xu, and Byron Choi. Authenticated keyword search in scalable hybrid-storage blockchains. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 996–1007, 2021.

[145] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. Gem2-tree: A gas-efficient structure for authenticated range queries in blockchain. In *Proc. of IEEE International Conference on Data Engineering (ICDE)*, pages 842–853, 2019.

[146] Chuan Zhang, Mingyang Zhao, Liehuang Zhu, Weiting Zhang, Tong Wu, and Jianbing Ni. FRUIT: A blockchain-based efficient and privacy-preserving quality-aware incentive scheme. *IEEE Journal on Selected Areas in Communications*, Early Access, 2022.

[147] Fuzheng Zhang, Nicholas Jing Yuan, Defu Lian, Xing Xie, and Wei-Ying Ma. Collaborative knowledge base embedding for recommender systems. In *Proc. of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 353–362, 2016.

[148] Hengtong Zhang, Tianhang Zheng, Jing Gao, Chenglin Miao, Lu Su, Yaliang Li, and Kui Ren. Data poisoning attack against knowledge graph embedding. In *Proc. of the 28th International Joint Conference on Artificial Intelligence*, pages 4853–4859, 2019.

[149] Hongming Zhang, Xin Liu, Haojie Pan, Yangqiu Song, and Cane Wing-Ki Leung. Aser: A large-scale eventuality knowledge graph. In *Proceedings of The Web Conference 2020*, WWW '20, page 201–211, New York, NY, USA, 2020. Association for Computing Machinery.

[150] Kai Zhang, Yu Wang, Hongyi Wang, Lifu Huang, Carl Yang, and Lichao Sun. Efficient federated learning on knowledge graphs via privacy-preserving relation embedding aggregation. In *Proc. of ACL 2022 Workshop on Federated Learning for Natural Language*, 2022.

[151] Meihui Zhang, Zhongle Xie, Cong Yue, and Ziyue Zhong. Spitz: a verifiable database system. *Proc. of the VLDB Endowment*, 13(12):3449–3460, 2020.

[152] Meihui Zhang, Zhongle Xie, Cong Yue, and Ziyue Zhong. Spitz: A verifiable database system. *Proc. VLDB Endow.*, 13(12):3449–3460, August 2020.

[153] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vsql: Verifying arbitrary sql queries over dynamic outsourced databases. In *Proc. of IEEE Symposium on Security and Privacy (SP)*, pages 863–880, 2017.

[154] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. Integridb: Verifiable sql for outsourced databases. In *Proc. of ACM Computer and Communications Security (CCS)*, pages 1480–1491, 2015.

[155] Shangfei Zheng, Weiqing Wang, Jianfeng Qu, Hongzhi Yin, Wei Chen, and Lei Zhao. Mmkgr: Multi-hop multi-modal knowledge graph reasoning. In *Proc. of the 2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 96–109. IEEE, 2023.

[156] Xiangrong Zhu, Guangyao Li, and Wei Hu. Heterogeneous federated knowledge graph embedding learning and unlearning. In *Proc. of the ACM Web Conference 2023*, pages 2444–2454, 2023.

[157] Yanchao Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, and Ying Yan. SEBDB: Semantics empowered blockchain database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1820–1831. IEEE, 2019.