CEUR-WS.org/Vol-2721/paper476.pdf

# Demonstration of GOOSE: A Secure Framework for Graph Outsourcing and SPARQL Evaluation*

Radu Ciucanu[1] and Pascal Lafourcade[2]

[1] INSA Centre Val de Loire, Univ. Orléans, LIFO EA 4022, France
radu.ciucanu@insa-cvl.fr
[2] Université Clermont Auvergne, LIMOS CNRS UMR 6158, France
pascal.lafourcade@uca.fr

**Abstract.** We demonstrate GOOSE, an open-source framework for secure graph outsourcing and SPARQL evaluation. We showcase the workflow of GOOSE over various real-world use cases, the scalability of GOOSE, and the security properties that GOOSE guarantees in the honest-but-curious cloud security model.

## 1 Introduction

Enhancing Semantic Web technologies with security and privacy guarantees is an important and popular problem [8]. Several systems have been proposed to tackle different settings, from both security (e.g., [7]) and privacy (e.g., [6]) viewpoints.

We take a complementary look by addressing the security issues that occur when outsourcing an RDF graph to the cloud and querying the outsourced graph with SPARQL. Our scenario is inspired by the *database as a service* cloud computing model [5], where a *data owner* outsources some data to the cloud, then a *user* is allowed to submit queries to the cloud, which computes and returns the query answers to the user. We assume that the cloud is *honest-but-curious* i.e., executes tasks dutifully, but tries to gain as much information as possible.

We demonstrate GOOSE, an open-source framework that relies on cryptographic schemes and secure multi-party computation to achieve desirable security properties: (i) no cloud node can learn the graph, (ii) no cloud node can learn at the same time the query and the query answers, and (iii) an external network observer cannot learn the graph, the query, or the query answers. GOOSE has been presented[3] as a full paper at the DBSec 2020 [4] conference. The goal of this demo paper is to showcase GOOSE to the Semantic Web community. Indeed, GOOSE is an innovative system that allows secure data outsourcing and query evaluation relevant to popular Semantic Web technologies (RDF and SPARQL).

In Sect. 2, we present an overview of GOOSE, whereas in Sect. 3 we describe our demonstration scenarios. Due to lack of space, we omit several details (related work, theoretical and empirical analysis) that can be found in the GOOSE

* Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[3] https://www.youtube.com/watch?v=ZhtpulFf3rs

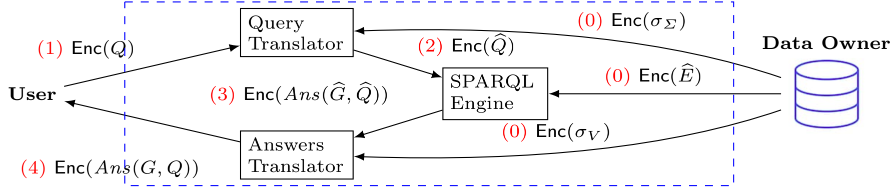


**Fig. 1.** Architecture of GOOSE.

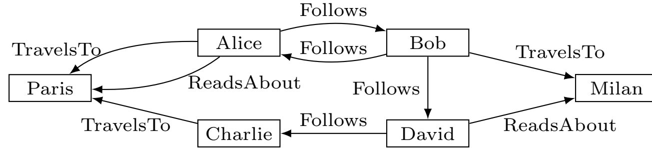


**Fig. 2.** Example of graph database.

conference paper [4]. The open-source code of GOOSE, as well as the different use cases and data that we use throughout our demonstration scenarios are available on GitHub [4].

## 2 System Overview

In Fig. 1, we show the architecture of GOOSE, which has 5 participants: *data owner* (DO), who owns the graph that it outsources to the cloud in order to be queried, *user* (U), who submits graph queries to the cloud and receives query answers, and 3 cloud participants: *query translator* (QT), *SPARQL engine* (SE) and *answers translator* (AT). Each Enc from Fig. 1 uses the AES [1] key shared between the 2 concerned participants. We next outline GOOSE via an example.

**Graph Data and Queries.** An RDF[5] graph is a set of triples (subject, predicate, object). For the goal of this paper, we simply assume that a graph $G{=}(V, E)$ is a *directed, edge-labeled graph*, where $V$ is a set of nodes and $E \subseteq V \times \Sigma \times V$ is a set of directed edges between nodes of $V$, with labels from an *alphabet* $\Sigma$. The graph in Fig. 2 has 6 nodes, an alphabet of 3 possible edge labels, and 9 edges.

We focus on *Unions of Conjunctions of Regular Path Queries* (UCRPQ) that are at the core of SPARQL 1.1[6], including recursive queries via Kleene star. By $Ans(G, Q)$ we denote the answers of query $Q$ over a graph $G$, using standard SPARQL semantics. For example, the UCRPQ $(?x, ?z) \leftarrow (?x, \text{Follows}^+, ?y), (?y, \text{TravelsTo}, ?z)$ selects nodes $?x, ?z$ s.t. there exists node $?y$ s.t. one can go from $?x$ to $?y$ with a path in the language of "Follows$^+$" and can go from $?y$ to $?z$ with a path in the language of "TravelsTo". The answers of this query on the

[4] https://github.com/radu1/goose
[5] https://www.w3.org/TR/rdf11-concepts/
[6] https://www.w3.org/TR/sparql11-query/

graph from Fig. 2 are (Alice, Milan), (Alice, Paris), (Bob, Milan), (Bob, Paris), (David, Paris). For example, the tuple (Alice, Paris) is an answer because of paths Alice $\xrightarrow{\text{Follows}}$ Bob $\xrightarrow{\text{Follows}}$ David $\xrightarrow{\text{Follows}}$ Charlie and Charlie $\xrightarrow{\text{TravelsTo}}$ Paris, where $?x, ?y, ?z$ are mapped to Alice, Charlie, Paris, respectively.

**Step 0.** The *graph outsourcing* (i.e., the 3 outgoing arrows from DO in Fig. 1) is done only once at the beginning by DO. Intuitively, DO sends to each cloud participant a piece of $G$ s.t. each participant can perform its task during query evaluation but no participant can reconstruct the entire graph. To to so, DO generates 2 random bijections: $\sigma_\Sigma$ (for edge labels) and $\sigma_V$ (for graph nodes). By $\sigma^{-1}$ we denote the inverse of $\sigma$ (this is needed later on at the end of query evaluation). For our example graph in Fig. 2, DO may generate:

$$\sigma_V = \{\text{Alice} \to 5, \text{Bob} \to 3, \text{Charlie} \to 0, \text{David} \to 1, \text{Milan} \to 2, \text{Paris} \to 4\}$$
$$\sigma_\Sigma = \{\text{Follows} \to 1, \text{ReadsAbout} \to 2, \text{TravelsTo} \to 0\}.$$

DO uses these 2 functions to hide graph edges: by $\widehat{E}$ we denote the hidden set of edges generated from $E$, where nodes are replaced using $\sigma_V$, and edge labels are replaced using $\sigma_\Sigma$. On our example in Fig. 2, edge (Alice, Follows, Bob) becomes (5, 1, 3), edge (Alice, ReadsAbout, Paris) becomes (5, 2, 4), and finally:

$$\widehat{E} = \{(5,1,3), (5,2,4), (5,0,4), (3,1,5), (3,1,1), (3,0,2), (0,0,4), (1,1,0), (1,2,2)\}.$$

Each message sent over the network is encrypted with the key shared between DO and the corresponding cloud participant, which can decrypt the message upon reception. Messages are encrypted to avoid that an external observer that sees them in clear is able to learn the graph $G$. The distribution of graph storage among cloud participants makes that none of them can learn the graph $G$.

We next discuss *query evaluation* i.e., steps 1–4 cf. Fig. 1, done for each query submitted by U. Each message exchanged over the network during query evaluation is encrypted with the key shared between corresponding participants, such that an external observer cannot learn the query and its answers.

**Step 1.** U submits query $Q$ to QT. For example, recall the aforementioned query $(?x, ?z) \leftarrow (?x, \text{Follows}^+, ?y), (?y, \text{TravelsTo}, ?z)$.

**Step 2.** QT translates $Q$ by replacing all labels used in $Q$ using the function $\sigma_\Sigma$ received from DO. By $\widehat{Q}$ we denote the query $Q$ translated using $\sigma_\Sigma$. On our example, query from step 1 becomes $(?x, ?z) \leftarrow (?x, 1^+, ?y), (?y, 0, ?z)$.

**Step 3.** SE evaluates translated query $\widehat{Q}$ received from QT at step 2 on the graph with hidden nodes and edge labels as defined by $\widehat{E}$ received from DO during step 0. To do so, SE simply uses some standard SPARQL engine as a black-box, without any change to the query engine[7]. We denote the result of SE by $Ans(\widehat{G}, \widehat{Q})$, where the true answers $Ans(G, Q)$ are still hidden using function $\sigma_V$. On our example, $Ans(\widehat{G}, \widehat{Q}) = \{(5, 2), (5, 4), (3, 2), (3, 4), (1, 4)\}$.

**Step 4.** AT uses function $\sigma_V^{-1}$ to translate hidden query answers $Ans(\widehat{G}, \widehat{Q})$ into true query answers. On our example, AT recovers $Ans(G, Q) =$ {(Alice, Milan), (Alice, Paris), (Bob, Milan), (Bob, Paris), (David, Paris)} that AT sends to U.

[7] In our implementation, we rely on Apache Jena `https://jena.apache.org/`

## 3 Demonstration Scenarios

We (i) introduce via examples the complete workflow of GOOSE and the class of supported SPARQL queries using real-world scenarios, (ii) emphasize the scalability of GOOSE, and (iii) point out the security properties of GOOSE and the security model in which these properties hold.

**(i) GOOSE by example.** On the GitHub repository of GOOSE (URL given in Sect 1), in the directory `running-example`, we included the script `example.sh` that reproduces the running example from Sect. 2 and [4]. To analyze the graph, query, and query answers used by this script, see sub-directories `example` and `example-secure` for standard and GOOSE versions, respectively. In particular, the files from `example-secure` hide nodes and edges as outlined in Sect. 2. Notice that we chose to initially specify the UCRPQ in an XML format and then translate them in SPARQL. The aforementioned XML format and the translation to SPARQL are based on gMark [2,3], a state-of-the art generator of synthetic graphs and UCRPQ workloads. Our choice is motivated by the observation that GOOSE can be easily extended to secure graph outsourcing and UCRPQ evaluation, regardless the practical language used to encode the UCRPQ. Indeed, one can easily modify GOOSE to translate the UCRPQ in SQL with recursive views instead of SPARQL and use PostgreSQL instead of Apache Jena, and hence obtain a practical system guaranteeing exactly the same security properties.

Going back to the demo, we stress that the aforementioned running example script provided on the GOOSE repository can be easily run on a laptop. If one tries the script and gets some error, it is likely that there are some missing packages. GOOSE is written in Python, and uses Apache Jena (written in Java) for SPARQL evaluation and gMark (written in C++) for graph and query workload generation. The script `install-libraries.sh` installs the necessary libraries. Before running this script, one should be aware that, depending on the former state of her computer, it may be more or less trivial to go back to that state[8].

In addition to the running example, we have a predefined script that relies on four real-world cases based on gMark: *uniprot* (biological data where proteins interact with other proteins, are encoded on genes, etc.), *shop* (online shop selling different types of products to users, etc.), *social-network* (social network where persons know other persons, work in companies, etc.), and *bib* (bibliographical data about researchers that author papers published in journals or conferences, etc.). We discuss how to use this script when describing the next scenario.

**(ii) Scalability.** The idea of this scenario is to generate graphs of increasing sizes and observe that GOOSE has a linear time behavior for the graph outsourcing. As for the query evaluation, we generate queries having diverse properties (w.r.t. arity, selectivity, shape, and use of recursion), run GOOSE, and zoom on the time taken by each step of GOOSE. One should observe that the bottleneck of secure query evaluation in GOOSE does not come from the use of cryptographic primitives, but is due to the SPARQL engine used as a black-box, in particular for

[8] We leave as future work the "dockerization" suggested by Anonymous Reviewer 1.

evaluating recursive queries. These observations should confirm the theoretical and empirical analysis detailed in the full paper on GOOSE [4].

The script `script-complete-workflow.sh` allows to run such a complete workflow. As currently configured, the script should generate the large-scale experiment reported in [4], which took 8 days and generated 46GB of data (total size for graphs, queries, and query answers). To generate quicker scalability experiments, one can simply tune to smaller values the 5 parameters that have self-explanatory names with `scaling_factor` as a substring. To change the gMark graph and query workload configurations, one can tune the XML files from directory `gmark/use-cases` (see [2,3] for the meaning of the gMark constraints).

**(iii) Security.** To emphasize the challenges of building a system like GOOSE and to understand what GOOSE design choices make it secure in the honest-but-curious cloud adversary model, we refer to the cryptographic tools and security theorems from [4]. For instance, the non-deterministic encryption mode AES-CBC that we chose for GOOSE implies that: for a given graph, if two distinct queries yield identical answer sets, then these answer sets are encrypted differently, hence an external network observer (e.g., a curious cloud admin) that analyzes the messages exchanged over the network cannot know whether two queries are equivalent on a specific graph. On the other hand, if one assumes stronger attacks (e.g., a network observer that has as background knowledge some partial knowledge on the graph), that could break some GOOSE security properties by leaking partial knowledge on the queries and their answers.

## References


1. Advanced Encryption Standard (AES) (2001), FIPS Publication 197
2. Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H.L., Lemay, A., Advokaat, N.: Generating Flexible Workloads for Graph Databases. PVLDB **9**(13), 1457–1460 (2016)
3. Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H.L., Lemay, A., Advokaat, N.: gMark: Schema-Driven Generation of Graphs and Queries. IEEE TKDE **29**(4), 856–869 (2017)
4. Ciucanu, R., Lafourcade, P.: GOOSE: A Secure Framework for Graph Outsourcing and SPARQL Evaluation. In: DBSec. pp. 347–366 (2020), https://doi.org/10.1007/978-3-030-49669-2_20
5. Curino, C., Jones, E.P.C., Popa, R.A., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., Zeldovich, N.: Relational Cloud: a Database Service for the Cloud. In: CIDR. pp. 235–240 (2011)
6. Delanaux, R., Bonifati, A., Rousset, M., Thion, R.: Query-Based Linked Data Anonymization. In: ISWC. pp. 530–546 (2018)
7. Fernández, J., Kirrane, S., Polleres, A., Steyskal, S.: $\text{HDT}_{crypt}$: Compression and Encryption of RDF Datasets. Semantic Web Journal (2018)
8. Kirrane, S., Villata, S., d'Aquin, M.: Privacy, Security and Policies: A Review of Problems and Solutions with Semantic Web Technologies. Semantic Web **9**(2), 153–161 (2018)