Towards Fine Grained RDF Access Control

Jyothsna Rachapalli, Vaibhav Khadilkar, Murat Kantarcioglu and Bhavani Thuraisingham
The University of Texas at Dallas
{jxr061100, vvk072000, muratk, bxt043000}@utdallas.edu

ABSTRACT

The Semantic Web is envisioned as the future of the current web, where the information is enriched with machine understandable semantics. According to the World Wide Web Consortium (W3C), "The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries". Among the various technologies that empower Semantic Web, the most significant ones are Resource Description Framework (RDF) and SPARQL, which facilitate data integration and a means to query respectively. Although Semantic Web is elegantly and effectively equipped for data sharing and integration via RDF, lack of efficient means to securely share data pose limitations in practice. In order to make data sharing and integration pragmatic for Semantic Web, we present a query language based secure data sharing mechanism. We extend SPARQL with a new query form called `SANITIZE` which comprises a set of sanitization operations that are used to sanitize or mask sensitive data within an RDF graph. The sanitization operations can be further leveraged towards RDF access control and anonymization, thus enabling secure sharing of RDF data.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.4.6 [**Operating Systems**]: Security and Protection—*Access Control*

Keywords

RDF; SPARQL; Sanitization; Access Control; Security

1. INTRODUCTION

The Relational Database Management System (RDBMS) is one of the most popular paradigms to capture data and build applications in the IT domain. It is good for enterprise scale performance and reliability, however, it falls short on these metrics at the scale of the Internet. Semantic Web is a popular emerging paradigm for capturing data based on requirements pertaining to Internet scale and has been perceived as the future of the web, comprising a global database.

There has been a significant amount of research in the database community in the area of data security, which can be broadly classified into access control research and data privacy research [1]. Relational databases have developed sound features for providing access control [2], [3], [4] and data privacy [5] (such as k-anonymity, l-diversity and t-closeness), which also includes differential privacy [6]. Although there have been attempts at developing access control techniques for RDF (see for example, [7], [8], [9], [10], and [11]), there is no agreed upon consensus for securing the underlying RDF data. Since security or privacy definitions and intent change by virtue of application requirements that span a vast number of domains, we need a general and fundamental mechanism for securing RDF graphs, which should essentially comprise RDF graph transformation operations. Towards this end, we present a language for sanitizing RDF graphs, which is an attempt at providing a formal foundational framework that will help in implementing various security and privacy features for RDF data.

We propose to secure RDF graphs through sanitization, which is a process of masking sensitive data within an RDF graph with an appropriate replacement value in order to reduce the risk of data exposure. There are two crucial requirements for performing sanitization of RDF graphs. Firstly, one needs to automate the generation of replacement values for any given sensitive data item. Secondly, one should be able to perform automatic synchronization, *i.e.,* a given sensitive data item is masked consistently not only within the given sensitive RDF subgraph but also everywhere else it occurs in the graph. Further, RDF sanitization can be useful under two use cases. In the case where an RDF dataset needs to be outsourced and shared with a third party, for *e.g.*, when a hospital wants to release patient data for research purposes. In this case sanitization can be performed on the entire RDF dataset before sharing. In an access control like scenario where data is present in original/unmasked form but needs to be hidden from those who are not authorized, sanitization can be performed dynamically, on a per query basis, on the relevant subgraph of the RDF dataset, which is being accessed by the user query.

Having described the process of RDF graph sanitization, we now describe why SPARQL based sanitization is crucial for fine grained RDF access control. Relational model pertaining to RDBMS is based on first-order predicate logic and set theory. The query language SQL, which is based

SACMAT'14, June 25–27, 2014, London, Ontario, Canada.
Copyright 2014 ACM 978-1-4503-2939-2/14/06 ...$15.00.
http://dx.doi.org/10.1145/2613087.2613092.

on relational algebra or calculus, serves as a means to retrieve or operate upon data. On the other hand, in Semantic Web knowledge bases, the knowledge representation or ontology languages (OWL dialects) are based on description logics, which in turn are decidable fragments of first order logic with varying levels of expressivity. The data model used within Semantic Web is RDF, which is a graph data model. SPARQL [12] is an RDF query language that can be used to formulate queries ranging from simple graph pattern matching to complex queries, which may include operators such as union, optional, filters, path expressions *etc.* Since SPARQL is rich in expressivity (when compared to SQL), it can be used to formulate highly sophisticated queries to access or retrieve a wide variety of data items from RDF stores. Therefore, the mechanism to secure access to RDF data is required to commensurately match this expressivity.

In relational databases, views provide a very powerful security mechanism for controlling what information can be accessed. A view based approach essentially gleans data from various relations, which can be safely shared with an authorized user. A similar approach could be used to secure access to RDF data. However, unlike RDBMS where data is modeled as relations, in RDF data is captured as a graph comprising a set of triples. Creation of a view in the context of RDF would translate to creation of a view subgraph constructed by gleaning required data. However, since RDF is a graph data model, the problem of subgraph extraction may not always be feasible or it may be too tedious to create the desired subgraph by gleaning suitable data. Since this process may not lend the desired flexibility required for fine grained RDF access control, we take the process of securing RDF graphs a step further. Instead of creating secure view subgraphs from RDF datasets by gleaning data, we choose to dynamically and selectively mask the sensitive data within an RDF graph, which also preserves the correctness of ontological definitions of an RDF graph. We model the various graph transformation tasks in the form of query operations.

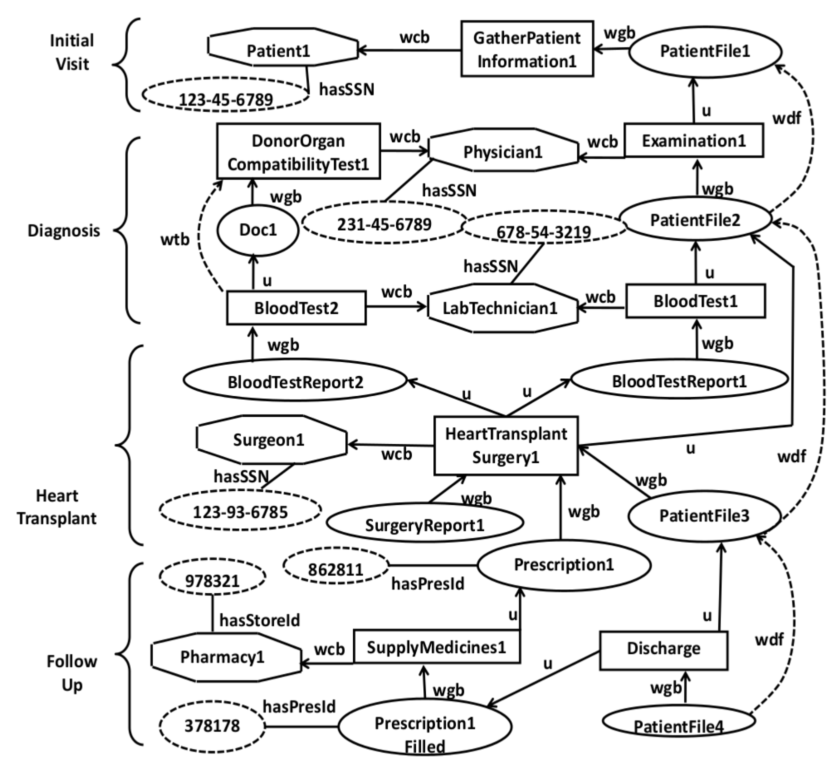
Since the existing SPARQL operations such as `UPDATE` or `CONSTRUCT` do not support RDF graph sanitization (details provided in Appendix 9.1), we propose a language for sanitizing RDF graphs in order to enable secure RDF data sharing. However, this language is not a replacement of a policy based RDF access control language. In fact, it is an essential tool that can be leveraged by a policy based language in order to effectively and efficiently provide fine grained RDF access control. Although we present a preliminary version of such a policy based access control language, we leave the extensive version of this language as a part of the future work and restrict the scope of this work to details of the RDF sanitization language.

In this paper we do not try to address the question: "*What Data to Sanitize?*", we instead try to address "*How to sanitize RDF data?*". In the following we explain the reason behind the same. The data captured in RDF, which is a graph data model, is semi-structured. Therefore, the flexibility of RDF can be used to capture a wide variety of data ranging from relational data (table) to social networks (graph) and provenance (directed acyclic graph). Since we provide a framework/language for sanitizing RDF data which is semi-structured, there is no formal general model (unlike RDBMS with relational model) upon which one can build general privacy definitions and formal adversarial model. Additional domain specific information, such as vocabulary or ontology of the domain along with the ontology language used is required to build privacy definitions. Since the structure of the data that can be captured in RDF is variable and can change from one domain to another, privacy definition and utility requirements will change depending on the domain. Therefore, coming up with a general threat model or a formal adversarial model and privacy definitions is not feasible at RDF level. However, if such a definition exists, then using our framework and especially the masking function (*genUniq*), one can create their own custom masking functions, which are suitable for their data and application scenarios and override the original definition to meet the desired privacy requirements suitable for target application domain. The scope of this paper is to address the problem of RDF graph sanitization, which is general enough to be used with RDF data from any domain to build more sophisticated security features like fine grained access control and data privacy. *Our contributions are listed below:*

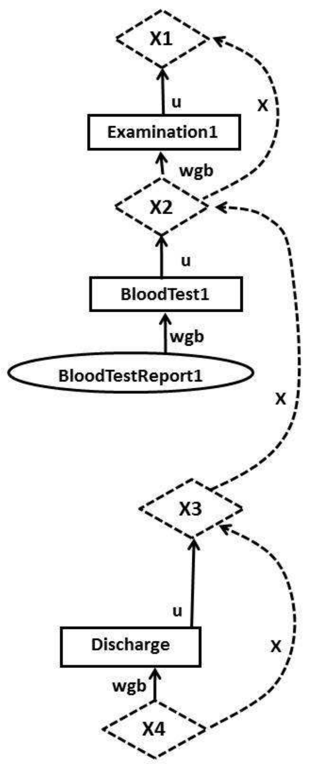
- We identify, formulate and delineate the requirements of RDF graph sanitization.
- We present a sanitization extension of SPARQL comprising a set of RDF graph sanitization operations along with their time complexity analysis.
- We present denotational/compositional semantics of this extension of SPARQL.
- We present a prototype system and its architecture based on a healthcare provenance scenario and illustrate how one can build a fine grained access control mechanism using our graph sanitization operations.
- We present empirical results showing the performance of the sanitization operations, which were evaluated on synthetic as well as real world datasets.

2. RDF GRAPH SANITIZATION

An RDF graph is made up of a set of triples which in turn are made up of a subject, predicate and an object [13]. A sensitive data item in an RDF graph can be (among many forms) a literal value or a resource node (IRI), a set or class of resource nodes or data values, an edge or a relationship between two nodes and a path or a subgraph containing multiple nodes connected by edges. We present a set of corresponding graph sanitization operations, namely Sanitize Node (SNode), Sanitize Edge (SEdge) and Sanitize Path (SPath). The paper first describes the sanitization operations in imperative (procedural) style, in the form of algorithms, which is technology agnostic, implementation friendly and facilitates complexity analysis and better comprehensibility. We then present the formal model using denotational semantics, which is declarative, *i.e.*, it unambiguously states what needs to be done without stating how it needs to be done. This offers flexibility of allowing others to come up with a more efficient algorithm for same semantics. In the following, we first describe a healthcare provenance scenario that will be used in the rest of the paper to better illustrate the functionality of the sanitization operations. We then describe some of the related formal notation, sanitization algorithms and formal semantics.



(a) Healthcare Provenance



(b) SPath

Figure 1: Provenance Healthcare Scenario and SPath Sanitization Operation Illustration

2.1 Scenario: Healthcare Provenance

We built the healthcare provenance usecase based on the Open Provenance Model (OPM) [14], which is a general model of provenance. An OPM provenance graph comprises three types of nodes, namely Artifacts, Processes and Agents, which are represented by an oval, rectangle and hexagon respectively. It additionally defines five types of dependencies represented by edges connecting these nodes namely, used (u), wasGeneratedBy (wgb), wasControlledBy (wcb), wasTriggeredby (wtb) and wasDerivedFrom (wdf). We capture our usecase with an OWL ontology by extending Jun Zhao's Open Provenance Model Vocabulary (OPMV) [15], which is a lightweight vocabulary used to describe the core concepts of OPM.

Figure 1(a) depicts a sample provenance of a patient's electronic healthcare record (EHR). The provenance, or medical history of Patient1 is shown in various stages namely, **Initial Visit**, **Diagnosis**, **Heart Transplant** and **Follow Up**. During the initial visit, the patient filled out his information through the GatherPatientInformation1 process and an artifact PatientFile1 was generated by it. Physician1 conducted Examination1 (which generated document PatientFile2) and decided to perform a heart transplant surgery. Therefore, he carried out DonorOrganCompatibilityTest1, which triggered the BloodTest2 process (at donor side). This process was controlled by LabTechnician1, who also controlled process BloodTest1 for the recipient, Patient1. Subsequently, Surgeon1 controlled the HeartTransplantSurgery1 process, which used artifacts BloodTestReport1, BloodTestReport2, PatientFile2 and in turn generated SurgeryReport1, Prescription1 and PatientFile3. The process SupplyMedicines1 used document Prescription1, generated document Prescription1Filled and was controlled by Pharmacy1. Finally, the process Discharge used documents Prescription1Filled and PatientFile3 and generated PatientFile4. In the following, we will use this medical workflow to illustrate the various graph sanitization operations and will refer to it as G_q.

2.2 Related Formal Notation

RDF terms denoted by T, comprise disjoint infinite sets, I, B and L (IRI's, Blank nodes and Literals respectively). A triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, where s is a subject, p a predicate, and o is an object. A triple pattern, TP, is a triple of the form $(sp, pp, op) \in (I \cup V \cup L) \times (I \cup V) \times (I \cup V \cup L)$, where V is an infinite set of variables that is disjoint from the sets I, B, and L; and sp, pp, and op are a subject pattern, predicate pattern and object pattern respectively. A mapping μ is a partial function $\mu : V \rightarrow T$. Given a triple pattern t, $\mu(t)$ denotes the triple obtained by replacing the variables in t according to μ [16]. Domain of μ, $dom(\mu)$ is the subset of V where μ is defined and Ω is defined as a set of mappings μ. A property path is a possible route through a graph between two graph nodes [12]. A property path expression is similar to a string regular expression but over properties, not characters. The ends of the path may be RDF terms or variables. Variables cannot be used as part of the path itself, but only at the ends. A property path pattern is a triple pattern created by using a property path expression in place of a property or predicate. It is a member of the set $(T \cup V) \times PE \times (T \cup V)$, where PE is the set of all property path expressions. In the following, we introduce some terms in addition to the standard ones provided in [12]: (1) *Predicate Triple Pattern* $TP' = \{tp|\ tp$ is a triple pattern where $sp \in V$, $pp \in I$ and $op \in V\}$. (2) *Type Triple Pattern* $TTP = \{tp|tp$ is a triple pattern where the predicate pattern, *pp*, is $rdf : type$, $sp \in V$ and $op \in I\}$. (3) *Type Graph Pattern* $GP' = \{gp|gp$ is a graph pattern of form tp_1 AND tp_2 where $tp_1 \in TTP \wedge tp_2 \in TP' \wedge (tp_1.sp = tp_2.sp \ \vee \ tp_1.sp = tp_2.op)\}$.

2.3 Node Sanitization (SNode)

The purpose of the Sanitize Node operation is to sanitize and protect a sensitive node/resource. This can be accomplished in two ways. Firstly, by masking or protecting the identifying attributes (object values) such as name, SSN's, health insurance id's (datatype predicates), *etc.* of a sensitive resource (subject of a triple). An example SPARQL query Q1 to perform such data sanitization is shown below. Query Q1 takes a triple as input and sanitizes the object part of the triple. When Q1 is run on G_q, only one triple is sanitized and the remaining graph stays the same, *i.e.*, the query transforms the SSN (U.S. Social Security Number) value from "123-45-6789" to "XXX" as shown in Figure 2(a). If one needs to conceal all SSN values in G_q, one may use the variation of SNode shown in Q2, where the sensitive data item is depicted by TP'. However, if one needs to conceal SSN values of a class of individuals in G_q, such as Physicians, then one may use the variation of SNode shown in Q3. Secondly, we consider an interesting case for masking or protecting a node/IRI, for *e.g.*, "Hide all the secret service agents who worked on top secret missions". Similarly, in Q4 as well, we try to hide the node identified in association with or dependent on a sensitive resource (surgery, top secret mission, *etc.*), which itself need not be hidden. Further, one may wish to perform synchronization, *i.e.*, "not only hide the surgeon (agent) in the given triple but also in the remaining triples of the graph". This step however, is optional depending on the application requirements.

```
Q1: SANITIZE Gq WHEREs {SNode (Surgeon1 hasSSN "123-45-6789")}

Q2: SANITIZE Gq WHEREs {SNode (?s hasSSN ?o)}

Q3: SANITIZE Gq WHEREs {SNode (?s rdf:type Physician . ?s hasSSN ?o)}

Q4: SANITIZE Gq WHEREs {SNode (?s rdf:type Surgery . ?s wcb ?o)} SYNC
```

The Sanitization query form comprises a sanitize clause, a where clause and optionally a synchronization clause. The sanitize clause uses keyword `SANITIZE` and specifies the sensitive graph that needs to be sanitized. The where clause uses keyword `WHEREs` and is used to specify the sanitization operation being used and the pattern used to access the sensitive resource. The letter "*s*" at the end of the keyword `WHEREs` is shown in lower case for better readability and used to indicate that a sanitization where clause is different from the SPARQL where clause keyword. Synchronization represented by the keyword `SYNC` is used only for consistently masking nodes or IRI's as synchronization of literal values is not meaningful. If keyword `SYNC` is not present then no synchronization is performed. In the case of SNode there is only one possible meaning or default meaning of Synchronization, *i.e.*, synchronization of the object node(s).

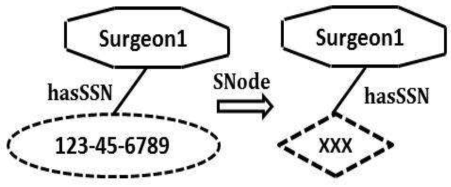
The SNode operation is essentially used to hide the object portion of sensitive triples, for *e.g.*, SSN values, Secret service agents, *etc.* As depicted by Algorithm 1, it takes as input the graph G_q, which needs to be sanitized before being shared with a user. In addition, it takes as input a $Pattern$ denoted by P, which specifies the sensitive data item that needs to be protected. The final argument $Sync$ is used as a flag to guide synchronization decision. The sensitive data item can be contained in a triple, or a set of triples represented by a predicate triple pattern TP' or a type graph pattern GP'. When the input $Pattern$ is a *triple*, the algorithm replaces the object portion of the triple with a sanitized value. If the object is a resource, then the sanitized value is also a resource and further, if the object is a data value/literal then the sanitized value is also a literal. When the input $Pattern$ is a predicate triple pattern or a type graph pattern, it is evaluated upon the input graph and the resulting subgraph triples are sanitized to protect their object values. The complexity of evaluating TP'/GP' is polynomial, as stated in [16], since only AND operation is involved and moreover, the number of triple patterns can be at most two. The result of evaluating TP'/GP' is a mapping set which can be converted into triples by appropriately substituting the input triple patterns. Note that, in case of GP' we obtain the result triple set by substituting the triple pattern other than type triple pattern.

Algorithm 1 SNode()

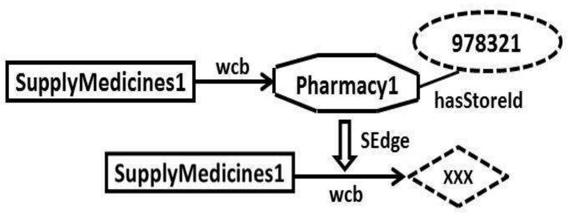
Input: Graph G_q, $Pattern\ P$, Boolean $Sync$

```
 1: $sensitiveTrpls = \emptyset, sanitizedTrpls = \emptyset, nodeSet = \emptyset,$
    $map = \emptyset, G_s = \emptyset$
 2: if $P$ is a triple $t$ then
 3:     if $t \in G_q$ then $G_s = t$ end if
 4: else
 5:     if $P$ is $TP'$ or $GP'$ then $G_s = eval(G_q, P)$ end if
        {$G_s$ is the result triple set from evaluation of $P$ on $G_q$}
 6: end if
 7: $G_q = G_q - G_s$
 8: for $t \in G_s$ do
 9:     if $map.contains(t.o)$ then $o' = map.get(t.o)$
        else $o' = genUniq(t.o)$ end if
10:     if $t.o \in I$ then $map.put(t.o, o')$; $nodeSet.add(t.o)$ end if
11:     $newTriple = createTriple(t.s, t.p, o')$
12:     $sanitizedTrpls += newTriple$
13: end for
14: if $Sync$ then
15:     for $n \in nodeSet$ do
16:         $G_n = triples\ in\ G_q\ containing\ n$
17:         for $t \in G_n$ do $sensitiveTrpls += t$;
            $sanitizedTrpls += synchronizeTrpl(t, map)$ end for
18:     end for
19: end if
20: $G_q = G_q - sensitiveTrpls$; $G_q = G_q \cup sanitizedTrpls$
```

The masking function *genUniq* takes a resource or literal value as input and returns a unique sanitized value. However, depending on the application/privacy requirements one can override *genUniq* with a suitable custom masking function. Our current implementation, supports other custom masking functions, for *e.g.*, a more appropriate sanitized value for SSN "123-45-6789" could be "XXX-XX-6789". However, a comprehensive coverage of such masking functions and their use has been left as a part of future work and we will continue to use masking function *genUniq* in the rest of the paper. Further, *map* is a data structure that stores pairs comprising a sensitive data item and its corresponding sanitized value. The operation *createTriple* creates a triple using the three input arguments. If the object of the triple is a resource, then it may be a part of other triples



(a) SNode



(b) SEdge

Figure 2: Sanitization Operations

as well. To sanitize such object resources (or perform synchronization), they are first collected in *nodeSet* (line 10) and then, all triples of G_q containing nodes from *nodeSet* are sanitized (lines 15 to 18). Synchronization is done when the boolean flag *Sync* is true (line 14) and it is performed using the function *synchronizeTrpl*, which takes a triple as input and updates the sensitive parts (only IRI's) of the triple with corresponding values from the *map* if any. Consequently, the overall complexity of the SNode algorithm is $O(n^2)$, where n is the number of triples in G_q. However, if the flag *Sync* is false then no synchronization is performed. Finally, a more sophisticated version of SNode, called Star, is presented in Appendix 9.2.

2.4 Edge Sanitization (SEdge)

This operation is designed to hide a relationship/link/edge between two nodes and alternatively to perform edge contraction, *i.e.*, to protect an edge along with its two nodes (where one of the nodes can be a data value). The example query Q5 using SEdge is illustrated with Figure 2(b).

```
Q5: SANITIZE Gq
    WHEREs {SEdge (Pharmacy1 hasStoreId 978321)} SYNC {Pharmacy1}

Q6: SANITIZE Gq
    WHEREs {SEdge (?s rdf:type Prescription . ?s hasPresId ?o)}
    SYNC{?s}
```

Algorithm 2 SEdge()
Input: Graph G_q, *Pattern P*, Boolean *SyncSub*, *SyncObj*

```
1: $sensitiveTrpls = \emptyset, sanitizedTrpls = \emptyset, nodeSet = \emptyset,$
   $map = \emptyset, G_s = \emptyset$
2: if $P$ is a triple $t$ then
3:     if $t \in G_q$ then $G_s = t$ end if
4: else
5:     if $P$ is $TP'$ or $GP'$ then $G_s = eval(G_q, P)$ end if
6: end if
7: $G_q = G_q - G_s$
8: for $t \in G_s$ do
9:     if $t.s \in I$ then $map.put(t.s, genUniq(t))$;
       if $SyncSub$ then $nodeSet.add(t.s)$ end if end if
10:    if $t.o \in I$ then $map.put(t.o, genUniq(t))$;
       if $SyncObj$ then $nodeSet.add(t.o)$ end if end if
11: end for
12: if $SyncSub \vee SyncObj$ then
13:     for $n \in nodeSet$ do
14:         $G_n$ = triples in $G_q$ containing $n$
15:         for $t \in G_n$ do $sensitiveTrpls += t$;
            $sanitizedTrpls += synchronizeTrpl(t, map)$ end for
16:     end for
17: end if
18: $G_q = G_q - sensitiveTrpls$; $G_q = G_q \cup sanitizedTrpls$
```

Algorithm 2 takes as input G_q and pattern P containing the sensitive data item that needs to be protected, which can be a triple or TP'/GP'. Additionally, it also takes boolean flags *SyncSub* and *SyncObj* which are used to determine the type of synchronization that needs to be performed. If one of the flags is true then only corresponding subject or object node of a sensitive edge are sanitized. However, if both the flags are true, then subject as well as the object node of a sensitive edge is sanitized. Since there are multiple ways in which synchronization can be performed in the case of SEdge, the appropriate arguments (variables) within the `SYNC` clause of the `SANITIZE` query form can be used to precisely state the type of synchronization desired. The default semantics of synchronization for SEdge operation is sanitization of both the nodes of the sensitive edge and is represented by keyword `SYNC` without any arguments. However, if synchronization clause is absent then no synchronization is performed.

When the input pattern P is TP' or GP', it is evaluated upon the input graph and the resulting subgraph triples need to be sanitized. This is the case in which one might want to sanitize a set of edges that connect subject or object resources belonging to some class. For example, Q6 tries to hide the edge or attribute `hasPresId` of the members of class `Prescription` and their respective values. The set G_s is computed in lines 2-6 in a similar manner as described in the SNode algorithm. Line number 7 accomplishes the task of hiding the link between the two nodes of a triple belonging to set G_s. Although the concrete triple is now deleted, the sensitive resources of the triple, its subject and object, may still be present as resources in other connected triples, *i.e.*, we may want to perform synchronization. For example, in Figure 2(b), there are two triples, SupplyMedicines1 wcb Pharmacy1 and Pharmacy1 hasStoreId 978321. The deletion of sensitive triple Pharmacy1 hasStoreId 978321, does not however, remove resource Pharmacy1 from the first triple: "SupplyMedicines1 wcb Pharmacy1". Now, it may be required by an application to sanitize all such connected triples (that contain the subject/object resource of t as their subject/object) by replacing the sensitive nodes with the corresponding sanitized value using the *map*, as shown in lines 13 to 16, which also leads to a O(n^2) complexity of the algorithm (where $n = |G_q|$). The meaning of *genUniq* and *synchronizeTrpl* remains the same. Further, *map* stores the subject and object resources of a triple with their corresponding sanitized value (the hash value of the triple, which effectively results in edge contraction).

2.5 Path Sanitization (SPath)

The SPath operation is designed to protect a path containing multiple nodes connected by edges. Query Q7 using SPath hides the provenance of `PatientFile4`, which is expressed using the regular expression `(PatientFile4 [wdf]+ ?o)`.

It is illustrated with Figure 1(b), which only shows the portion containing the sanitized subgraph as the remaining graph stays the same. The set of triples we get by evaluating the path pattern expression `(PatientFile4 [wdf]+ ?o)`, on G_q, comprises three triples: PatientFile4 wdf PatientFile3, PatientFile3 wdf PatientFile2 and PatientFile2 wdf PatientFile1. Unlike SEdge, where we perform edge contraction, in SPath we simply replace each s, p and o value with it's corresponding sanitized value to preserve connectivity, provenance definition and directed acyclic graph property of the provenance RDF graph. Please note that the operations SNode, SEdge and SPath provide a variety of techniques for sanitizing a variety of sensitive data items and together they form a sanitization language with rich expressivity. The default semantics of synchronization for SPath is represented using keyword `SYNC` without any arguments and is performed by consistently sanitizing all the subject and object nodes of sensitive triples representing the path. Synchronization is not performed when the corresponding clause is absent. These are the only two possible synchronization semantics that have been provided for SPath.

```
Q7: SANITIZE Gq WHEREs { SPath (PatientFile4 [wdf]+ ?o)} SYNC
```

Algorithm 3 SPath()

Input: Graph G_q, *PathPattern PP*, Boolean *Sync*

```
1: $sensitiveTrpls{=}\emptyset$, $sanitizedTrpls{=}\emptyset$, $PathTrpls{=}\emptyset$, $nodeSet{=}\emptyset$,
   $map{=}\emptyset$, $setFoundTrpls{=}\emptyset$
2: $PathTrpls = eval(G_q, PP)$
3: for $t \in PathTrpls$ do
4:     if $map.contains(t.s)$ then $s' = map.get(t.s)$
       else $s' = genUniq(t.s)$ end if
5:     if $map.contains(t.p)$ then $p' = map.get(t.p)$
       else $p' = genUniq(t.p)$ end if
6:     if $map.contains(t.o)$ then $o' = map.get(t.o)$
       else $o' = genUniq(t.o)$ end if
7:     if $t.s \in I$ then $map.put(t.s, s')$; $nodeSet.add(t.s)$ end if
8:     if $t.o \in I$ then $map.put(t.o, o')$; $nodeSet.add(t.o)$ end if
9:     $sanitizedTrpls+ = createTriple(s', p', o')$
10:    $G_q = G_q - t$
11: end for
12: if $Sync$ then
13:    for $n \in nodeSet$ do
14:        $G_n$ = triples in $G_q$ containing $n$
15:        for $t \in G_n$ do $sensitiveTrpls+ = t$;
           $sanitizedTrpls+ = synchronizeTrpl(t, map)$ end for
16:    end for
17: end if
18: $G_q = G_q - sensitiveTrpls$; $G_q = G_q \cup sanitizedTrpls$
```

The inputs to Algorithm 3 are graph G_q and a SPARQL *PathPattern*, *PP* and flag *Sync*. The aim of this algorithm is to firstly, identify the sensitive data item, which is a path or subgraph (in this case). The triples comprising the sensitive subgraph are evaluated in line 2, and deleted from the graph in line 10. Secondly, the algorithm stores pairs of original and corresponding sanitized resources in a *map* (lines 4, 5 and 6). The mapping helps one to perform consistent sanitization. A sanitized triple is constructed from the sanitized values of the original resources, s, p and o of the sensitive triple (line 9). Next, we perform synchronization in lines 13 to 16, which is again optional depending on whether the flag *Sync* is true or not. Finally, we delete the sensitive triples from G_q and add the sanitized triples to it (line 18). The overall complexity of this algorithm depends on the evaluation of a path pattern (done by function *eval*), which is shown to be intractable [17].

3. DENOTATIONAL SEMANTICS FOR SANITIZATION OPERATIONS

Pérez *et al.* provided set-based denotational semantics for the core set of SPARQL operations in [16], which was later adapted by the W3C standard specification of SPARQL [12]. We further extend SPARQL by providing denotational semantics for the sanitization operations. Denotational semantics is compositional, *i.e.*, the meaning of an expression can be derived from the meaning of its subexpressions. It is given in terms of, abstract syntax definition of the language, semantic algebras and the valuation functions. In the following, we give a brief introduction to denotational semantics, a detailed exposition of which can be found in [18]. We assume the reader is familiar with the compositional semantics of SPARQL [16].

- **Abstract Syntax** is specified as BNF grammar. A Syntax domain denotes a collection of values with common syntactic structure.
- **Semantic Algebra** consists of a semantic domain accompanied by a set of operations. The operations are given in terms of the following parameters: (i) **Functionality** of a function $f : D_1 \times D_2 \times \cdots \times D_n \to A$, implies f needs an argument from each domain D_1, $D_2, \cdots, D_n$ to produce an answer belonging to domain A. (ii) **Description** is generally given as an equational definition.
- **Valuation Function** maps elements of syntax domain to elements of semantic domain.

Note: (1) In the following, we use compound domains known as disjoint union, tagged union or sum. It is a form of union construction on sets that keeps the members of the respective sets R and S separate: $R + S = \{(zero, x) | \ x \in R\} \cup \{(one, y) | \ y \in S\}$. (2) The cases operation for $m \in R+S$ is defined as follows: cases (m) of $isR(x) \to ..x..| \ isS(y) \to ..y..$, which should be read as "if m is an element whose tag component is R and value component is x, then the answer is $..x..$ else if the tag component is S and the value component is y then the answer is $..y..$".

We now describe the denotational semantics for SPARQL sanitization operations. The Abstract Syntax, shown in Figure 3(a), is given in terms of syntax domains and BNF rules. The phrase $SExpr \in SanitizeOpExp$ indicates that $SanitizeOpExp$ is a syntax domain and $SExpr$ is a nonterminal that represents an arbitrary member of the domain. The meaning of the other phrase can be inferred similarly. The first BNF rule states that a sanitization query expression ($SExpr$) can be one of the three possible kinds, namely, *SNode*, *SEdge* or *SPath*. The second rule states that a *Pattern* can be a *Triple or* TP'*or* GP'. We then present semantic algebras for semantic domains TRIPLE, MAP and GRAPH in Figure 3(b). *Assumptions:* We use set semantics instead of bag semantics. Further, we present denotational semantics assuming the default synchronization semantics for SNode, SEdge and SPath operations. These simplifications make the exposition more comprehensible without compromising the core complexity of the problem.

The operation *updateObj* is defined over the semantic domain TRIPLE, which takes a triple and a resource/value as inputs and retains the first and second elements of the triple, represented by $t.s$ and $t.p$, as they are and replaces the triple's object with the resource/value. Note that, we will subsequently interpret the set union symbol $\cup$ of the semantic domain TRIPLE as disjoint union.

3(a) Abstract Syntax	
Syntax Domain:	$SExpr \in SanitizeOpExp,\ Pattern \in PatternExp$
BNF Rules:	$SExpr ::= SNode\ Pattern \mid SEdge\ Pattern \mid SPath\ PP$
	$Pattern ::= Triple \mid TP' \mid TTP\ AND\ TP'$

3(b) Semantic Algebras for TRIPLE, MAP and GRAPH

$Semantic\ Domain$: $TRIPLE = (I \cup B) \times I \times (I \cup B \cup L)$

$Operations$: $updateObj : TRIPLE \times (I \cup L) \rightarrow TRIPLE$

$\boldsymbol{updateObj}(t, r) = (t.s, t.p, r)$

$Semantic\ Domain$: $MAP = I \rightarrow I$

$Operations$: $newmap : MAP$ a newmap is an empty map, which maps all index arguments to null/error

$access : I \times MAP \rightarrow I$ $\boldsymbol{access}(i, m) = m(i)$

$genMap : (I + \mathcal{P}(I)) \rightarrow MAP$

$\boldsymbol{genMap}(r) = cases(r)\ of$

$isI(i) \rightarrow \{(i, i') \mid i' = genUniq(i)\} \mid is(\mathcal{P}(I))(S) \rightarrow \{(i, i') \mid i \in S\ \wedge i' = genUniq(i)\}$

$Semantic\ Domain$: $GRAPH = \{x \mid x\ is\ TRIPLE\}$

$Operations$:

$union, diff : GRAPH \times GRAPH \rightarrow GRAPH$	$maskSPO : GRAPH \rightarrow GRAPH$
$\boldsymbol{union}(g_1, g_2) = \{t \mid t \in g_1 \vee t \in g_2\}$	$\boldsymbol{maskSPO}(g) = \{(s, p, o) \mid t \in g \wedge (s = genUniq(t.s)$
$\boldsymbol{diff}(g_1, g_2) = \{t \mid t \in g_1 \wedge t \notin g_2\}$	$\wedge\ p = genUniq(t.p) \wedge o = genUniq(t.o))\}$
$getGraph : \Omega \times TP' \rightarrow GRAPH$	$genSO : GRAPH \rightarrow \mathcal{P}(I)$
$\boldsymbol{getGraph}(\Omega_s, tp) = \{\mu(tp) \mid \mu \in \Omega_s\}$	$\boldsymbol{genSO}(g) = \{n \mid t \in g \wedge n \in I \wedge (n = t.s \vee n = t.o)\}$

$sensitive : GRAPH \times (I + \mathcal{P}(I)) \rightarrow GRAPH$

$\boldsymbol{sensitive}(g, r) = cases(r)\ of\ isI(i) \rightarrow \{t \mid t \in g \wedge (t.s = i \vee t.p = i \vee t.o = i)\}$

$\mid is(\mathcal{P}(I))(R) \rightarrow \{t \mid t \in g \wedge (t.s \in R \vee t.p \in R \vee t.o \in R)\}$

$sanitize : GRAPH \times (I + \mathcal{P}(I)) \times MAP \rightarrow GRAPH$

$\boldsymbol{sanitize}(g, r, m) = cases(r)\ of$

$isI(i) \rightarrow \{(t.s, t.p, t.o) \mid t \in g \wedge (\text{if } t.s = i \text{ then } t.s = m(t.s))$

$\wedge(\text{if } t.p = i \text{ then } t.p = m(t.p)) \wedge (\text{if } t.o = i \text{ then } t.o = m(t.o))\} \mid$

$is(\mathcal{P}(I))(R) \rightarrow \{(t.s, t.p, t.o) \mid t \in g \wedge (\text{if } t.s \in R \text{ then } t.s = m(t.s))$

$\wedge(\text{if } t.p \in R \text{ then } t.p = m(t.p)) \wedge (\text{if } t.o \in R \text{ then } t.o = m(t.o))\}$

$sNode : GRAPH \times (TRIPLE + GRAPH) \rightarrow GRAPH$

$\boldsymbol{sNode}(g, r) = cases(r)\ of$

$\boldsymbol{isTRIPLE(t)} \rightarrow \boldsymbol{(}cases(t.o)\ of$

$isL(o) \rightarrow Let\ t' = updateObj(t, genUniq(t.o))\ in\ union(diff(g, \{t\}), \{t'\}) \mid$

$isI(o) \rightarrow Let\ g_s = sensitive(g, t.o) \wedge m = genMap(t.o)\ in\ union(diff(g, g_s), sanitize(g_s, t.o, m))\boldsymbol{)}$

$\mid \boldsymbol{isGRAPH(gx)} \rightarrow Let\ t \in gx,\ gx' = diff(gx, \{t\}),\ g' = sNode(g, t)\ in\ sNode(g', gx')$

$sEdge : GRAPH \times (TRIPLE + GRAPH) \rightarrow GRAPH$

$\boldsymbol{sEdge}(g, r) = cases(r)\ of$

$\boldsymbol{isTRIPLE(t)} \rightarrow \boldsymbol{(}cases(t.o)\ of$

$isL(o) \rightarrow (Let\ g' = diff(g, \{t\})$

$in\ (Let\ g_s = sensitive(g', t.s)\ and\ m = genMap(t.s)\ in\ union(diff(g', g_s), sanitize(g_s, t.s, m)))) \mid$

$isI(o) \rightarrow (Let\ g' = diff(g, \{t\})\ and\ nl = \{t.s, t.o\}$

$in\ (Let\ g_s = sensitive(g', nl)\ and\ m = genMap(nl)\ \ in\ union(diff(g', g_s), sanitize(g_s, nl, m))))\boldsymbol{)}$

$\mid \boldsymbol{isGRAPH(gx)} \rightarrow Let\ t \in gx,\ gx' = diff(gx, \{t\}),\ g' = sEdge(g, t)\ in\ sEdge(g', gx')$

$sPath : GRAPH \times GRAPH \rightarrow GRAPH$

$\boldsymbol{sPath}(g, gx) = Let\ g' = diff(g, gx)\ and\ ml = genSO(gx)\ and\ gx' = maskSPO(gx)\ in$

$Let\ g_s = sensitive(g', ml)\ and\ m_1 = genMap(ml)\ in$

$union(diff(g', g_s), union(sanitize(g_s, ml, m_1), gx')$

3(c) Valuation Functions	
S:	$SanitizeOpExp \times GRAPH \rightarrow (TRIPLE \cup GRAPH)$
	$\mathbf{S}\ [\![SNode\ Pattern]\!]\ (G) = sNode(G, \mathbf{P}\ [\![Pattern]\!]\ (G))$
	$\mathbf{S}\ [\![SEdge\ Pattern]\!]\ (G) = sEdge(G, \mathbf{P}\ [\![Pattern]\!]\ (G))$
	$\mathbf{S}\ [\![SPath\ PP]\!]\ (G) = sPath(G,\ pathSubGraph(\ [\![PP]\!]_{\ G}, PP))$
P:	$PatternExp \times GRAPH \rightarrow (TRIPLE \cup GRAPH)$
	$\mathbf{P}\ [\![Triple]\!]\ (G) = Triple$
	$\mathbf{P}\ [\![TP']\!]\ (G) = getGraph(\ [\![TP']\!]_{\ G}, TP')$
	$\mathbf{P}\ [\![TTP\ AND\ TP']\!]\ (G) = getGraph(\ [\![TTP]\!]_{\ G} \bowtie\ [\![TP']\!]_{\ G}, TP')$

Figure 3: Denotational Semantics for Sanitization Operations

The semantic domain MAP uses elements of set I as an index to access its contents. It is a function that maps elements of I to unique elements of set I, where these elements are the sanitized versions of the original resources. The operation *access* fetches the value stored in map m for element i and is denoted by $m(i)$. The operation $genMap$ takes as argument an element that is either from domain I or from power set of I denoted by $\mathcal{P}(I)$ and returns a corresponding MAP. If the input argument is a resource i, it returns a MAP in which i is paired with unique sanitized value i', which is generated by function $genUniq$. However, if the input argument is a subset of I, then the function returns a MAP comprising pairs of resources belonging to the subset along with their sanitized values generated by $genUniq$. The operation *sanitize*, sanitizes the triples in the input graph (g) by first checking whether any of its components (s, p, o) match with the sensitive input resource (r) and if it does match then it is updated using the corresponding sanitized value from the map (m). The function $maskSPO$ masks all the three components of a triple belonging to the input graph by replacing them with unique values generated by $genUniq$.

The operation SNode is defined based on two cases namely, base case and recursive case, which in turn are based on the input data item being sanitized. When the input data item to be sanitized is a triple then two sub cases follow from this base case, which depend on whether the object of the triple is a literal or an IRI. When the object is a literal the original triple is deleted from the graph and a sanitized version is added to the graph in replacement. However, when the object value is an IRI then the sensitive triples are computed, which are those that contain the IRI. These triples are deleted from the original graph and a sanitized version of them is added back to the remaining graph. The recursive case is defined for an input of type graph in contrast to the base case where the input was a triple. In the recursive case the triples belonging to gx need to be sanitized. A triple t is selected from set gx and sanitized within graph g to obtain g'. Then the triple is deleted from gx and gx' is obtained. Finally, a recursive call to $sNode$ is made using inputs gx' and g'. The semantics of operations $sNode$, $sEdge$ and $sPath$ are self-explanatory and similar to algorithms 1, 2 and 3 respectively.

Figure 3(c) gives a valuation function for each syntax domain along with a set of equations, one per option of the corresponding BNF rule for that domain. The domain of valuation function is the set of derivation trees of a language. The meaning of a derivation tree is determined by determining the meaning of the subtrees. For example, the functionality for S states that, S takes a member of the syntax domain *SanitizeOpExp* and a graph (sensitive graph) as input and returns a graph (sanitized graph) as output. The semantics of $[\![TTP]\!]_G \bowtie [\![TP']\!]_G$ is the same as defined in [16], thus leveraging the compositional nature of denotational semantics. Similarly, the semantics for $[\![PP]\!]_G$ has been adapted from [12]. The function *pathSubGraph*, like function *getGraph* renders the output of $[\![PP]\!]_G$ as a set of triples (graph).

4. PROTOTYPE ARCHITECTURE AND EXPERIMENTAL EVALUATION

In this section, we present a prototype system architecture, shown in Figure 5. A querying user submits a SPARQL query to the user interface. The user interface first transforms the query into an appropriate construct query, q', and forwards it to the SPARQL query engine and also forwards the query q along with the user information to the policy evaluator as depicted by the arrow labeled (2b). The construct query q' is used in order to obtain the subgraph that is being accessed by the user, which we will refer to as Graph Of Interest (GOI). It is obtained using the construct query by instantiating the original query's triple patterns as illustrated in Figure 4. Once the SPARQL query engine receives the query q', it performs computation on the RDF store and returns the resulting subgraph G_q (the GOI) to the policy evaluator depicted by the arrow labeled (5). The policy evaluator now computes a set of applicable policies from the original policy set based on the three aforementioned inputs it receives (steps 2b and 5). Subsequently, the policies are parsed and transformed into corresponding SPARQL sanitization queries such as SNode, SEgde and SPath. Once the list of sanitization queries is obtained, we then apply them on to the RDF graph and return the sanitized graph to the user interface. We now apply the user query q to G'_q (sanitized GOI) and return the result (R) to the user.

```
q) SELECT ?n ?o ?e
   WHERE { ?s hasName ?n. ?s hasSSN ?o . ?s hasEmail ?e}

q') CONSTRUCT {?s hasName ?n. ?s hasSSN ?o . ?s hasEmail ?e}
    WHERE {?s hasName ?n. ?s hasSSN ?o . ?s hasEmail ?e}
```

Figure 4: User query and it's Construct query

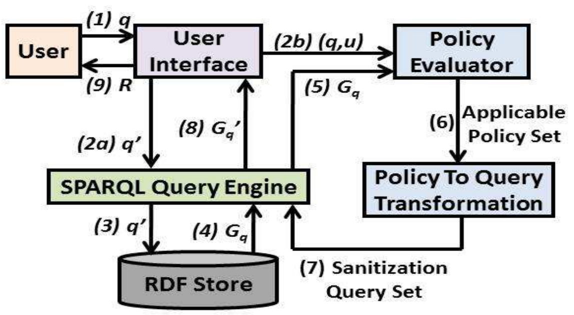


Figure 5: Prototype System Architecture

In the case of healthcare provenance scenario, a querying user can be an insurance agent, researcher conducting epidemic studies or someone who may want to publish the data online. The policy author needs to carefully author policies to protect sensitive patient information. The policy language we use is a modification of the XML-based language proposed in [19]. A sample policy that uses this language to protect SSN values of all individuals is given in Figure 6, along with its translation to equivalent sanitization query. The tag *target* is used to specify the set of users and data items (belonging to given datasets) to which the policy is applicable. The tag, *operationType*, specifies the type of sanitization operation that is needed to protect the sensitive data item. The tag, *sanitizationPatterns*, specifies the sensitive data item, expressed in the form of a triple pattern or graph pattern. The tag *effect* specifies the intended consequence of a policy when it evaluates to true. We now

```
<policy id="1125">
 <target>
   <user>Insurance Agent</user>
   <dataSet>Gq</dataSet>
   <operationType>SNode</operationType>
   <sanitizationPatterns><pattern>
     <s>?s</s>
     <p>hasSSN</p>
     <o>?o</o>
   </pattern></sanitizationPatterns>
 </target>
 <effect>deny</effect>
</policy>
```

```
SANITIZE Gq WHEREs {SNode (?s hasSSN ?o)}
```

Figure 6: Example Policy and Equivalent Query

Dataset	Healthcare (≈ 1K)	Twitter (≈ 3M)	SEC (≈ 1.8M)
Nodes	297	1705116	866627
Edges	979	2850579	1813175
Predicates	50	52	16

Figure 7: Experimental Datasets

present the results of an experiment that was conducted to validate the effectiveness of the sanitization algorithms.

Experimental Setup: We conducted the experiment using an Intel Core i7 3610QM Processor with a 750GB hard drive and 8GB main memory. Further, JRE v1.7.0_21 as the Java engine and Jena v2.10.0, ARQ v2.10.0 and Gleen v0.6.1 were used for development of the experimental code.

Experimental Datasets and Queries: The sanitization algorithms we described earlier are equally applicable to graphs generated by processing user queries and for those representing entire datasets that need to be published online. To demonstrate this notion, we performed the experiment using the following datasets: (i) **Healthcare Scenario**: We used six variants of the healthcare scenario presented earlier, each containing between ≈ 600-1000 triples. This dataset captures the applicability of the sanitization algorithms to query result graphs. (ii) **Twitter**: We used five variants of a Twitter subgraph containing between ≈ 3000-7500 users (≈ 1M-3M triples). (iii) **U.S. Securities and Exchange Commission (SEC)**: We used five variants of the SEC RDF dataset[1] containing between ≈ 600K-1.8M triples. Datasets (ii) and (iii) capture the applicability of the sanitization algorithms to graphs representing entire datasets. Figure 7 shows the composition of each dataset, where the value in parentheses denotes the size of the largest variant of that dataset. Table 2 lists the queries that were executed on the corresponding datasets for evaluation. Note that, G_q denotes the particular dataset variant used for query evaluation, while "Affected Triples" denotes the number of triples that were sanitized or synchronized as a part of the given operation for the largest variant of the dataset. As one can observe, a larger number of affected triples entails a longer sanitization time (Figure 8).

Experiment with varying dataset sizes: The goal of this experiment was to measure the performance, in terms of overall sanitization time, of the different sanitization operations when applied to increasing dataset sizes. As shown in Figure 8, the time to execute the operations increases as dataset size increases. An operation with limited scope, *viz.* SNode, requires a shorter execution time when compared with complex operations such as SEdge and SPath. The sanitization time depends on many factors such as dataset size, number of affected triples, triple store implementation, connectivity of a node, *etc.* In conclusion, the sanitization overhead is reasonable for it to be used in practice.

[1]http://www.rdfabout.com/demo/sec/

Table 2: Queries used in Experimental Evaluation

Dataset	Queries	Affected Triples
Health care	SANITIZE G_q WHEREs { SNode(?s hasHealthcareId ?o) }	1
	SANITIZE G_q WHEREs { SEdge(?s type Patient . ?s hasSSN ?p) }	16
	SANITIZE G_q WHEREs { SPath(Prescription1Filled "[wgb]/[wcb]" ?o) }	20
Twitter	SANITIZE G_q WHEREs { SNode(?s Twitter_Id ?o) }	7286
	SANITIZE G_q WHEREs { SEdge(?s Tweet_Text ?o) }	37300
	SANITIZE G_q WHEREs { SPath(User63752681 "[Has_Tweet]/[Reply_To_User]" ?o) }	92
SEC	SANITIZE G_q WHEREs { SNode(?s name ?o) }	144179
	SANITIZE G_q WHEREs { SEdge(?s Street ?p) }	529176
	SANITIZE G_q WHEREs { SPath(?s "[hasRelation]/[type]" TenPercentOwnerRelation) }	202283

5. RELATED WORK

In [20], the authors address the problem of RDF data sanitization. This work essentially tries to address - what data needs to be sanitized? and proposes a solution based on the open world assumption. In contrast, our work addresses the problem of how to perform sanitization and thus complements the work done in [20]. A SPARQL update language is presented in [12] and [21], and further, [21] also provides operational semantics. In contrast, we present a language that not only allows updates but also consistent sanitization or synchronization. We also present denotational semantics, which blends well with the existing compositional semantics of SPARQL. In [22], authors presented a graph grammar based approach to secure RDF-based provenance data by performing redaction. However, the redaction mechanism was built at the application level and addressed only OPM-based provenance graphs. Additionally, the user is required to manually suggest a replacement resource and embedding connections. Since the number of resources in a provenance (or RDF) graph is exponential in the number of nodes [19], the manual embedding procedure is limited and not scalable. The approach presented in this paper overcomes this shortcoming by providing masking function *genUniq*, which

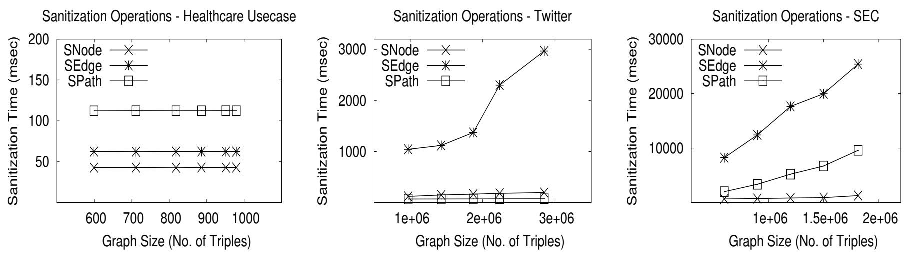


Figure 8: Performance of various types of sanitization operations on different datasets

enables automatic generation of a replacement, and a *map* data structure that enables consistent automatic embedding or synchronization of the replacement resource within the remaining graph. Additionally, our sanitization mechanism, which is built at the query language level, is general and applicable to any domain including OPM-based provenance data. More precisely, it is the first of its kind to address general purpose RDF sanitization, which primarily uses graph transformation to selectively mask sensitive data, thus enabling fine-grained RDF access control.

6. CONCLUSIONS

In this paper, we made an initial attempt to provide a fundamental, formal framework for securing RDF graphs through a set of sanitization operations. These graph transformation operations are built as an extension to the SPARQL query language as a new query form called `SANITIZE`. As a result, any system with RDF data, can build more sophisticated security, privacy and anonymization features using it. SPARQL Sanitize provides a valuable layer of security, which not only promotes integration of RDF datasets but also makes it practical and takes Semantic Web a step closer towards the vision of a global database. We illustrated the utility of our approach with a healthcare provenance scenario and showed how one can secure the data using the sanitization infrastructure provided by us. As a part of the future work, we would like to further fine tune the sanitization operations with pluggable masking routines to maintain a greater amount of control over data exposure and in turn meet a broad range of application requirements.

7. ACKNOWLEDGMENTS

This work was partially supported by National Institutes of Health Grants 1R0-1LM009989 and 1R01HG006844, National Science Foundation (NSF) Grants Career-CNS-0845803, CNS-0964350, CNS-1016343, CNS-1111529, CNS-1228198 and Army Research Office Grant W911NF-12-1-0558.

8. REFERENCES

[1] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. Database Access Control and Privacy: Is there a common ground? In *CIDR*, 2011.

[2] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *Computer*, 29(2):38–47, February 1996.

[3] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2), 2001.

[4] M. Bishop. *Introduction to Computer Security.* Addison-Wesley Professional, 2004.

[5] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[6] C. Dwork. Differential privacy. *Automata, languages and programming*, 2006.

[7] Oracle. Fine-Grained Access Control for RDF Data. `http://goo.gl/WJSNB`.

[8] L. Kagal, T. W. Finin, and A. Joshi. A Policy Based Approach to Security for the Semantic Web. In *ISWC*, 2003.

[9] T. W. Finin, A. Joshi, L. Kagal, J. Niu, R. S. Sandhu, W. H. Winsborough, and B. M. Thuraisingham. R*OWL*BAC: Representing Role Based Access Control in *OWL*. In *SACMAT*, 2008.

[10] J. Hollenbach, J. Presbrey, and T. Berners-Lee. Using RDF Metadata To Enable Access Control on the Social Semantic Web. In *CK2009*, volume 514, 2009.

[11] B. Carminati, E. Ferrari, R. Heatherly, M. Kantarcioglu, and B. Thuraisingham. A semantic web based framework for social network access control. In *SACMAT*, 2009.

[12] S. H. Garlik, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 Query Language. `http://www.w3.org/TR/sparql11-query/`.

[13] O. Lassila, R. R. Swick, and World Wide Web Consortium. Resource Description Framework (RDF) Model and Syntax Specification, 1998.

[14] L. Moreau, B. Clifford, and J. Freire *et. al.* The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems (FGCS)*, 27, 2011.

[15] O. Hartig and J. Zhao. Provenance Vocabulary Core Ontology Specification, 2010.

[16] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. In *ISWC*, 2006.

[17] M. Arenas, S. Conca, and J. Pérez. Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *WWW*, 2012.

[18] D. A. Schmidt. *Denotational semantics: A methodology for language development.* William C. Brown Publishers, Dubuque, IA, USA, 1986.

[19] T. Cadenhead, V. Khadilkar, M. Kantarcioglu, and B. Thuraisingham. A Language for Provenance Access Control. In *CODASPY*. ACM, 2011.

[20] M. Bishop, J. Cummins, S. Peisert, A. Singh, B. Bhumiratana, and D. A. Agarwal. Relationships and Data Sanitization: A Study in Scarlet. In *NSPW*, 2010.

[21] R. Horne, V. Sassone, and N. Gibbins. Operational Semantics for SPARQL Update. In *JIST*, 2011.

[22] T. Cadenhead, V. Khadilkar, M. Kantarcioglu, and B. Thuraisingham. Transforming Provenance using Redaction. In *SACMAT*, 2011.

9. APPENDIX

9.1 SPARQL UPDATE vs SANITIZE

The intent of the SPARQL UPDATE operation as its name suggests is to refresh the truth value, and is designed for that purpose by using Insertion and/or Deletion. It is possible to perform updates manually as the new value to update with is available. However, when one needs to conceal large amounts of sensitive data, the following questions arise - What should be the replacement value(s)? How to manage the magnitude (possibly millions) of replacements manually? How to preserve consistency or perform synchronization? How to perform updates securely with minimal manual intervention? As the number of replacements can be significant, the manual replacement strategy will neither scale nor be secure. Further, the semantics of the CONSTRUCT clause is to construct a new graph based on certain constraints, whereas the semantics of SANITIZE is to modify a given graph. Unlike CONSTRUCT operation, it does not return a graph. Since the intent of our sanitization operations is to secure and protect sensitive or identifying information in RDF graphs, it is designed accordingly with automatic masking and synchronization. None of the functionality offered by SPARQL shares the same intent, and is therefore not specifically designed or suitable for it. We do not want to extend or modify the existing operators/constructs as we do not want to alter their original semantics and overload them with this new functionality.

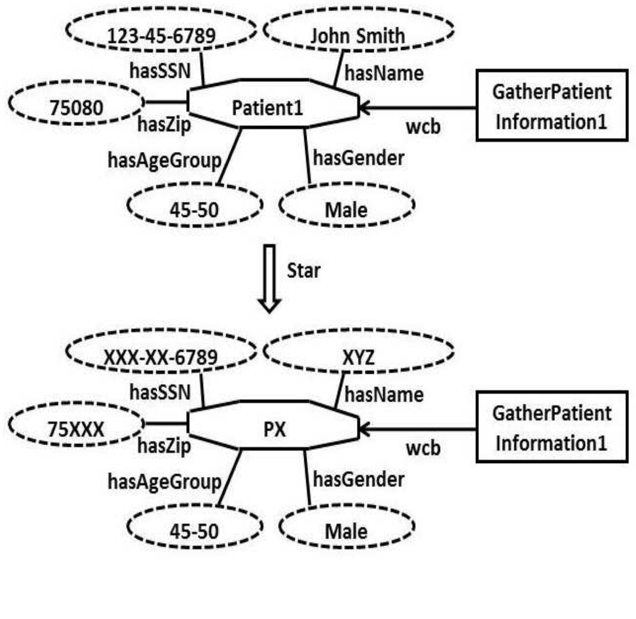


Figure 9: Star Sanitization Operation

9.2 Sanitize Node with identifying attributes

We now present a more sophisticated version of SNode operation and refer to it as the Star operation. The operation is named Star as the pictorial representation of a node and the set of identifying attributes represents a star shape. The intent of the Star operation is to mask a set of identifying attributes of a node or to mask a node along with its identifying attributes. For *e.g.*, one may want to hide identifying attributes such as SSN, Name, Employee ID (eid), *etc.*, of persons of type physician. Query Q8, describes the use of Star operation to perform sanitization. Unlike the previous sanitization operations, the Star operation takes two sets of arguments. The LHS set of arguments can be one among the following: $Null$, TTP, TP', GP'. The triple patterns used in the LHS enable one to identify the sensitive node/IRI that we need to access. The RHS of the Star operation comprises a list. The head of the list is either a node/IRI (for *e.g.* Patient1) or a variable (for *e.g.* $?s$) representing a set of IRI's. The rest of the list is the set of attributes or datatype predicates whose value needs to be masked for the given node (head of RHS list). The example query Q8 sanitizes the attribute values SSN, Name and Id of all physicians.

```
Q8: SANITIZE Gq
    WHEREs {(?s type Physician) Star (?s hasSSN hasName hasId)}

Q9: SANITIZE Gq
    WHEREs {Star (Patient1 hasSSN hasName hasZip)} SYNC{Patient1}

Q10: SANITIZE Gq
     WHEREs {(?s hasSSN ?o) Star (?s hasSSN hasName)}

Q11: SANITIZE Gq
     WHEREs {(?s type Surgery . ?s wcb ?o) Star (?o hasSSN hasName)}
     SYNC  {?o}
```

Algorithm 4 STAR()

Input: Graph G_q, *Pattern lhs*, List *rhs*, Boolean *Sync*

```
1: $sensitiveTrpls = \emptyset, sanitizedTrpls = \emptyset, map = \emptyset, nodeSet = \emptyset$
2: $nodeSet = getNodeSet(Gq, lhs, rhs.head)$ {sensitive node set computed using access pattern lhs and head of the list rhs}
3: for $n \in nodeSet$ do
4:     $sensitiveTrpls$ = set of triples containing $n$ from $G_q$
5:     $sensitiveTrpls1 = \emptyset; sensitiveTrpls2 = \emptyset$
6:     for $t \in sensitiveTrpls$ do
7:         if $t.s == n \wedge rhs.contains(t.p)$ then
8:             $sensitiveTrpls1 += t$ {Triples containing $n$ and a predicate from $rhs$}
9:         else $sensitiveTrpls2 += t$ {Other triples containing $n$}
10:    end for
11:    $s' = genUniq(n); map.put(n, s')$
12:    for $t \in sensitiveTrpls1$ do
13:        $o' = genUniq(t.o)$
14:        if $Sync$ then
15:            $newTriple = createTriple(s', t.p, o')$
16:            $sanitizedTrpls += newTriple$
17:        else $newTriple = createTriple(t.s, t.p, o')$
18:            $sanitizedTrpls += newTriple$ end if
19:    end for
20:    if $Sync$ then
21:        for $t \in sensitiveTrpls2$ do
22:            $sanitizedTrpls += synchronizeTrpl(t, map)$
23:        end for
24:    end if
25: end for
26: $G_q = G_q - sensitiveTrpls;\ G_q = G_q \cup sanitizedTrpls$
```

We now illustrate other ways of employing the Star operation. If we know a specific node/IRI whose attributes we want to sanitize then we do not need an access pattern on the LHS. Query Q9, shown in Figure 9, is used to sanitize `Patient1` along with the identifying attributes such as SSN, Name and Zip. The node, `Patient1`, is consistently sanitized (or synchronized) using the synchronization clause. In Q10, we identify a set of nodes or IRI's using TTP. The set of sensitive nodes whose attributes need to be masked can easily be computed using TTP and the first element of RHS list. Once the set of sensitive nodes has been computed we then sanitize them along with their identifying attribute values (SSN, Name). Query Q11 is more expressive as it uses GP' as the access pattern on the LHS to identify the set of sensitive resources which need to be sanitized along with the attribute values. The query masks surgeons, who have performed surgeries, along with attributes SSN and Name. The semantics of synchronization for the Star operation is as follows: If the synchronization clause is absent then only identifying attributes of a node are hidden. However, if the synchronization clause is present then it must specify the Node/IRI/Variable as an argument to the keyword `SYNC`. In this case, the sensitive node is consistently sanitized along with the sensitive identifying attributes as shown in Figure 9.

Algorithm 4, takes as input, G_q, an access pattern lhs, a list rhs and boolean flag $Sync$. It first computes the set of sensitive nodes in line 2 and stores it in $nodeSet$. Next, we iterate over the nodes in $nodeSet$ and compute two sets of sensitive triples, namely $sensitiveTrpls1$ and $sensitiveTrpls2$, which contain triples containing node n and a predicate present in list rhs and triples containing n and a predicate not present in list rhs respectively (lines 8 and 9). The triples of set $sensitiveTrpls1$ are sanitized by masking the subject as well as object values (lines 15-16) if the $Sync$ flag is true, otherwise only the object values are sanitized (lines 17-18). If $Sync$ is true then synchronization is also performed on triples belonging to $sensitiveTrpls2$ (lines 20-24).