Privacy Enforcement for Distributed Healthcare Queries

Michael Siegenthaler
Dept. of Computer Science
Cornell University
msiegen@cs.cornell.edu

Ken Birman
Dept. of Computer Science
Cornell University
ken@cs.cornell.edu

***Abstract*—In the healthcare industry and others, sensitive private information must be stored and shared between various organizations in the course of running their business. We have developed an architecture in which distributed data can be queried as if it resided in a single centralized database, while revealing minimal information beyond the answer to the query. In this paper we review the architecture and show how queries can be filtered to enforce user-specified privacy policies. We present a system for tracking information flow that is flexible enough to permit revealing sensitive data to those who have a need to know, while limiting the amount of useful information that can be obtained by a less-than-honest participant.**

I. Introduction

It has become commonplace to store patient health records in electronic databases, and great benefits in productivity and quality of care have been observed within organizations that make effective use of pervasive computing. The next logical step is to support sharing data across distinct organizations, yet this must be done carefully due to the sensitive nature of patient data. Some approaches have pushed data into a trusted, centralized repository[1][2], while ours retains the data at the individual organizations that produced it but allows queries to treat the distributed system as if it were centralized. Our approach requires a security architecture that allows certain queries and disallows others according to a specified policy, and in this paper we present how such an architecture might be designed.

A typical query in this environment would be something like a drug interaction check performed by a pharmacist. Suppose that the pharmacist has a list of drugs or health conditions which, if they appear in the patient's history, might raise a specific alert with regard to the medication currently being dispensed. The patient's health record is distributed across various *data owners*, which might include the primary care provider, several specialists, and a data repository from a home health monitoring device. A query against this distributed health record should check for a potential conflict, and return a yes/no answer, but should not reveal any of the source data unless the patient subsequently authorizes the pharmacist to investigate the nature of the conflict. More concretely, our system supports the following properties:

(a) *Data privacy*: The query asker learns only the answer to the query, and not any data used to compute it.
(b) *Query privacy*: The data owner does not learn the query, only that a query was performed against a particular user's information.
(c) *Anonymous communication*: Query askers and data owners do not know who the opposite party is.

Naturally, data privacy is desirable so that the pharmacist does not learn more than he needs to know about the patient's history. Query privacy is important because in some cases the query itself might reveal a piece of sensitive information to one of the data owners; for example a query for an HIV test result might indicate that the patient had a reason for wanting to be tested. Along the same lines, anonymous communication is needed because private information might be inferred from the knowledge that a record for the patient exists at a particular organization, for instance an AIDS clinic.

We have developed a system in which an SQL-like query, written as if it were for a logically centralized database, gets broken into pieces to be executed at the various data owners. The intermediate results are collected at a third party in order to compute the final result and return it to the query asker. This third party is assumed to be honest but curious; it can be trusted to execute the query protocol correctly but may misuse

This work was supported, in part, by the Institute for Information Infrastructure Protection (I3P) and the National Science Foundation.

any private data it learns. For this reason the identities of the patient and of the query asker and data owners are kept secret from the third party; it only sees the relevant medical data but cannot do anything useful with it due to the lack of context. The third party is thus called a *blind comparer*. As an additional level of protection, the system might periodically execute fake queries, so that the blind comparer does not even know whether the data it is seeing at any given time is real.

Data privacy only holds to the extent that the answer to a query does not reveal the data used to compute it. Clearly, there must be a mechanism in place to decide which queries should be allowed based on a user-specified policy. Our system quantifies the privacy leakage that would result from a particular query, and uses a currency-like system to make the query asker pay for the right to know the answer. A more revealing query has a higher cost. The cost is not one of real-world money, but of symbolic tokens which are supplied by the system to organizations that are authorized to perform queries. Tokens are issued according to a budget that takes into account each organization's business needs and gives it enough to execute the required queries to fulfill those needs, but does not leave a significant excess to be used on extraneous queries.

It would be unthinkable to impose security constraints so strict that an emergency room doctor does not have access to vital information for a patient who has been injured during distant travel from his home hospital and primary care provider. At the same time, there are less critical situations in which limited information should be revealed because the data user, while authorized, may not be trusted to guard the information as stringently as the data owner [3]. Our contribution is a technology that enables such heterogeneous use-scenarios by soft enforcement of policy that is configurable according to business needs.

In section II we review prior work in the area of database privacy. Section III then provides some details about our architecture, which is covered more fully in [4]. In section IV we present our security architecture, including the mechanics of how a policy is enforced.

II. Related Work

A standard approach to enforcing privacy in databases is fine grained access control (FGAC), for example the implementation by Agrawal et al. [5], which has the ability to enforce a P3P [6] privacy policy. There are two basic approaches to FGAC; either the query may be augmented with an additional WHERE clause to ensure that only allowable data is selected [7], or a new view may be created which only contains allowable data. Zhu et al. [8] point out that the former method can be tricked into leaking information when the full set of SQL operations are permitted, whereas the latter is safe but comes with a higher performance penalty if many clients are each subject to a different policy.

Simply preventing access to private fields, however, is not enough; many queries, including our drug interaction check example, require the value of some private field to be used in computing an answer which is not considered private. The answer inevitably reveals something about the private fields used to compute it. Hsu et al. [9] shows how to quantify this inherent privacy leakage. Konduri et al. [10] additionally addresses the scenario where multiple sequential queries might leak information that a single query could not. Both of these works are relevant to our architecture because the cost of a query should be chosen reflect the information leakage.

If a query is disallowed do to a tight dependence on some private data, it may be desirable to provide the closest allowable answer which omits the private data in question. Ding et al. [11] show how to transform the underlying data into a form that does not cause information leakage, even in the case where simply omitting a piece of data is not sufficient because it may be inferred from a combination of other data. Their approach specifically deals with the publishing of several related XML documents, but would apply for a certain class of relational database schema as well.

Work also exists on how to partition private data across multiple semi-trusted database providers such that any one of them does not have access to the private data [12]. Although this is not essential in our current usage model where data owners are themselves responsible for ensuring the persistence and availability of their share of a patient's medical record, it opens the door to outsourcing part of the data owner's role in a secure manner, and could potentially also be used to boost availability by mirroring data to semi-trusted hosts.

In this paper, we leverage the prior work, but found that prior solutions aren't enough to solve our distributed problem. Accordingly, we explore the additional mechanisms required to protect privacy in our distributed database architecture.

III. Query Execution

The process of answering a query is divided into two phases, a *global search* followed by the actual *query execution*. A patient is identified by some globally unique

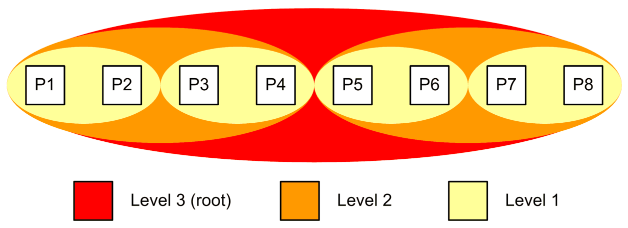

Fig. 1. Providers organize into a hierarchy of groups

identifier (UID), which might be a social security number or a combination of name and date of birth. Since a third party is involved in query execution to hide the identities of the query asker and data owner from each other, it is undesirable to use the UID in connection with sensitive data. Further, the query asker does not (and should not) know which other organizations in the system have records for a particular UID. The global search converts a UID into a set of *data handles*, each of which is a reference to a record stored at some data owner and provides a way to anonymously send a message to that data owner, but does not by itself reveal who the data owner is nor permit any observer other than the owner to recover the associated UID.

Global search works by organizing providers into a hierarchy of groups, as shown in figure 1, and building an index using Bloom filters [13] of the patients whose records appear within a provider group. A Bloom filter is a space-efficient mechanism for tracking set membership, and has the desirable privacy property that a list of members cannot be recovered from the set representation. Only two operations are supported: add a member, or test whether a specified member belongs to the set. The test operation sometimes gives false positives, which helps to ensure privacy because each patient will appear to have records in many different locations, only a subset of which actually exist. A search query starts at the root and traverses the tree of groups, returning a data handle at any point where a true match is found. The data handles returned by the search are onion skin routes [14], which can be used to route a message over a multi-hop path that conceals the destination from the origin and vice versa. For the full details on this technique, see our prior publication [4].

Once a set of data handles for the patient in question is known, query execution can proceed. Consider again the example of the drug interaction query. This may be written as follows:

```
SELECT EXISTS (
  SELECT * FROM conflicts
  CROSS JOIN nonces
  INNER JOIN remote(drug_history)
  ON nonces.nonce = drug_history.nonce
  WHERE conflicts.drug = drug_history.drug
);
```

The conflicts table referred to in the query is a simple list of the conflicting drugs. The nonces table contains the set of data handles for the patient, discovered via an earlier global search operation.

TABLE I
CONFLICTS

drug
A——
B——

TABLE II
NONCES

nonce
⊙(34)
⊙(56)

The parse tree, shown in figure 2, may be executed starting at the leaves for the local tables and proceeding up the tree.

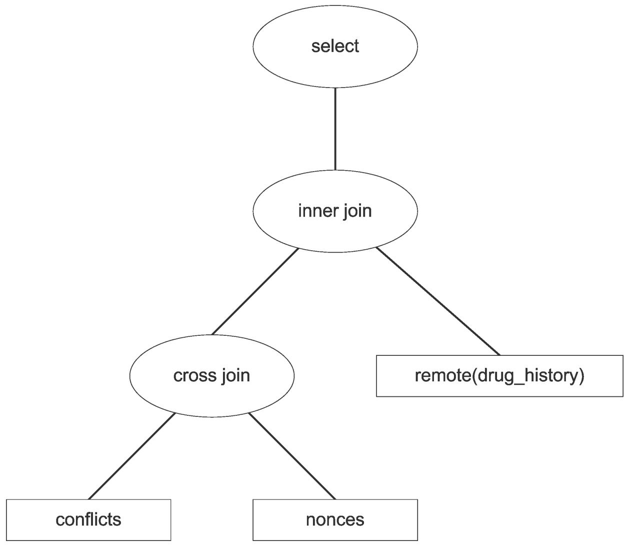

Fig. 2. Original query

When a join depending on remote data is reached, a filter query is sent anonymously to the data owner for each data handle, asking it to send the relevant data to a randomly selected blind comparer.

```
SEND (
  SELECT nonce,drug FROM drug_history
  WHERE drug_history.nonce = O(34)
);
```

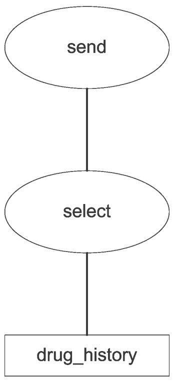

Fig. 3. Filter query, to be executed by the data owner

The parse tree for this is shown in figure 3.

The intermediate table at the query asker (query_table) and the remaining portion of the query to be executed is sent to the blind comparer as well.

```
SELECT EXISTS (
  SELECT * FROM query_table
  INNER JOIN drug_history
  ON query_table.nonce = drug_history.nonce
  WHERE conflicts.drug = drug_history.drug
);
```

This parse tree is shown in figure 4.

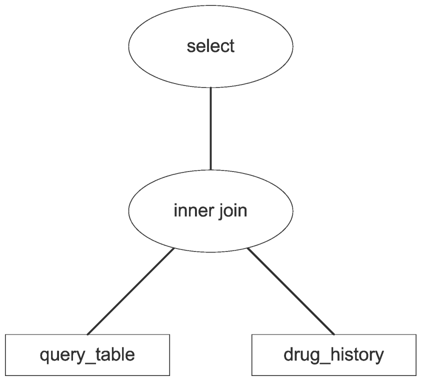

Fig. 4. Remainder of the original query that must be executed at the blind comparer

At this point, the blind comparer knows the answer to the query and returns it to the asker.

IV. Security Architecture

The framework presented thus far supports arbitrary queries. While no information is revealed beyond the result of the query, this begs the question: what if the result itself leaks information? To give the user policy control, it is necessary to label data items with an access control policy. We propose using labels similar to those from programming language information flow tracking [15]. As tables are manipulated by relational algebra operations, the label from each cell taints the label of any result which depends on that cell. Information may safely be revealed if it has been computed based on data that is also safe to reveal.

TABLE III
QUERY_TABLE

drug	nonce
A——	34
A——	56
B——	34
B——	56

Aggregate operations generally require declassifying in order to reveal the answer to a party which is not permitted to see the underlying data. For example, the boolean answer to the drug interaction check would be labeled with the most restrictive set of labels protecting the patient's health history, and must have its label downgraded to be of any use. Our concept of declassification varies subtly from that in operating systems literature [16][17], since we are not asserting that the result is no longer sensitive, merely that the business reasons for needing to reveal it are sufficient to outweigh the small loss of privacy.

A. Access Control

Each cell in a table has a label consisting of an *owner* and set of permitted *readers*.

$$L = \{o : r_1, r_2, \ldots r_n\}$$

A reader is authorized to access the plaintext contents of the cell; an owner may do the same and additionally may change the label on the cell or specify aggregation operations whose results may carry forth labels making lesser security guarantees. In general a label may be a conjunction $L_1 \sqcup L_2$ or disjunction $L_1 \sqcap L_2$ of labels.

The semantics of relational algebra operators such as selects, projects, and joins must be defined with regard to their effect on labels. Projections are trivial: columns may be projected away without changing any labels. Joins, in the every-element-against-every-element sense also do not change the labels. Selections, however, and joins that filter the results based on a selection criteria, must propagate any labels from the fields referenced in the selection condition to all other fields in the same row, tainting that whole row. This happens because the presence or absence of a particular row in the output may reveal certain facts about the fields that were selected upon, even if those fields are later projected away.

B. Declassification

The basic building block for queries is a join operation on two tables with a selection condition for equality of two fields. This can be used both to match an incoming query with the proper patient's record, and for asking a question against that data such as whether the drug history contains a certain entry. An arbitrary select can be represented as a join by the use of a temporary table, included with the query, to indicate one of the values of the selection condition. An important variation on the basic join is to wrap it with an EXISTS clause to return a boolean based on whether or not the join produced any rows at all. This capability was used in the drug interaction example.

The exact privacy implications of a particular join depend on the sizes of the two tables and on the security labels of the fields. If there exists only one drug that might conceivably conflict with the medication currently being dispensed, then the boolean answer reveals whether or not it has been taken. If a larger list of possible conflicts is specified, however, only a probabilistic information leak occurs. The probability can be estimated by considering how easily a data user can locate individuals who match the description revealed by the leaked information, as shown by Hsu [9]. Past queries should also be considered in determining the set of conclusions that can be inferred; Konduri [10] developed techniques to support this.

C. Declassification Tokens

What is necessary then, is to associate a symbolic cost with each operation, and thus with each query, which represents a price that the query asker must pay in order to receive access to the answer. A higher cost is associated with a more egregious privacy leakage. Some number of *declassification tokens* must be given to the asker for it to spend on queries, and the budget of these tokens should be set so that any one organization cannot afford to commit privacy violations in a sustainable manner.

Tokens are issued by a *token bank* entrusted by the data owner to issue tokens on its behalf. Since the token bank does not actually see any private data, it is permissible for it to know who the data owner is and to whom the tokens are being issued. In fact, it may be desirable for the token bank to keep logs of its activity which can be audited or turned over to courts in the event of a legal dispute between data owners and query askers.

Since the sensitive portion of a query's execution is performed at the blind comparer, that host is also responsible for enforcing the privacy constraints by making sure that it collects enough tokens to pay for the operation and devalidates them with the bank. The blind comparer does not have any incentive to cheat here; a token itself is of no use to someone who doesn't know which data owner it is associated with.

V. Conclusion

We have developed a system for running queries against private health records that are stored in a distributed manner at the organizations that produced their respective portions of the record. Queries written as if they were intended for a centralized database get split up into pieces to execute at various providers, and anonymous communication, data privacy, and query privacy is supported. Information flow is tracked and the security level of results may be downgraded after certain aggregation operations, but the use of this essential feature is carefully limited by a token system to prevent abuse.

References

[1] Google Health, "http://www.google.com/health."

[2] Microsoft HealthVault, "http://www.healthvault.com/."

[3] T. C. Rindfleisch, "Privacy, information technology, and health care," *Commun. ACM*, vol. 40, no. 8, pp. 92–100, 1997.

[4] M. Siegenthaler and K. Birman, "Sharing private information across distributed databases," in *submitted for publication*, 2009.

[5] R. Agrawal, A. Evfimievski, and R. Srikant, "Information sharing across private databases," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. ACM Press, 2003, pp. 86–97.

[6] L. Cranor, M. Langheinrich, M. Marchiori, and J. Reagle, "The platform for privacy preferences 1.0 (p3p1.0) specification," W3C Recommendation, Apr. 2002. [Online]. Available: http://www.w3.org/TR/P3P/

[7] Oracle Corporation, "The virtual private database in oracle9ir2: An oracle technical white paper," 2002.

[8] H. Zhu, J. Shi, Y. Wang, and Y. Feng, "Controlling information leakage of fine-grained access model in dbmss," *Web-Age Information Management, International Conference on*, vol. 0, pp. 583–590, 2008.

[9] T. sheng Hsu, C.-J. Liau, D.-W. Wang, and J. K.-P. Chen, "Quantifying privacy leakage through answering database queries," in *ISC '02: Proceedings of the 5th International Conference on Information Security*. London, UK: Springer-Verlag, 2002, pp. 162–176.

[10] S. Konduri, B. Panda, and W.-N. Li, “Monitoring information leakage during query aggregation.” in *ICDCIT*, ser. Lecture Notes in Computer Science, T. Janowski and H. Mohanty, Eds., vol. 4882. Springer, 2007, pp. 89–96.

[11] D. Yixiang, P. Tao, and J. Minghua, “Secure multiple xml documents publishing without information leakage,” in *ICCIT ’07: Proceedings of the 2007 International Conference on Convergence Information Technology*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 2114–2119.

[12] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu, “Two can keep a secret: A distributed architecture for secure database services,” in *Proc. CIDR*, 2005.

[13] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[14] D. L. Chaum, “Untraceable electronic mail, return addresses, and digital pseudonyms,” *Commun. ACM*, vol. 24, no. 2, pp. 84–90, 1981.

[15] A. C. Myers, “Jflow: practical mostly-static information flow control,” in *POPL ’99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1999, pp. 228–241. [Online]. Available: http://dx.doi.org/10.1145/292540.292561

[16] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazires, “Labels and event processes in the asbestos operating system.”

[17] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, “Securing distributed systems with information flow control,” in *NSDI’08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 293–308.