On Security of Sovereign Joins

Einar Mykletun
University of California Irvine
mykletun@ics.uci.edu

Gene Tsudik
University of California Irvine
gts@ics.uci.edu

Abstract

The goal of a sovereign join *operation is to compute a query across independent database relations such that nothing beyond the join results is revealed. Each relation involved in a sovereign join is owned by a distinct entity and the party posing the query is distinct from the relation owners; it is not permitted to access the original relations.*

One notable recent research result proposed a secure technique for executing sovereign joins. It entails data owners sending their relations to an independent database service provider which executes a sovereign join with the aid of a tamper-resistant secure coprocessor. This achieves the goal of preventing information leakage during query execution. However, as we show in this paper, the proposed technique is actually insecure as it fails to prevent an attacker from learning the query results. We also suggest some measures to remedy the security problems.

1 Introduction

At ICDE 2006, Agrawal, et al. [1] proposed a technique for secure execution of so-called *sovereign joins*. The proposed technique uses an outsourced database server equipped with a tamper-resistant secure coprocessor (SC) which performs the actual query execution. A set of specific security goals was stated in [1] and several algorithms were proposed that satisfied these goals. However, as we will show in the paper, even when the stated goals are satisfied, it is still possible for attackers to violate the security requirements of sovereign joins and learn query results.

A sovereign join can be viewed as a join over two (or more) independent (sovereign) relations such that no information – other than the query results – is revealed. Assuming a simple two-relation sovereign join, four parties are involved: the two relation owners, the party posing the query, hereafter called a *client*, and the party executing the query, hereafter called a *server*. The basic tenets of a sovereign join are as follows:[1]

1. The owners of the two relations do not trust each other: no data from one relation should be learned by the owner of the other.
2. The party posing the join is distinct from both owners.
3. The server is not trusted at all, by anone; it should learn nothing about the two relations and query results.
4. Neither relation owner should learn anything about the query results, even if it colludes with the server.

We now describe the model for executing sovereign joins in [1]. It is predicated upon the server being equipped with a tamper-resistant secure coprocessor. (Section 2 gives an overview of secure coprocessors). A server hosts databases belonging to owners who do not have the resource capacity, such as database administrators and software/hardware, to manage their own databases. However, a server is not trusted with owners' data. Therefore, all relations are encrypted prior to being outsourced to a server. The secure coprocessor (SC) is a tamper-resistant programmable computational device hosted at each server. Its memory cannot be accessed and its internal computations cannot be observed. However, due to limited on-board storage capabilities, encrypted relations are stored outside SC, at the server. It is therefore possible for the server to monitor data flowing to and from the *SC*, i.e., the I/O pattern. The main security goal in [1] is to prevent information leaks through this I/O pattern. Figure 1 depicts the key elements of the sovereign join model.

1.1 Sovereign Join Applications

While the main contribution of this paper is the security analysis of [1], we consider sovereign joins to be a very useful primitive/operation. To this end, we present two motivating (though still imagined) example applications for sovereign joins:

1. A commercial flight from Canada to Mexico is scheduled to make a transit stop in the United States. A

[1] The applicability of sovereign joins will become clearer with examples below (see section 1.1).

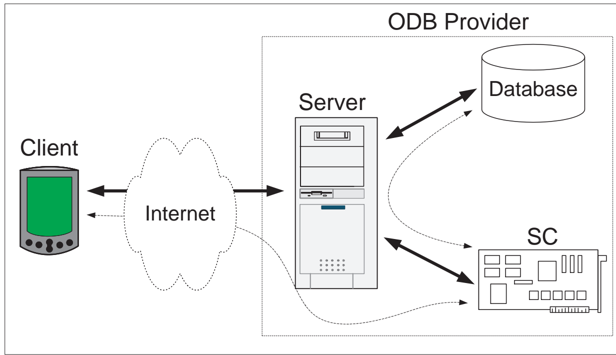

Figure 1. Sovereign Join Model from [1] Thicker arrows indicate physical, and thinner — logical, connections.

U.S. Government agency (e.g., TSA) needs to check whether any passengers are on a certain secret (e.g., FBI-maintained) terrorist watch list. TSA is not permitted to learn about members of this list who are not among the passengers and neither about passengers who are not on the terrorist watch list. A sovereign join is necessary to compute the intersection between the passengers and terrorists, without disclosing either the entire watch list or the full passenger manifest to TSA. In this case, the client is TSA and the two owners are the airline and FBI.

2. The U.S. Department of Justice (DoJ) is required to periodically check whether any undercover secret agency (e.g, CIA) employees have been convicted of any felony in any of the 50 United States. A sovereign join between an individual state's (e.g., California's) records of convicted felons and the list of CIA agents would disclose any such "rogue" agents. At the same time, the list of convicted felons is kept secret by California, the list of agents is kept secret by CIA and DoJ should learn neither, unless the two sets intersect. In this case, the client is DoJ and the two owners are CIA and California.

As can be seen from the above examples, the ability to execute sovereign joins is of great value when the client posing the query (TSA and DoJ, respectively) is not allowed to learn any data which does not satisfy its join predicates. In general, we anticipate that sovereign joins are most valuable in environments, such as healthcare and law enforcement , where privacy is particularly important.

1.2 Organization:

this paper is organized as follows: section 2 introduces the concept and the capabilities of a secure coprocessor. Section 3 summarizes the sovereign join scheme by Agrawal, et al. [1], including the assumed security and threat model. In section 4 we describe our attacks on that proposed scheme. Section 5 describes how to protect against our attacks and section 6 briefly discusses some related work.

2 Secure Coprocessors

A secure coprocessor (SC) is a general-purpose computer trusted to perform its computations undisturbed, even when an adversary has direct physical access to the device [2]. Besides a processor, an SC is typically equipped with non-volatile secure memory, input devices, a backup battery and a cryptographic accelerator. It is fully enclosed in a tamper-resistant container (shielding it from any type of penetration) that cannot be opened without triggering sensors, which incapacitate the device and securely erase all cryptographic material. An SC is usually meant to be installed on a host computer to provide a *secure perimeter* wherein sensitive data may be stored and processed. The physical security of an SC stems from it being equipped with a multitude of sensors that can detect a variety of physical attacks.

The IBM PCI-X product is among the most advanced SCs available on the market today [3]. It is equipped with a 266 Mhz processor and 64 MB on-board memory, as well as 16 MB of read-only flash-based ROM. One area where the PCI-X SC does not perform well is host-to-card communication (host being the server where SC is installed). Although the card throughput is not specified in [3], we can base the performance on its predecessor, the IBM 4758 [2], which had reported throughput of around 600-800 Kbytes/sec [4].

Since the SC is a programmable unit, it is possible for it to execute a query execution engine. Indeed, this has already been done in [5] where an entire query processor is run on a smartcard, a device even more restricted in its resources than an SC such as IBM PCI-X. One of the biggest hurdles associated with implementing a database engine on such an SC is its limited onboard storage. Consequently, either only small relations can be processed (which is unrealistic) or there must be some dependence upon external (to the SC) and thus untrusted storage. For this reason, in the sovereign join scheme of [1] Agrawal, the server stores the (encrypted) relations while the *SC* executes cryptographic and query processing operations.

3 Sovereign Join Approach of [1]

The sovereign join approach in [1] takes advantage of the third-party server which hosts outsourced database and a secure coprocessor at that server. The main goal is to prevent leakage of information through I/O patterns during query processing. In particular, the server should not learn which (encrypted) input tuples are included in query results. Data owners should not learn each other's relation contents, and server should learn nothing about either relation or query results. What we show in this paper is that, even if information leakage through I/O patterns is prevented, the approach suggested in [1] is vulnerable to attacks which result in private data being revealed to unauthorized parties.

3.1 Model

We now recap the query model in [1]. There are players P_A, P_B, P_C, *server*, and SC. P_A and P_B are owners of the sovereign relations over which the join is executed. They are neither authorized to view the query results nor each other's data. P_C is the client, the party that poses the query and receives the query result. P_C does not have access to relations owned by P_A and P_B, and is not authorized to learn anything beyond the joined tuples. *Server* is the database service provider that receives encrypted relations from P_A and P_B. SC is the secure coprocessor hosted at the server. It is responsible for executing the join and is trusted with the relations' contents. It shares encryption keys with relation owners P_A and P_B. We let A and B denote P_A and P_B's relations, respectively, and use C to refer to the query result received by P_C.

P_C transmits its join query to the server. We assume that P_C and SC communicate over an authentic and private channel, i.e., all communication is encrypted and authenticated (which means that they share, or can establish, pairwise secret keys). The server may already have encrypted relations A and B; if not, it asks P_A and P_B to upload their respective relations. It then forwards the query to SC which decrypts and commences with query execution. SC requests encrypted tuples from relations A and B from the server during query processing. SC writes tuples back to the server as part of intermediate computation and eventually outputs the query results C. C is encrypted under a key shared between SC and P_C. Finally, the server sends C to P_C.

Throughout query execution, the untrusted server can monitor exactly which tuples are read by SC. These tuples are encrypted and the server does not learn the contents. However, if tuples are merely selected and written out, the server can easily determine the exact records from A that match (via join predicate) records from B. The technique in [1] involves masking selected tuples by ensuring that SC's tuple access pattern is identical for all sovereign joins over A and B. This is accomplished through a combination of multiple passes over the data and oblivious sorting. (An oblivious sorting algorithm sorts a list of elements such that no observer learns the relationship between the position of any element in the original list and the output list [1]. Bitonic sort is an example of an oblivious sorting algorithm [6].) It is assumed that, even if one of the relation owners (P_A or P_B) colludes with the server, they are still unable to learn anything useful. We consider the same threat model in section 4.

3.2 Clarification

Due to conflicting statements in [1], it is difficult to precisely define which parties the authors attempt to protect against. In the above recap of the model, we labeled P_C as the querier and recipient of query result C, and P_A and P_B as the data owners. However, in [1] it is initially stated:

> "... the recipient of the join result **can** be a party different from one of the data providers ..."

This implies that it is possible for P_A or P_B to be recipients of C. In a later paragraph, the initial statement is contradicted as follows:

> "... the result C is sent to the party P_C, which **is not** P_A or P_B ..."

Upon further thought, it is clear that neither P_A nor P_B can be the recipient of query results C, if the security goals in [1] are to be met. Otherwise, there would be no use in protecting against information leakage from I/O patterns between server and SC, since a data owners who colludes with the server and learns C, they already know exactly which tuples that contribute to the query result. In this paper we proceed with the understanding that a recipient of query result C cannot be one of the data owners.

3.3 Query Result Size

An important observation regarding [1] is that the size of query results is rigid. Basically, the number of tuples returned from a join on relations A and B is $N * |A|$, where N represents the maximum number of tuples from B that match any tuple from A. The size of results sets is therefore always a factor of $|A|$. This makes it easy to determine N when knowing the number of tuples in A (i.e. $|A|$). In a sovereign join, where relations are joined over an attribute unique within each relation (such as the *social security number*), N can only take on values 1 (if one or more tuples join) and 0 (if none of the tuples join). As such, no tuple from one relation can satisfy the join predicate with more than one tuple from another relation.

4 Attack

We now describe a simple attack that targets the sovereign join algorithm in [1]. The attacker is one of the relation owners P_A colluding with the server with the goal of learning query results. This fits within the attack model in [1]. In other words, we show how the attacker can identify the joined tuples. The attack therefore violates the security requirements of sovereign joins.

We imagine the following sovereign join: relation A contains a list of convicted felons in the state of California while B represents a list of undercover secret agents. Both relations contain an attribute SSN (social security number), and the join is computed across this attribute, i.e., an *equi-join*. The agents' identities need to remain secret unless they are convicted felons, in which case they must be identified. Similarly, the list of convicted felons needs to remain secret due to privacy concerns. The U.S. Department of Justice (DoJ), a party with no relation to the owners of A and B, poses the sovereign join query.

As mentioned in section 3.3, the size of query result sets is rigid – a multiple of one of the input relation's size. We exploit this feature as it allows the attacker to monitor the size of the query reply C which is sent from the server to P_C.

The attacker's goal is to learn whether a particular tuple (T_A) joins with a tuple from the other relation. The definition of a sovereign join states that P_A should not have access to relations other than its own. By learning the relationship between a tuple in A and tuples from the other relation, the attacker violates the security of sovereign joins. We assume that the attacker (P_A and the server) knows the query that P_C poses. This is a reasonable assumption since the only attribute in each relation is the SSN. The attacker wants to find out whether a particular tuple T_A in A joins with any tuple in B.

One flavor of the attack occurs when the attacker (P_A) manipulates its own relation by adding and removing tuples at will. For example, P_A can construct a relation A' that consists of the single tuple T_A. Once A' and B have been uploaded to the server, SC proceeds to execute the join and produces the query result C which the server forwards to P_C. The server (under attacker's control) can easily observe the size of C. If $|C| = null$ (i.e. $N = 0$) then the attacker knows that T_A did not match any tuple in B. However, if $|C| = |A'|$, the attacker concludes that T_A matches at least one tuple in in B. In other words, one of the felons in A is indeed is an undercover secret agent. Therefore, the attacker (P_A in collusion with the server) learns information which it is not authorized to know.

A natural and trivial fix is to impose a policy whereby the SC does not execute join queries where one of the input relations is short, e.g., less than t records. However, the same attack would apply, with one minor modification: in addition to T_A, the attacker inserts a number ($> t$) of fake random tuples into relation A'. The query is executed as above and the result *isolates* T_A as the only tuple that determines the query results. The attack therefore still succeeds.

Another flavor of the attack occurs if the attacker knows that P_A and P_B's tuple attributes are unique within their own relations A and B, respectively. We now assume that the attacker (again, P_A in collusion with the server) **cannot** simply manipulate its relation A by truncating it to a single tuple or by padding it with fake tuples (perhaps because a single tuple would look suspicious). In this case, since all tuple attributes are unique, the parameter N can be at most 1, meaning that the query result C is either of size $|A|$ or 0 (if $N = 0$). We again assume that P_A wants to determine whether a given tuple T_A in A matches some tuple in B. P_A adds a duplicate of T_A to a copy of relation A, creating A'. When the sovereign join is run over A and B, it returns a query result set of size $|C| = N * |A|$. If $|C| = 0$, the attacker knows that there is match for T_A. However, if $X \neq 0$, the attacker has no idea whether T_A has a match in B (since many other tuples in A might have matches). That is why the attacker re-runs the same sovereign join query over A' and B. If the result set C' is such that $|C'| > |C|$, the attacker concludes that T_A has a certain match in B.

Knowledge of the client's query One assumption in the above attacks is that the attacker knows P_C's query. This is reasonable when the purpose of the sovereign join query is clear from the context, as in our example of the terrorist watch-list and the airline passenger manifest. Another example is when the attacker can deduce the query based on the attributes in a relation. For example, if a relation contained the single attribute – "social security number", then one can expect an equi-join query based on this attribute.

Replay of the client's query Our second attack makes a further assumption that the attacker can replay the client's query. (Recall that the genuine query is run over A and B and the replayed version is run over A' and B.) This assumption might be considered too strong (i.e., unrealistic) but nothing in the sovereign join model of [1] explicitly precludes it. In fact, authenticating the origin of the query by the SC is not sufficient. Suppose that we fix the problem by asking the client P_C to always sign its query requests. However, even signed messages can be easily replayed. To protect against replays, each client query request must be *timely* and *fresh*. These are standard notions in cryptographic protocols and, instead of treating them here, we refer to [7]. Suffice it to say that timeliness and freshness necessitate the use of sequence numbers and timestamps. The former complicate matters by requiring the SC to maintain *per client* state (i.e., a current sequence number) while the

latter require the SC to maintain synchronized clocks with all clients. These are clearly non-negligible costs. An alternative to using timestamps and/or sequence numbers is to impose a real-time challenge-based authentication protocol between the client and the SC. This would also impose certain costs upon the SC since to authenticate the client (and its query) the SC would need to challenge the client which would translate into (at least) a 3-message protocol and the need to keep state in the interim.

5 Suggested Fix

Our attacks rely on observing the size of the query results returned by algorithms in [1]. These results sets reveal the parameter N: the maximum number of tuples from relation B that match any tuple in A. One possible fix that can make the algorithms resistant to our attacks is the addition of random noise to the query results, e.g., by padding with superfluous (encrypted) random tuples. This would make the size of the result non-deterministic; the same query run multiple times over identical relations would yield different result sets.

A more secure fix would involve always returning results of the same size, i.e., $N * |A|$. This would completely address our attacks. However, such a solution is clearly impractical due to the significant additional communication and storage overhead. A better approach would allow the client to specify the desired security level through a *privacy parameter* which would indicate the amount of noise (padding) inserted into the query results by the SC. This way, the client could achieve a custom trade-off between efficiency and privacy. It is also quite likely that more effective fixes can be obtained by re-engineering the algorithms in [1] to take into account the attacks presented in this paper.

6 Related Work

To the best of our knowledge, the work of Agrawal, et al. [1] is first to propose a solution for sovereign joins based upon an outsourced database model and a 3rd party server equipped with an SC. The closest related work, although more general in scope, is the use of logic circuits for secure function evaluation [8, 9]. The goal of a two-party secure function evaluation protocol is to enable parties X and Y, with respective inputs x and y, to jointly compute some function $F(x, y)$, such that they learn only the function's output, but nothing about each other's inputs[2]. Although it has been shown that logic circuits can be used to securely compute complex functions, such work remains mostly of theoretical interest. Albeit, a recent result by Malkhi, et al. [10] has alluded to the existence of more practical solutions.

In [4], Smith and Safford explore the practicality of achieving private information retrieval (PIR) through the use of secure coprocessors hosted at a database server. The goal is to completely hide – from the database server and outsiders – any identifying information about tuples selected during query execution. The performance measurements are based on the IBM 4758 coprocessor [2], which was the state-of-the-art at the time of publication. The conclusion of [4] is that the IBM 4758 is technologically insufficient when attempting to realize a practical and secure database query model. The main claimed bottleneck is the relatively high latency between the coprocessor's cryptographic engine and its internal RAM. The developers of the follow-up to IBM 4758, the next generation IBM PCIX coprocessor, acknowledge that this bottleneck still remains [3].

A more general outsourced database model was introduced by Haĉigümus, et al. in [11]. That work focused on enabling an untrusted server to execute queries over encrypted data, using techniques such as data bucketization and homomorphic encryption functions [12]. It provided a solid foundation for further research in this area, but does have a few major disadvantages. These include the computation and bandwidth overheads incurred by querying clients, as well as the limitations in types of queries that can be executed by the server. A more recent paper [13] suggested overcoming some of these shortcomings by extending the model with a secure coprocessor. A high-level architecture for this new model was suggested along with sketches for the execution of basic database operations; however, no performance analysis or experimental results have been provided.

7 Conclusion

In this paper we revisited the work by Agrawal, et al. [1] which focused on secure execution of sovereign joins achieved through the use of a secure coprocessor hosted at a 3rd party database server. We demonstrated that the proposed technique is subject to an attack that allows a data owner to learn query results and, thus, the other owners' data. This represents a violation of the sovereign join goals. Two slightly different attack scenarios were described, both based upon observing the size of the query result.

References

[1] R. Agrawal, D. Asonov, M. Kantarcioglu, and Y. Li, "Sovereign Joins," in *International Conference on Data Engineering*, 2006.

[2] Although we here give a definition of secure function evaluation in terms of two-party protocols, they can just as well be n-party protocols, where function $F()$ takes n inputs.

[2] J. G. Dyer, M. Lindemann, R. S. R. Perez, L. van Doorn, and S. W. Smith, "Building the IBM 4758 Secure Coprocessor," in *EEE Computer*, pp. 57–66, 2001.

[3] T. W. Arnold and L. P. V. Doorn, "The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer," `http://www.research.ibm.com/journal/rd/483/arnold.pdf`.

[4] S. W. Smith and D. Safford, "Practical server privacy with secure coprocessors," in *IBM Systems Journal*, pp. 683–695, 2001.

[5] L. Bouganim and P. Pucheral, "Chip-Secured Data Access: Confidential Data on Untrusted Servers," in *International Conference on Very Large Data Bases*, pp. 131–142, 2002.

[6] K. E. Batcher, "Sorting Networks and their Applications," in *AFIPS Spring Join Computing Conference*, vol. 32, 1968.

[7] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC Press series on discrete mathematics and its applications, CRC Press, 1997. ISBN 0-8493-8523-7.

[8] O. Goldreich, "Foundations of Cryptography," in *Cambridge University Press*, vol. 2: Basic Applications, 2004.

[9] A. C. Yao, "How to Generate and Exchange Secrets," in *Symposium on Foundations of Computer Science*, pp. 162–167, 1986.

[10] B. P. D. Malkhi, N. Nisan and Y. Sella, "Fairplay - a secure two-party computation system," in *USENIX Security*, 2004.

[11] H. Hacigümüş, B. Iyer, and S. Mehrotra, "Providing Database as a Service," in *International Conference on Data Engineering*, March 2002.

[12] H. Hacigümüş, B. Iyer, and S. Mehrotra, "Efficient execution of aggregation queries over encrypted databases," in *International Conference on Database Systems for Advanced Applications (DASFAA)*, 2004.

[13] E. Mykletun and G. Tsudik, "Incorporating a Secure Coprocessor in the Database-as-a-Service Model," in *nternational Workshop on Innovative Architecture for Future Generation High Performance*, 2005.