Space-optimized Implementation of a Database Obfuscator for Group Privacy

Cheng-Fang Yang

May 5, 2006

1 Introduction

Privacy of data concerns data owners, customers, and marketers. Breaches in privacy populate headlines frequently, including one case where ID numbers and contact information of Hong Kong citizens who complained about the police in the past five years were exposed on the Internet [11]. The research community has proposed data privacy protection through statistical data randomization [1, 5], query auditing [10], secure multi-party computation [9], and private information retrieval [4]. We would like to protect the data differently by making imprecise queries to a database computationally infeasible to evaluate. In other words, we restrict allowed accesses to the database to those that should be computed quickly. We can do this through obfuscation, which effectively moves access control into the target database, allowing database owners to release the database knowing that only precise queries can be evaluated efficiently, depending exponentially on the number of records that would satisfy the query. Thus, access becomes non-interactive–the owner or any trusted party does not have to oversee or approve every query submitted by the database user.

Legal accesses are embodied by the owner-specified access/privacy policy, which describes the *ideal functionality* of the obfuscated database by specifying which queries can be satisfied efficiently. *Ideal functionality* is an abstraction that allows us to decide whether the obfuscator is secure [7]. We can imagine a trusted third party who provides the ideal functionality by filtering all accesses from a simulator. We will consider our construction secure if an attacker's actions can be emulated by this simulator and trusted third party. Because our construction leaks meta information about the content of the database, we work with a weaker notion of obfuscation than a "virtual black box" [2]. The construction is still useful because the leakage does not compromise the security of access control, which is our primary focus.

1.1 Purpose of obfuscation

In program obfuscation, we garble the source code of the program so that users who have access to the result will only be able to garner as much information

as if they had only the set of inputs and outputs that the unobfuscated program supports. In addition, the program should not be much more inefficient than the original program and the two should be approximately the same in functionality [2].

Similarly, we would like to obfuscate a database so that it remains functional, is efficient when the original is efficient, and its representation is inscrutable to an attacker's investigation. In other words, users can only learn information that their queries give them access to according to the privacy policy. Only allowed queries will reveal new information after it is evaluated. Obfuscation allows the owner of a database to distribute freely his database, knowing that the user will be able to interact with the database only according to the owner's access policy, which we integrate in the database along with its enforcement. By integrating the access control in the database, we make circumvention computationally infeasible.

1.2 Group privacy

Narayanan and Shmatikov proposed an obfuscation algorithm that ensures a new notion of privacy, group privacy [15]. The resulting obfuscated database returns the results of a query quickly as long as the query is precise. Queries will be evaluated in exponential time–exponential in the number of records that satisfy the query. A precise query is satisfied by a small set of records and would be evaluated quickly. An imprecise query would be computationally infeasible.

An obfuscator that can ensure group privacy in a database could help protect large public datasets. It could help prevent a nurse from finding out all the details of all the patients in a hospital database, an employee from gathering contact information from a customer relationship management database and selling it, and spammers from indiscriminately harvesting all the telephone numbers in a directory.

For example, a financial advisor may outsource some of his duties safely by first obfuscating his client database. A recipient of the database would only be able to efficiently access those records that are allowed by the financial advisor's privacy policy, since the policy enforcement comes with the database. Consequently, the recipient cannot harvest all the financial advisor's clients' information and spam the clients with his own financial offerings or sell their information to other spammers.

1.3 Space usage

An existing implementation of the algorithm posited by Narayanan and Shmatikov, written by Chris Nienhuis, demonstrates the possibility of obfuscating a database; however, the space requirement of $O(N^2)$ restricts the size of databases that can be obfuscated and still be usable. We reduce space usage by employing reverse Merkle trees, which are described later. The blowup in size using the reverse Merkle tree is $Nlog(N)$ at best, when all the records in the database are unique,

and up to N^2 otherwise. We evaluate the improvement on a database of function call frequencies.

1.4 Organization of paper

We discuss related work and provide a short comparison of our notion of obfuscated databases and previous work in the next section. Then we establish the foundation upon which we build the obfuscator in section 3. We also discuss how we evaluate the implementation in section 3. In section 4, we describe the original obfuscating algorithm and which tools we need for the improved version, including the reverse Merkle tree. The modified obfuscation algorithm is described and analyzed in section 5, and the implementation is detailed under section 6. Finally, we evaluate the implementation in section 7 compared to an implementation that is not optimized and draw conclusions in section 8.

2 Related Work

Barak et al. define the objectives of obfuscation and demonstrate the impossibility of creating a general obfuscator, an obfuscator that can obfuscate all possible programs. Goldwasser and Kalai also show that obfuscation with regard to auxiliary input is impossible [8]. Their result does not apply to our obfuscator because they base it on a different class of programs. While a general obfuscator may elude us according to Barak et al., we can still construct obfuscators for particular purposes. We show that our database obfuscator is one such program. We use point functions, in particular hash functions, which return the value True on only one input value. Goldwasser and Kalai show that point functions are obfuscatable even with auxiliary input [8], and Lynn et al [12] and Wee [17] construct obfuscators for point functions. Their work assures us that we can use obfuscated point functions.

We would like to use obfuscation to ensure database privacy. Both academia and industry have researched database privacy. However, initiatives like Microsoft Research's [14] and Agrawal et al's [1] address the accumulation of data, statistical knowledge that can be gleaned from a database and risks to the privacy of individual records. We are concerned only with access to the data; therefore, our studies are orthogonal to one another. We allow accesses to individual records as long as users can specify them precisely while the other studies seek to protect individual records amongst statistical analyses or data mining operations. To control access, we integrate the enforcement of an access policy into the database and so, create a non-interactive privacy enforcer. Chawla et al. [3] also have a non-interactive scheme, but like research in private information retrieval, they focus on the privacy of data rather than access to the data.

Our implementation is based on the algorithm and heuristic proposed by Narayanan and Shmatikov [15]. A previous implementation in Java was written by Chris Nienhuis without the proposed heuristic.

3 Security Context

3.1 Foundation

Our obfuscator satisfies a weaker notion of security than the "virtual black box" property as defined by Barak et al. In the "virtual black box" model, the only queries that can be efficiently answered by an adversary with the obfuscated database are queries that can be efficiently answered by a simulator that communicates only with a trusted third party that enforces the ideal functionality of the database, the "virtual black box." The ideal functionality is embodied in an owner-specified access policy. Our notion of a good obfuscator allows the attacker some information about the database; specifically, how much repetition is there. On the other hand, the attacker cannot see which values are repeated unless they know the attribute values and can determine if the supplied attribute values are repeated often. We accept this leakage since it does not allow the attacker to violate the access policy. Our construction simulates a trusted third party and thus, integrates access control into the database upon obfuscation. This foregoes the process of establishing the third party, which requires its own protection and is difficult to establish.

3.2 Evaluation database

To test the effectiveness and efficiency of the obfuscator implementation, we will be testing it with a database of system call traces that is used by the Navel system. The Navel system allows a group of users to share information about system failures, including the system call traces that lead up to a failure and possible solutions to prevent the same failure in the future. The information used by Navel can be sensitive; for example, the number of calls to particular functions could provide clues about the input before the failure. Therefore, we would like to obfuscate the database to prevent them from browsing sensitive system information, but still allow legitimate users to submit their data and retrieve precisely requested data.

4 Building blocks

4.1 Original Algorithm

We modify the obfuscation algorithm proposed by Narayanan and Shmatikov in [15]. In summary, the original obfuscation replaces a given database with records of hashes and encryptions. Every hash function call is salted with a random number, to prevent attackers from comparing values to find out which values are equal. Since the hash functions are one-way, the resulting representation gives attackers no information about the individual values of each entry.

To obfuscate a database, the data owner first partitions the database into *query attributes* and *data attributes*. Then the obfuscator obfuscates the database one record at a time. For each record in the database, it *randomly* generates a

secret key that is the same length as the number of records in the database. We compute and store the hash of the secret key so that the user can see if they have the correct key to retrieve the data attribute value for this record. The user needs the correct key because we xor the data attribute value with another hash of the secret key. (Every hash is computed with a randomly salted hash function call.) From the secret key, we derive *shares*, which are copies of the secret key with bits (possibly different) missing. A share is associated with each query attributes, and the share's n^{th} bit is missing when the n^{th} record's query attribute value is the same as the current record. We compute and store the hash of each query attribute value so that the adversary cannot simply look at the database representation to find the values. For the same reason, we xor the share for each query attribute value with another hash of the query attribute value. Once this procedure has been done for each record in the database, the database has been obfuscated.

In retrieval, the user supplies query attribute values to collect as many shares as possible to reconstruct the whole key. Because each share is missing the same number of bits as the number of records that have the same query attribute values, more common query attribute values are less useful in secret key recovery. Once the user recovers as many bits as possible, there may still be missing bits, so the user must guess the remainder to form the secret key. Because the number of guesses increases with the frequency of given query attribute values and only precise queries can be satisfied quickly, the obfuscated database ensures group privacy.

4.2 Tools

To improve the construction's space usage, instead of randomly generating the key, we use a reverse Merkle tree to *pseudo-randomly* generate a secret key. A Merkle tree is a binary tree of hashes where the leaves are hashes of data and the parent nodes are hashes of the children [13]. So, the parent nodes' values are determined by the children nodes. In a reverse Merkle tree, the children nodes are determined by the parent nodes. In our implementation, we feed the value of the parent node to a length-doubling pseudo-random number generator to create values for the children nodes. If a parent node has 2 bits, then the pseudo-random number generator generates a number with 4 bits, 2 for each child. We form the key by concatenating the values of the leaves. Because of the dependence of children on parents, we can store just the root node and reconstruct the tree using the same algorithm. We depend on the determinism of the pseudo-random number generator. While we would like keys to be unpredictable, we need to be able to reconstruct the key with the reverse Merkle tree.

The hash function that we use to hash the key and the query attributes must be one-way, so that an attacker cannot determine the value of the query from the hash of the query. It just needs to be weakly collision resistant because we salt the hash each time we use it. Because of these constraints, we can use SHA-1. Wang, Yin, and Yu can find a collision in 2^{69} operations instead of

Name	Flight	Detail	Secret Key
Smith	88	Acme	0100
Brown	500	Counter	1100
Jones	88	Nonrevenue	1110
Jones	100	Agent	0110

Table 1: Ticket sales database

2^{80} [16], but this is still within our constraints of weak collision resistance. Our adversaries cannot use attacks on SHA-1's weak collision resistance to subvert our obfuscator since we salt, with a random number, each SHA-1 function call, simulating a different function each time. We put these tools together in a detailed description of the algorithm in the next section.

5 Modified obfuscation

In this section, we will describe the modified algorithm in detail and illustrate the obfuscation with a small example, slightly modified from [15]. Let Name and Flight be query attributes and Detail be a data attribute. The example database (below) has only 4 elements so the attacker could launch a successful brute-force attack of just 2^4 guesses for the secret key, but our construction invalidates the brute-force strategy with large datasets.

5.1 Algorithm

5.1.1 Obfuscation

Modified from [15], the obfuscation algorithm follows:

For each record i, $< x_{i1}, x_{i2}, x_{i3}, x_{i4}, ...y >$ where the x's are query attributes and y is the data attribute, we create new record, $< salts, u_i, z_i, v_{i1}, w_{i1}, v_{i2}, w_{i2}, v_{i3}, w_{i3}, ... >$ where

- salts = The salts that are used in each of the following hash function calls for this record.
- $u_i = SHA(salt_1, r_i)$ where r_i is the complete secret key for this record.
- $z_i = SHA(salt_2, r_i) \oplus y$
- $v_{ij} = SHA(salt_3, x_{ij})$
- $w_{ij} = SHA(salt_4, x_{ij}) \oplus s_{ij}$ where s_{ij} is the share for this query attribute.

The share for each query attribute increases the size of the database quadratically, since its length in bits is the number of records in the database. We reduce this overhead to $Nlog(N)$ by using a reverse Merkle tree to generate the secret key. First, the data owner specifies how many bits (call it k) are at each node.

With k and the number of rows N, we can construct the tree, keeping in mind which bits are missing. So, for $k = 3$ and $N = 24$, a share with missing bits in the 2nd, 4th, and 5th bit positions can be represented as below.

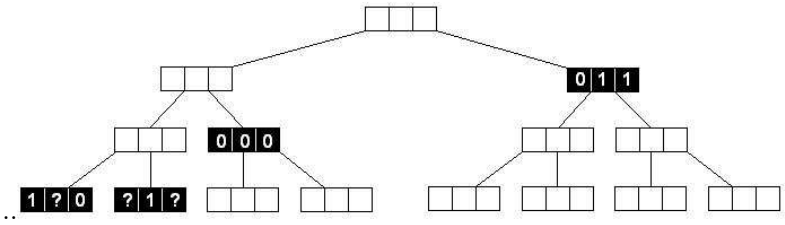

Figure 1: A reverse Merkle tree with security parameter k = 3

Because of the nature of reverse Merkle trees (the children depend on the parents), we can reconstruct the key with knowledge of just the highlighted nodes. We reduce the storage required for this key, from 24 bits to 12 bits, not including the position information for each stored node, which is constant and does not rely on the size of the database.

For our example database, the first record $< Smith, 88, Acme >$ would be replaced by the following if its secret key were 0100:

$< salts[]$,

$SHA(salt_1, 0100), SHA(salt_2, 0100) \oplus Acme$,

$SHA(salt_3, "Smith"), SHA(salt_5, "Smith") \oplus ?100 >$.

The ?'s indicates which bits in the salt are missing. Because the first record has Name=Smith, only the first bit is missing in the share. The reverse Merkle tree stored for this share value ?100:

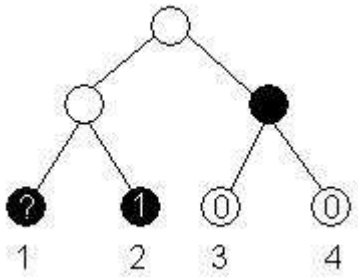

Figure 2: A reverse Merkle tree with security parameter k = 1 and 1 bit missing

The first and second bits are stored along with the parent node of the third and fourth. This is enough to reconstruct the share since each child depends on their parent node.

5.1.2 Retrieval

In retrieval, the user inputs as many query attribute values as he can to describe the record he would like to access. With each of the supplied query attribute values, the database access program compares the hash of the supplied query attribute value to each corresponding entry in each record of the obfuscated database to find matches. Matches will allow it to retrieve that query's stored share, which reveals a part of the record's key. Upon gathering all the shares possible with the users' input, the program consolidates the shares to find out how much computation is required to find the key. If there are still n bits missing after the shares are collected, then we must guess 2^n values. If the user were not precise enough and ended up with 10 missing bits, they would have to guess up to 2^{10} values. We guess until the hash of our guess matches the hash that is stored in the obfuscated record. Once the key is found, we can compute the exclusive or with another hash of the query and the stored attribute z to find the data attribute.

Suppose a user queries for records that have query attribute value, $Name = Jones$, from our example database. The obfuscated database is processed one record at a time. From the third record, the user acquires the share 11?? because the third and fourth records of the database have $Name = Jones$. He guesses all possible secret keys from this information 1100, 1101, 1110, 1111, and eventually finds 1110, the correct key. He then uses this to decrypt the data attribute Nonrevenue. He repeats this procedure for the fourth record.

5.2 Tradeoffs

While we can reduce the storage requirement for the obfuscated database by using Merkle trees, we also reduce time efficiency. We must reconstruct the Merkle tree for every query attribute, recursively find the missing bits, and guess the value of the secret key. Furthermore, the space reduction depends heavily on the dataset: a dataset with many repeated query attribute values will require more time to compute the key and force users to specify precise queries to reduce this time. On the other hand, the number of missing bits will also increase, undercutting the benefits of using a Merkle tree. Storing the Merkle tree reveals the size of the incomplete keys, which indicates the number of other records that have the same query value. We could mitigate this information leakage, but then our space efficiency would be compromised. Finally, the obfuscation can be effectively applied only to large datasets. Since a brute-force attack would guess all possibilities for they key and the key as long as the number of records in the database N, it would take 2^N trials. Large dataset would have N large enough for 2^N trials to be computationally infeasible. As seen in the four-record example, smaller datasets are subject to brute force attacks.

6 Implementation

We implement the new obfuscator in Python, so it can run on Windows, Linux, Mac OS X. Python is freely available online [6]. The implementation includes an obfuscator, the reverse Merkle data structure, utilities for parsing, and a retrieval program. Besides the standard Python library, we also use BitVector, created by Avinash Kak.

We follow strictly the construction in section 5; however, we represent the information differently to save on storage space. For example, each node does not explicitly need a string describing its position in the reverse Merkle tree. Rather, when the nodes that must be kept are recorded, we record them in the order of reconstruction. This way, in recovering the share from the tree, we only need to know the depth of the node stored to know when to stop expanding and concatenate our result for this subtree to the secret that we are reconstructing. Consider this subtree for instance:

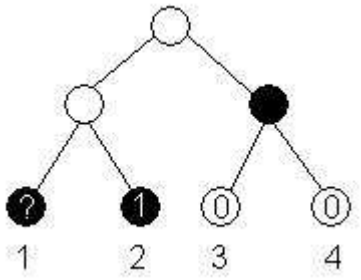

Figure 3: A reverse Merkle tree with security parameter k = 1 and 1 bit missing

We record the shaded nodes in order, 1, 2, and the last shaded node. In reconstructing the share, we have ? concatenated to 1, concatenated to the expansion of the last shaded node. We expand the node as if we were making a new Merkle tree with its depth.

Also, we arbitrarily set the missing bits to 0 so that we can use BitVector and use just one bit instead of allocating a whole character's worth of memory to use '?'. This means that we must incur additional execution penalties to traverse the obfuscated database to find out which 0 in the share is in fact a 0 or a missing bit marker. Finally, instead of storing a vector of salts for each record, we store only a seed for a pseudo-random number generator. Then, in retrieval, we must take more time to call the pseudo-random number generator to obtain the salts for the hash function calls.

7 Evaluation

We test the implementation on a database of function calls and their frequencies from the Navel system. Each combination of function calls and frequencies is associated with a *class*, which is the diagnosis of the error. We would like to protect this information since it could reveal the inputs prior to the error, so *class* is the data attribute. The frequencies of each function call are query attributes. There are 531 different functions so there are 531 query attributes.

We conduct the experiments on a PC with a Pentium 4 3.4 GHz processor. The following table gives a comparison of the running time and resulting obfuscated database size. Our implementation runs much slower than the original implementation, which did not have to process the trees. For this dataset we also came out with dramatically larger obfuscation–about 5 times the original obfuscation.

Output	Records	Java Time	Python time	Java space	Python space
215K	128	7.968s	8m 30.912s	7.4M	45M
420K	256	16.445s	23m 41.665s	18M	108M
830K	512	48.627s	67m 53.842s	46M	262M

Table 2: Time and space usage comparison

The experimental results contradict what we expect from using Merkle trees. Possibilities for this discrepancy include particular implementation details and serialization techniques in Python and Java. In addition, with 531 query attributes, we have to construct as many reverse Merkle trees, one for each share in each record. Finally with similar query attribute values across records, we cannot take advantage of the space savings. It seems that the overhead of constructing and maintaining the trees trumped any savings that would theoretically have been possible.

8 Conclusion

We implemented a database obfuscator that ensures group privacy [15]. A usable obfuscator could benefit data owners who would like to embed access control into their database. We sought to improve on the space usage of a previous implementation so that we could use it for larger datasets. Instead, we found that the loss in time-efficiency was a prohibitively expensive tradeoff. Moreover, maintaining the trees, whether because of serialization details or its implementation, presented an insurmountable overhead in obfuscating the Navel database. We have demonstrated the possibility of using reverse Merkle trees in the obfuscator, but we must improve the preliminary implementation further before we can use it practically. A large dataset without as much repetition of query attributes' values would theoretically leverage the reverse Merkle tree's

capabilities to store just $NlogN$ in the best case and N^2 at worst. Further investigation is needed to realize this theoretical result.

References

[1] Agrawal, R., and Srikant, R. Privacy-preserving data mining. In *Proc. of the ACM SIGMOD Conference on Management of Data* (May 2000), ACM Press, pp. 439–450.

[2] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. On the (im)possibility of obfuscating programs. *Lecture Notes in Computer Science 2139* (2001), 1–18.

[3] Chawla, S., Dwork, C., McSherry, F., Smith, A., and Wee, H. Toward privacy in public databases. In *TCC* (2005), pp. 363–385.

[4] Chor, B., and Gilboa, N. Computationally private information retrieval (extended abstract). In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (New York, NY, USA, 1997), ACM Press, pp. 304–313.

[5] Evfimievski, A. Randomization in privacy preserving data mining. *SIGKDD Explor. Newsl. 4*, 2 (2002), 43–48.

[6] Foundation, P. S. The python programming language, 2006. http://www.python.org/ (accessed 10 April, 2006).

[7] Goldreich, O. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, New York, NY, USA, 2000.

[8] Goldwasser, S., and Kalai, Y. T. On the impossibility of obfuscation with auxiliary input. In *FOCS* (2005), pp. 553–562.

[9] Halevi, S., Krauthgamer, R., Kushilevitz, E., and Nissim, K. Private approximation of np-hard functions. In *STOC* (2001), pp. 550–559.

[10] Kleinberg, Papadimitriou, and Raghavan. On the value of private information. *TARK: Theoretical Aspects of Reasoning about Knowledge 8* (2001).

[11] Leyden, J. HK police complaints data leak puts city on edge, Mar. 2006. http://www.theregister.co.uk/2006/03/28/hk_data_leak_rumpus/print.html (accessed 10 April, 2006).

[12] Lynn, B., Prabhakaran, M., and Sahai, A. Positive results and techniques for obfuscation. In *Benjamin Lynn, Manoj Prabhakaran, Amit Sahai, Positive Results and Techniques for Obfuscation , Proceedings of Eurocrypt 2004.* (2004).

[13] Merkle, R. Secrecy, authentication, and public key systems, 1979.

[14] Microsoft. Database privacy, April 2006. http://research.microsoft.com/research/sv/DatabasePrivacy/ (accessed 10 April, 2006.

[15] Narayanan, A., and Shmatikov, V. Obfuscated databases and group privacy. In *ACM Conference on Computer and Communications Security* (2005), pp. 102–111.

[16] Wang, X., Yin, Y. L., and Yu, H. Finding collisions in the full sha-1. In *CRYPTO* (2005), pp. 17–36.

[17] Wee, H. On obfuscating point functions. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing* (New York, NY, USA, 2005), ACM Press, pp. 523–532.