International Journal of Information Security manuscript No.
(will be inserted by the editor)

Fernando Esponda · Stephanie Forrest · Paul Helman

Negative Representations of Information

Abstract In a negative representation a set of elements (the positive representation) is depicted by its complement set. That is, the elements in the positive representation are not explicitly stored, and those in the negative representation are. The concept, feasibility, and properties of negative representations are explored in the paper; in particular, its potential to address privacy concerns. It is shown that a positive representation consisting of n l-bit strings can be represented negatively using only $O(ln)$ strings, through the use of an additional symbol. It is also shown that membership queries for the positive representation can be processed against the negative representation in time no worse than linear in its size, while reconstructing the original positive set from its negative representation is an $\mathcal{NP}$-hard problem.

Fernando Esponda‡†
‡Previous Address
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131-1386
†Current Address
Department of Computer Science
Instituto Tecnológico Autónomo de México
México, D.F.
E-mail: fesponda@cs.unm.edu

Stephanie Forrest
Santa Fe Institute
1399 Hyde Park Road
Santa Fe, NM 87501
E-mail: forrest@cs.unm.edu

Paul Helman
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131-1386
E-mail: helman@cs.unm.edu

The paper introduces algorithms for constructing negative representations as well as operations for updating and maintaining them.

Keywords Negative Databases · Immune-inspired Algorithms · Privacy · Information Hiding · Data Representations

1 Introduction

Large collections of data are ubiquitous, and the demands we place on them continue to increase. We expect data to be available on demand but to be protected from malicious parties; we would like the ability to search data collections in new ways, drawing inferences about large-scale patterns and trends, while preventing the wrong kinds of inferences (as in baseless racial profiling). Content and the rules for accessing it must be continually updated and, eventually, we will want the ability to audit the uses to which our personal data are put. Although many of these problems are old, they must now be solved more quickly for larger and more dynamic collections of data under more stringent privacy requirements.

In this paper we introduce an approach to representing data that addresses some of these issues, particularly those related to privacy and distributed data. Our goal is to devise data representations that prevent inappropriate queries and inferences, while supporting legitimate operations.

There are several motivating scenarios for our work. Consider, for example a watch list that is to be made available to airline agents. It is desirable for these agents to have the ability to verify whether a given name is on the list, but at the same time not to have the ability to arbitrarily browse its contents (or even assess its size), lest it fall into the wrong hands. A second goal involves distributed data, where we would like to privately determine the intersection of sets owned by different parties. For example, two or more entities might wish to determine which of a set of possible items (e.g. transactions) they have in common without revealing the totality of the contents of their database or its cardinality. A longer term motivation concerns a large database of personal records, which an outside entity might need to search, for example, to identify suspicious activities or to conduct epidemiological studies. Under this scenario, it is desirable that the database support only the legitimate queries while protecting the privacy of individual records, say from inspection by an insider.

We present negative databases as a specific example of representing data negatively. In this approach, the negative image of a set of data records is represented rather than the records themselves (Figure 2). Initially, we assume a universe U of finite-length records (or strings), all of the same length l and defined over a binary alphabet. We logically divide the space of possible strings into two disjoint sets: DB representing the set of records that holds the information of interest, and $U - DB$ denoting the set of all strings not in DB. We assume that DB is uncompressed (each record is represented explicitly), but we allow $U - DB$ to be stored in a compressed form called NDB. We refer to DB as the *positive* database and NDB as the *negative* database.

Consider, for example, a banking database of the tuples <Name, Transaction type, Date> with a total length of 400 bits, where the first field stores the name of a client, the second has the description of the banking transaction like "withdrawal" or "deposit" and the third stores the date the transaction took place. The positive database will contain the actual names of the bank's clients, their transactions and the date they occurred, while the negative database will instead have all possible 400 bit combinations except the ones corresponding to the bank's factual data. From a logical point of view, either database will suffice to answer questions regarding DB. However, the different representations present different advantages. For instance, in a positive database, inspection of a single record provides meaningful information. However, inspection of a single (negative) record reveals little about the contents of the original database. Because the positive tuples are never stored explicitly, a negative representation would be much more difficult to misuse. Similarly, depending on the specific representation of NDB, the efficiency of certain kinds of queries may be significantly different than the efficiency of the same query under DB. Some applications may benefit from this change of perspective. Most applications seek to retrieve information about DB as efficiently and accurately as possible, and they typically are not explicitly concerned with $U - DB$. Yet, in situations where privacy is a concern it may be useful to adopt a scheme in which certain queries are efficient and others are provably inefficient.

This paper describes the concept of a negative representation, gives some initial results on its feasibility, and illustrates how alternative negative representations can produce distinct properties with respect to retrieving information or protecting privacy. We do not yet fully understand all of the properties of the negative data representations we present, and there may be other representations with different properties appealing to distinct applications.

In the following sections, we first show that implementing NDB is computationally feasible. We do this by introducing a scheme that requires $O(ln)$ negative records to represent the complement of a positive database consisting of n l-bit strings, and then giving an algorithm for finding such a representation efficiently. We next investigate some of the privacy implications of the negative scheme. In particular, we show that the general problem of recovering a positive database from our negative representation is $\mathcal{NP}$-hard. We then present a randomized algorithm for creating negative representations that are difficult to reverse, as well as operations for updating and maintaining a negative database. We discuss what types of queries can be carried out efficiently under this representation and how negative databases can be used to perform set intersection—an important operation among databases. Finally, we review related work, discuss the potential consequences of our results, and outline areas of future investigation.

2 Representation

In order to create a database NDB that is reasonable in size, it is necessary to compress the information contained in $U - DB$. We introduce one additional symbol to the binary alphabet, known as a "don't care," written as $*$. The entries in NDB will thus be l-length strings over the alphabet $\{0, 1, *\}$. The don't-care symbol has the usual interpretation and will match either a one or a zero at the bit position where the $*$ appears. Positions in a string that are set either to one or zero are referred to as "defined" or "specified" positions, and locations where a $*$ appears are referred to as "unspecified" positions. With this new symbol we can potentially represent large subsets of $U - DB$ with just a few entries.

For example, the set of strings $U - DB$ can be exactly represented by the NDB set shown below:

DB	$(U - DB)$		NDB
	001		
000	010		0*1
111	011	$\Rightarrow$	*10
	100		10*
	101		
	110		

The convention is that a binary string s is taken to be in DB if and only if s fails to match each of the entries in NDB. This condition is fulfilled only if for every string $t_j \in NDB$, s disagrees with t_j in at least one defined position.

2.1 The Prefix Algorithm

In this section we present an algorithm as proof that a negative database NDB can be constructed in reasonable time and of reasonable size. The prefix algorithm introduced here is deterministic and reversible—DB can be recoved by inspecting NDB—which has consequences for the kinds of inferences that can be made efficiently from NDB. The algorithm works by iteratively finding every prefix w_i not contained in DB. For each such w_i it creates a record in NDB consisting of w_i with the remaining positions set to *. Figure 1 gives the pseudo code for the algorithm. An implementation of the prefix algorithm and all the other algorithms described in this paper is available from http://cs.unm.edu/~forrest/negdb.html. An example DB, $U - DB$ and the NDB produced by the prefix algorithm is given in Figure 2.

Lemma 1 *The prefix algorithm creates a database NDB that matches exactly those strings not in DB.*

Proof Every string not in DB must have a minimum length prefix that is not a prefix of any string in DB. Step three of the algorithm (Fig. 1) finds these prefixes and, for every such prefix, it appends a representation of every possible string with that prefix to NDB (step five). If a pattern—a specific sequence of 1s and 0s— is not present in DB's window w_{i+1} and its own prefix is not in w_i then it must have been inserted in NDB before. Step two initializes W_0 so that the first iteration considers every pattern absent from DB.

```
Prefix algorithm($DB$)
Let $w_i$ denote an $i$-bit prefix and $W_i$ a set
of $i$-length bit patterns.
1.  $i \leftarrow 0$
2.  Set $W_i$ to the empty set
    Repeat
3.      Set $W_{i+1}$ to every bit pattern not present in
        $DB$'s $w_{i+1}$ but with prefix in $W_i$
4.      for each pattern $V_p$ in $W_{i+1}$\{
5.          Create a record using $V_p$ as its prefix
            with the remaining positions set to *
6.          Add record to $NDB$.\}
7.      Increment $i$ by one
8.      Set $W_i$ to every pattern in $DB$'s $w_i$
9.  Until $i = l$.
```

Fig. 1 The Prefix algorithm outputs a negative database NDB of size $O(l \cdot |DB|)$ representing the strings in $U - DB$. See Figure 2 for an example input/output of the Prefix algorithm.

Theorem 1 *The negative data set* $(U - DB)$ *can be represented using* $O(l \cdot |DB|)$ *records.*

Proof For every window of size i there are at most $|DB|$ "negative" records created and inserted in NDB (steps 4–6). The number of windows is at most l (step 9) therefore, the number of negative records is $O(l \cdot |DB|)$.

The NDB produced by the prefix algorithm has some interesting properties. For example, each string in $U - DB$ is matched by exactly one NDB record. This non-overlapping property allows NDB to support more powerful queries than simple membership, as shown in Section 4.1. Consider, for instance, the NDB produced by the prefix algorithm from the banking database with tuples <Name, Transaction type, Date> discussed in Section 1. It is easy to determine how many transactions "John Doe" made on "01/01/2003" by selecting, from NDB, all records that match the name and date, and then counting the number of *s in the transaction field to determine how many strings a particular negative record represents (by exponentiating 2 to this number). Since the records are non-overlapping, the addition of the number of strings each such record represents amounts to the total transactions *not* made by John Doe in 01/01/2003. Subtracting this quantity from the total number of possible transactions for a specific name on a specific date (given by exponentiating 2 to the number of bits in the Transaction field) will yield the desired total. This is particularly interesting if the records in NDB are distributed among several parties, as no proper subset of the parties can compute this value.

In general, we would like some inferences to be hard (e.g., inferring the original DB from NDB) and other inferences to be easy, depending on the application (e.g., finding certain kinds of correlations in DB as in the above example). In the following section, we focus on the question of how easy it is to recover the original DB from NDB.

DB	$U - DB$	NDB	c-keys	$RNDB$
0001	0000	11**	11**	11**
0100	0010	001*	0*1*	0*1*
1000	0011	011*	*11*	1110
1011	0101	0000	00*0	*111
	0110	0101	*1*1	00*0
	0111	1001	1*01	*1*1
	1001	1010	**10	0101
	1010			1*01
	1100			**10
	1101			*010
	1110			
	1111			

Fig. 2 Column 1 gives an example DB, column 2 gives the corresponding $U-DB$, column 3 gives the corresponding NDB generated by the prefix algorithm, column 4 presents some possible c-keys extracted from NDB, and column 5 gives an example output of $RNDB$(see Section 5).

3 Reversibility

In Section 2.1 we presented an algorithm for generating NDB that demonstrates the feasibility of a negative representation. We now turn our attention to one property of negative representations—the difficulty of inferring the positive database from the entire negative database. First we establish that the representation described in Section 2 is potentially difficult to reverse (it is an $\mathcal{NP}$-hard problem), and in Section 5 we present an algorithm aimed at producing hard-to-reverse instances.

Reconstruction of DB from NDB is $\mathcal{NP}$-hard in the following sense[1].

Definition 1 Self Recognition (SR):
INPUT: A set $U - DB$ of binary strings represented by a collection NDB of length l strings over the alphabet $\{0, 1, *\}$, and a candidate self set DB.
QUESTION: Does NDB represent the self set DB?

We establish that SR is $\mathcal{NP}$-hard. Note that NDB represents an arbitrary set $U - DB$, and we do not specify how it was obtained. First we establish the $\mathcal{NP}$-completeness of the following problem.

Definition 2 Non-empty Self Recognition (NESR):
INPUT: A set $U - DB$ of binary strings represented by a collection NDB of length l strings over the alphabet $\{0, 1, *\}$.
QUESTION: Is DB nonempty? That is, is there some string in $U = \{0, 1\}^l$ not matched by NDB?

Theorem 2 *NESR is $\mathcal{NP}$-complete.*

Proof NESR is clearly in $\mathcal{NP}$. (If we guess a string, it is easy to verify that it is not matched, and thus a member of DB, by comparing it against every record in NDB.)

[1] For historical reasons we sometimes refer to DB as Self.

The $\mathcal{NP}$-completeness of NESR is established by transformation from 3-SAT. Start with instance $\mathbf{I}$ of 3-SAT. Let X be the set of variables $\{x_i\}$, and suppose l is the number of variables. The constructed instance of NESR will be over length l strings. Each clause $\{L_i, L_j, L_k\}$ in $\mathbf{I}$ (L_i is a literal, which is either x_i or x_i complement) creates a length l string in NDB as follows. All positions other than i, j, or k contain $*$. Position i contains 0 if L_i is x_i and contains 1 if L_i is $\bar{x_i}$ (complemented x_i). A similar construction is used for the other two literals L_j and L_k in this clause. Figure 3 shows an example of this mapping.

Claim: There exists a truth assignment satisfying $\mathbf{I}$ if and only if there exists a string in $U = \{0,1\}^l$ not matched by NDB (and therefore in DB). In the following, if $\mathcal{A}$ is a truth assignment to the variables in X, $S(\mathcal{A})$ is the string in U obtained by setting the i^{th} bit to 1 if $\mathcal{A}$ assigns $x_i = T$ and the i^{th} bit to 0 if $\mathcal{A}$ assigns $x_i = F$.

We have:
$\mathcal{A}$ satisfies $\mathbf{I}$
$\iff$ for every clause $C_q = \{L_i, L_j, L_k\}$, at least one literal is satisfied
$\iff$ $S(\mathcal{A})$ fails to match at least one of the bits i, j, k of the q^{th} member of NDB (generated from C_q), because uncomplemented literal L_i generates 0 in the i^{th} position and complemented L_i generates 1 in i^{th} position, and similarly for L_j, L_k
$\iff$ $S(\mathcal{A})$ is in DB.

Corollary 1 *NESR is $\mathcal{NP}$-complete even if every record of NDB contains exactly three defined positions.*

Proof Our transformation always produces such an instance of NESR.

Corollary 2 *Empty Self Recognition (ESR, the complement of NESR, answers YES if and only if NDB represents the empty set) is $\mathcal{NP}$-hard.*

Proof Trivial Turing transformation from NESR.

Theorem 3 *Self Recognition (SR, defined above) is $\mathcal{NP}$-hard.*

Proof We have established this to be the case even when the candidate self set DB is empty, and even when every member of NDB contains exactly three defined positions.

4 Applications

In this section we give two examples of how a negative representation might be useful. First, we discuss queries against a negative database, a subject whose full treatment is left for future work. A second possible application involves distributed negative databases, where we use set intersection as an example

Boolean Formula		NDB
(x_1 or x_2 or $\bar{x_5}$) and		00**1
($\bar{x_2}$ or x_3 or x_5) and		*10*0
(x_2 or $\bar{x_4}$ or $\bar{x_5}$) and	$\Rightarrow$	*0*11
($\bar{x_1}$ or $\bar{x_3}$ or x_4)		1*10*

Fig. 3 Mapping SAT to NDB: In this example the boolean formula is written in conjunctive normal form (CNF) and is defined over five variables $\{x_1, x_2, x_3, x_4, x_5\}$. The formula is mapped to an NDB where each clause corresponds to a record and each variable in the clause is represented as a 1 if it appears negated, as a 0 if it appears un-negated and as a $*$ if it does not appear in the clause at all. It is easy to see that a satisfying assignment of the formula such as $\{x_1=$ FALSE, $x_2=$ TRUE, $x_3=$ TRUE, $x_4=$ FALSE, $x_5=$ FALSE $\}$ corresponding to string 01100 is *not* represented in NDB and is therefore a member of DB.

4.1 Queries

Using the representation described above, negative databases consist of a set of strings defined over $\{0, 1, *\}^l$. Queries to such databases are also expressed as strings defined over the same alphabet. Using the banking example from Section 1, the binary string representing "John Doe, withdrawal, 01/01/2003" is interpreted as the query Q: Did John Doe make a withdrawal on 01/01/2003? And the ternary string representing "John Doe, *, 01/01/2003" is interpreted as the query Q': What transactions did John Doe make on 01/01/2003?

When a query Q consists only of defined positions, i.e. it has no *s, we refer to it as an authentication or simple membership query. Answering such a query is straightforward as it is necessary to ascertain only if Q is matched by any one of the strings in NDB (matching is described in Section 2). On the other hand, if Q contains an arbitrary number of unspecified positions, answering it is equivalent to asking whether the corresponding SAT formula has any satisfying assignments when an arbitrary number of its variables have pre-assigned truth values. This remains an $\mathcal{NP}$-hard problem for arbitrary sets of pre-assigned truth values. This contrasts with a positive database DB, where the records are stored explicitly and answering such queries takes time proportional to the size of DB.

For example, consider the query Q': What transactions did John Doe make on 01/01/2003? and the corresponding string defined over $\{0, 1, *\}$. If Q' is issued to DB, and computed by comparing it against each entry of DB, it will return only those strings that match the specified fields, even though Q' might actually represent an exponential number of strings. However, if Q' is issued to NDB, it will be necessary to find which of all the possible strings of length $l = 400$ whose defined positions correspond to "John Doe" and "01/01/2003" that are *not* in NDB and output them. It is an $\mathcal{NP}$-hard problem to accomplish this under our representation of NDB for an arbitrary choice of defined positions. Note, however, that it is possible to construct NDBs with specific structures for which this query can be answered efficiently, as discussed in Section 2.1. Intuitively, what makes some queries inefficient is not the size of NDB, as it is only polynomially larger than DB, but the fact that a single element of U, a single tuple, can be represented by several NDB entries and that a single NDB entry represents several tuples. This makes it difficult to determine, in general, if there are even any instances at all of a given field in DB.

In summary our representation scheme opens the door for negative databases that naturally restrict the type of information that can be retrieved efficiently, limiting queries to the authentication class; queries of an exploratory nature will, in general, be intractable. Several applications may profit from having databases that support only this limited type of queries, an example is presented in the following section.

A longer term goal is to control this complexity boundary, either through a deeper understanding of the existing representations or by devising new ones. This would allow us to support a limited set of queries (say, those allowed by law) and prevent arbitrary exploratory searches.

4.2 Set Intersection

One potential use of negative databases is for privately computing the intersection of several sets. This operation has applications in many domains such as recommender systems aimed at matching sets of preferences or, for example, finding the common entries in a collection of watch-lists. Due to the inherent properties of negative databases it is possible to perform these computations while hiding, at the same time, some potentially sensitive information.

Consider the banking example from Section 1 where the database has the tuples <Name, Transaction type, Date>. Suppose there are n banks, each an owner of a database DB_i, that wish to establish as part of some money laundering investigation which items they have in common, i.e. $\{DB_1 \cap \cdots \cap DB_n\}$, without revealing the totality of their databases or their cardinality. If each party produces a (hard to reverse) negative database NDB_i representing all records not in their DB_i to share with the other parties (we are assuming that all parties encode their information in exactly the same way), the i^{th} bank can compute the set intersection by simply establishing which of the entries of its database DB_i are not matched by any string in $\{NDB_1 \cup \cdots \cup NDB_n\}$, i.e. $DB_i - \{U - DB_1 \cup \cdots \cup U - DB_n\}$. An operation that can be carried out efficiently as discussed in Section 4.1. It is easy to see, using De Morgans's Laws, that $x \in \{DB_1 \cap \cdots \cap DB_n\} \leftrightarrow x \notin \neg\{DB_1 \cap \cdots \cap DB_n\} \leftrightarrow x \notin \{U - DB_1 \cup \cdots \cup U - DB_n\}$

This simple scheme conceals the cardinality of each party's database because, as discussed in Section 3, it is $\mathcal{NP}$-hard to enumerate DB given its negative representation NDB. However, this scheme does not prevent party j from testing if an arbitrary string x is a member of DB_i, regardless of it being in the intersection or not; as such, the proposed set-up could leak unintended information.

Alternatively, assume the existence of a protocol that allows us to anonymously create $\{NDB_1 \cup \cdots \cup NDB_n\}$ (for instance, using anonymous routing [18,52] and the operations described in Section 5.2) in such a way that it is infeasible to determine which party contributed which strings. The resulting negative database can only be used to retrieve the intersection of the DB_is, regardless of whether it is hard to reverse or not (if it is hard to reverse a DB_i would be necessary to obtain it). This example illustrates how the characteristics of a negative representation can be exploited to naturally secure certain operations.

5 Negative Database Algorithms

The prefix algorithm presented in Section 2.1 is simple and demonstrates that a compact negative representation NDB can be obtained from DB. Although we have shown in Section 3 that the general problem of reversing a given set NDB to obtain DB is $\mathcal{NP}$-hard, using the simple prefix algorithm to obtain NDB from DB raises two concerns regarding privacy: (a) The prefix algorithm produces only an easy subset of possible NDB instances, and (b) If the action of the prefix algorithm (or any algorithm) that produces NDB from DB could be reproduced by an adversary, then the adversary could easily decide for a given NDB and candidate DB whether NDB represents $U - DB$. The two concerns are of course related, for if an algorithm were capable of producing only one NDB for each DB it is given as input, the image of the algorithm could not define an $\mathcal{NP}$-hard set of instances of NESR.

In this section we present algorithms that address both of these concerns. The section is divided into two subsections, the first addresses how to create an initial negative database while the second deals with how it can be updated to reflect changes in the composition of DB. In addition, each subsection analyzes the algorithm's correctness and examines some of its properties.

5.1 Initialization

The $RNDB$ algorithm in Figure 4 takes as input a positive database DB (which might be initially empty) and outputs a negative database NDB (chosen probabilistically) that exactly matches $U - DB$. Its basic strategy is similar to that of the prefix algorithm in that, for a given permutation π—an ordering of the bit positions of a string—applied to every string in DB, it finds every prefix V_p not present in $\pi(DB)$ (see steps 0,2,3,4). For every such prefix, the algorithm randomly chooses an additional $0 \leq n \leq O(log_2(l))$ positions and creates 2^n (linear in l) strings with V_p and the additional n positions set to every possible bit assignment (steps 7,8,9)(see Lemma 2 below). This allows us to create strings that have specified bits beyond the prefix but still limit the size of the resulting NDB. Function Pattern Generate (Figure 5) replaces some of the specified bits by * symbols, taking care that no string in DB is matched by the operation (see Definition 3 below).

```
Randomize_NDB(DB,l)
Let $w_i$ denote an $i$-bit prefix and $W_i$ a set
of $i$-length patterns.
0.  Find a random permutation $\pi$ and apply it to $DB$.
1.  Randomly select $1 \leq i \leq O(log_2(l))$
2.  Initialize $W_i$ to the set of every pattern of $i$ bits.
    Repeat
3.      Set $W_{i+1}$ to every pattern not present in
        $\pi(DB)$'s $w_{i+1}$ but with prefix in $W_i$
4.      for each pattern $V_p$ in $W_{i+1}$ {
5.        Randomly choose $1 \leq j \leq O(l)$
6.        for $k = 1$ to $j$ do {
7.          Randomly select an additional
            $0 \leq n \leq O(log_2(l))$ distinct positions.
8.          for every possible bit assignment $V_q$ of the
            selected positions (a total of $2^n$ patterns){
9.            $V_{pe} \leftarrow V_p \cdot V_q$
10.           $V_{pg} \leftarrow$Pattern_Generate($\pi(DB), V_{pe}$)
11.           Append $\pi'(V_{pg})$ to $NDB$.}}}[a]
12.     Increment $i$ by one
13.     Set $W_i$ to every pattern in $\pi(DB)$'s $w_i$
14. Until $i = l$ or $W_i$ is empty.

[a] $\pi'$ denotes the inverse permutation of $\pi$
```

Fig. 4 The Randomize_NDB ($RNDB$) algorithm randomly generates a negative database representing the strings in $U - DB$.

```
Pattern_Generate($DB$, $V_{pe}$)
1.  Find a random permutation $\pi$.
    Let $n \leftarrow |V_{pe}|$
2.  for $i = 1$ to $n$ do {
3.      Construct a pattern $\pi(V_{pe})^\dagger$ with all but the $i^{th}$ bit from $\pi(V_{pe})$
4.      if $\pi(V_{pe})^\dagger$ not in $\pi(DB)${
5.        $\pi(V_{pe}) \leftarrow \pi(V_{pe})^\dagger$
6.        Keep track of the value of the $i^{th}$ bit in a set indicator
          vector (SIV) }}
7.  Randomly choose $0 \leq t \leq |SIV|$
8.  $R \leftarrow t$ randomly selected bits from SIV
9.  Create a pattern $V_k$ using $\pi(V_{pe})$, the bits indicated by $R$ and *
    symbols in the remaining positions.
10. Return $\pi'(V_k)$. [a]

[a] $\pi'$ is the inverse permutation of $\pi$.
```

Fig. 5 Pattern_Generate produces a string over $\{0, 1, *\}$ that matches V_{pe} without matching any string in DB.

5.1.1 Correctness

Definition 3 A string y is subsumed by string x if and only if every string matched by y is also matched by x. A string x obtained by replacing some of y's defined positions with don't cares, subsumes y.

Lemma 2 *A set of 2^n distinct strings that are equal in all but n positions match exactly the same set of strings as a single string with those n positions set to the don't care symbol.*

Lemma 3 *Pattern_Generate(DB,V_{pe}) outputs a string that matches every string matched by the input pattern V_{pe} without matching any other strings in DB.*

Proof To see that *Pattern_Generate* (as shown in Figure 5) produces a string that matches everything V_{pe} matches, it suffices to note that the output string specifies a subset of the positions set in the input pattern V_{pe}: lines 1–6 discard some of the positions that comprise V_{pe}, while lines 7–9 reinstate some of them (see Definition 3).

Additionally, the subpattern found in lines 1–6 (a c-key according to Definition 4 in Section 5.1.2) is guaranteed not to match any string in DB (lines 3–4). This subpattern is included in the final string output by the function, ensuring it will not match any string in DB.

Theorem 4 *The Randomize_NDB algorithm, under any sequence of random choices, produces an NDB that exactly represents $U - DB$.*

Proof Let ns_j be any string in $U - DB$ and let i be the length of the smallest prefix V_p of ns_j that is absent from DB under permutation π. The algorithm will find this prefix at iteration i (line 3) and will insert a series of strings into NDB that match the same strings as V_p as follows: Lines 7–11 create a collection of strings that subsume V_p by augmenting it with additional positions (lines 7–9 and Lemma 2) and assigning every possible pattern to these positions. Then, for each augmented pattern, function Pattern_Generate (line 10) creates a string that subsumes it without matching anything in DB (see Lemma 3). The resulting string is finally inserted into NDB (line 11).

In the case where DB is empty, lines 1–3 will consider the strings represented by every possible pattern of length $i + 1$ in the $i + 1$ length prefix (under permutation π), which encompasses all of U. Lines 4–11 insert the appropriate strings into NDB as discussed above. The function iterates once and exits.

5.1.2 Properties

Section 3 presents a transformation from 3-SAT to NDB, and in what follows we will use the formalisms interchangeably. In particular, DB and sets of assignments will be used interchangeably, NDB and formula ϕ will be used interchangeably, and the output of the algorithms to be presented in this section can be viewed either as strings in NDB or clauses in ϕ. For this reason we restrict clauses in ϕ to have no repeated variables.

The algorithm presented in Section 5.1 has the flexibility, by manipulating some of its parameters, to produce NDBs or SAT formulae with varying structures (see instance-generation models [45,13,14]). The following are some properties of the outputs it is able to produce.

Definition 4 A c-key is bit pattern not present in DB with no extraneous bits: A c-key defines a minimal pattern in that the removal of any bit yields a pattern in DB (see Figure 2). A $\bar{c}$-key is the complement of a c-key.

Lemma 4 *Let DB be a set of assignments and ϕ a CNF formula. ϕ is satisfied by every $x \in DB$ if and only if every clause C_q in ϕ contains a $\bar{c}$-key with respect to DB.*

Proof Suppose clause C_q of ϕ contains a $\bar{c}$-key. Then, by Definition 4, no $x \in DB$ contains the complement pattern of a $\bar{c}$-key. Each $x \in DB$ contains at least one bit appearing in $\bar{c}$-key which satisfies the corresponding literal and therefore satisfies C_q.

Now assume each $x \in DB$ satisfies each clause of ϕ (that is, each x is a satisfying truth assignment for ϕ). Suppose to the contrary, that some clause C_q does not contain a $\bar{c}$-key. Then, the complement pattern of $\bar{c}$-key appears in DB, and in particular in at least one $x \in DB$. But then x contains no bit appearing in $\bar{c}$-key, thus failing to satisfy each of the corresponding literals in C_q. This contradicts our original supposition, hence, it must be that every clause C_q contains a $\bar{c}$-key.

Lemma 5 *For every possible clause satisfied by DB contained in the input pattern V_{pe}, there is some execution of Pattern_Generate (Fig. 5) (with an appropriate sequence of random choices) that will generate it.*

Proof Let C_q be a clause satisfied by DB and P_q its corresponding bit pattern (see Figure 3 for the mapping). Suppose P_q is contained in the input pattern V_{pe}, then by Lemma 4 it must have as a subpattern some c-key K. For every pattern V_{pe} and every c-key K contained in V_{pe}, there exists a permutation π such that K occupies the $|K|$ rightmost bit positions of $\pi(V_{pe})$ (step 1). The algorithm proceeds by discarding one by one, from left to right, every bit it examines for as long as there is a c-key present within the remaining subpattern (steps 2–6). It follows that since K is a c-key and occupies the $|K|$ rightmost positions of $\pi(V_{pe})$ that K is the pattern that will be found[2]. Steps 7–9 of the algorithm generate a string containing K plus, by the appropriate random choice, the additional bits that comprise C_q.

Lemma 6 *For every clause satisfied by DB there is at least one string in $U - DB$ that contains the corresponding pattern.*

Proof Suppose C_q is a clause satisfied by DB and P_q the corresponding bit pattern, then by Lemma 4 C_q has a $\bar{c}$-key and P_q a c-key K. By the definition of c-key (Definition 4) there is no string in DB with K as a subpattern, hence every string with K as a subpattern must be in $U - DB$, including the one containing P_q.

Theorem 5 *The RNDB algorithm, during any execution, can produce any clause with $O(log(l))$ or fewer literals that is satisfied by DB.*

[2] Note that it is not required for the c-key to be contiguous or to occupy the rightmost bits to be found. It is only convenient to focus on this case for the proof.

Proof Let C_q be a clause of $k \leq O(log(l))$ literals satisfied by DB and P_q its corresponding bit pattern. For each P_q there is at least one string N_c in $U - DB$ that contains it (Lemma 6). String N_c, under permutation π, has a prefix of length i that is not present in DB which will come under consideration at iteration i of the algorithm (line 3). Suppose m of the k bits of P_q are included in the i length prefix of N_c, the remaining $k - m$ positions will be set in steps 7–8 by the appropriate random choice and the string corresponding to C_q will be found by Pattern_Generate (Lemma 5).

The cycle of line 5 ensures that each prefix is considered $O(l)$ times allowing any particular clause contained within a string with that prefix to be found independently.

Corollary 3 *The RNDB algorithm can produce any sequence of $O(l)$ clauses with $O(log(l))$ literals that are satisfied by DB as part of its output.*

Proof Theorem 5 states that any clause satisfied by DB, can be generated during any execution of the algorithm. It follows that, since the algorithm can generate formulas with $O(l)$ clauses, it can generate any sequence of $O(l)$ clauses that are satisfied by DB as part of its output.

It is important to note that the $RNDB$ algorithm is unable to produce every (polynomial size) formula (in polynomial time) that is satisfied exactly by DB. In fact, it can be shown that there is no efficient algorithm that, given DB as input, can generate all and only formulae that are exactly satisfied by DB, unless $Co\mathcal{NP} = \mathcal{NP}$. We saw, however, that the algorithm can generate every formula of a given length that is satisfied exactly by DB together with clauses that are superfluous[3] (Corollary 3).

We have shown in [22] that the image of $RNDB$ algorithm does in fact define an $\mathcal{NP}$-hard problem as a function of the size of the resulting NDB, albeit not necessarily as a function of the size of the original DB. Further, given that $\mathcal{NP}$-hardness is a worst case analysis, this property alone is not sufficient to guarantee that a negative database is hard to reverse in practice.

The challenge is to generate databases that are hard to reverse on average. In section 3 we discussed the isomorphism between boolean formulas and negative databases. This relationship suggests that results from this discipline can be leveraged for our purposes; in particular, the SAT community has extensive analysis on what makes for hard SAT instances. For instance, formulas with the right ratio of clauses to variables and formulas with the right statistical distribution of literals tend to be hard [45,1,36]. The algorithms presented above provide the flexibility to induce the corresponding structures on our negative databases.

We believe, however, that there are many applications where even if it is infeasible to provide full cryptographic protection, some degree of protection is important. Examples include data collection (such as surveys where the act of answering a survey question cannot be encrypted), fingerprint databases (where exact matches are unlikely, so encryption could be problematic), or

[3] This observation implies that identifying superfluous clauses is an $\mathcal{NP}$-hard problem itself.

sensor networks where distributed negative databases could reduce the risk of individual sensors being compromised.

Finally we note that Pattern_Generate runs in time $O(l \cdot |DB|)$ and that the Randomize_NDB algorithm outputs a database with $O(l^2|DB|)$ entries in $O(l^3|DB|^2)$ time.

5.2 Updates

We now turn our attention to modifying the negative database NDB once it has been initialized. We review three operations: Insert, Delete and Clean-up, initially introduced in [21]. It is worth mentioning that the meanings of the insert and delete operations are inverted from their traditional sense, since we are storing a representation of what is *not* is some database DB. For instance, using the banking example of Section 1, the command "insert <John Doe, withdrawal, 01/01/2003> into DB" is implemented as "delete <John Doe, withdrawal, 01/01/2003> from NDB" and the request "delete <John Doe, deposit, 01/02/2003> from DB" executed as "insert <John Doe, deposit, 01/02/2003> into NDB".

The core operation for the procedures, named Negative_Pattern_Generate (Figure 6), creates a string over $\{0, 1, *\}^l$ that subsumes x and matches nothing else in DB. Its functionality is similar to that of Pattern_Generate (Figure 5) and could be replaced by it. However, the difference is that Negative_Pattern_Generate does not need DB to be available, a potentially useful feature. This variation is reflected in lines 3–5 where extracting a subpattern from input x is accomplished by determining if replacing a specified bit in x by a * yields a string that is represented by $NDB \cup \{x\}$. [4] Owing to the similarity between procedures, the proof that Negative_Pattern_Generate is correct is very similar to Lemma 3 and is therefore omitted.

5.2.1 Insert into NDB

The purpose of the insert operation is to cause the negative database to represent all the binary strings depicted by the input string $x \in \{0, 1, *\}$, i.e. to match every binary string matched x, together with those strings already represented by NDB (x might be a string with no * symbols at all). It is important to note that in order to insert a string x into NDB it would be sufficient to simply append to NDB. However, this would leave a record of the operation. In order to alleviate this, Insert may specify some additional positions, creating some additional strings to insert (see Lemma 2) (steps 6,7,8,9), and then select a subset of the total specified positions for the string to keep (step 10). The Insert operation may insert several strings (step 1) per input string x, it is important to note that all of these entries are expected to be different due to the random nature of adding extra positions (steps 6,7,8), as well as to the non-deterministic fashion in which specified positions

[4] Note that this subpattern does not necessarily constitute a c-key (it is easy to see that extracting c-keys form NDB is $\mathcal{NP}$-hard).

```
Negative_Pattern_Generate($NDB$, $x$)
1.  Create a random permutation $\pi$
2.  for all specified bits $b_i$ in $\pi(x)$
3.      Let $x'$ be the same as $\pi(x)$ but with $b_i$ complemented
4.      if $x'$ is subsumed [a] by some string in $\pi(NDB)$
5.          Keep track of the value of the $i^{th}$ bit in a set indicator vector (SIV)
6.          Set the value of the $i^{th}$ bit of $\pi(x)$ to the * symbol
7.  Randomly choose $0 \leq t \leq |SIV|$
8.  $R \leftarrow t$ randomly selected bits from SIV
9.  Create a pattern $V_k$ using the specified bits of $\pi(x)$, the bits of $R$,
    and * in the remaining positions.
10. return $\pi'(V_k)$ [b]
```

[a] See Definition 2 in Section 5.1.1.
[b] π' is the inverse permutation of π.

Fig. 6 Negative_Pattern_Generate. Takes as input a string x defined over $\{0, 1, *\}$ and a database NDB and outputs a string that matches x and nothing else outside of NDB.

```
Insert($x$, $NDB$)
1. Randomly choose $1 \leq j \leq O(l)$
2. $m \leftarrow$ number of unspecified bits in $x$
3. if $m > log_2(l)$
4.     $m \leftarrow log_2(l)$
5. for $k = 1$ to $j$ do
6.     Randomly select $0 \leq n \leq m$
7.     Randomly select from $x$, $n$ distinct unspecified bit positions
8.     for every possible bit assignment $V_p$ of the selected positions [a]
9.         $x' \leftarrow x \cdot V_p$
10.        $y \leftarrow$ Negative_Pattern_Generate($NDB$, $x'$) [b]
11.        add $y$ to $NDB$
```

[a] Note that the loop iterates 2^n times, when $n = 0$ x' should be the same as x.
[b] Note that Pattern_Generate(DB, x') can be used instead, provided DB is available.

Fig. 7 Insert into NDB.

become unspecified during the call to Negative_Pattern_Generate (step 10). Figure 7 shows the pseudocode for this operation.

Theorem 6 *Function Insert(x, NDB) outputs a negative database that exactly matches* $(U - DB) \cup \{x\}$.

Proof It follows directly from Lemma 2 and Lemma 3.

5.2.2 Delete from NDB

This operation removes a set of binary strings from NDB. The function first identifies all the NDB entries, D_x, that match x—the string to be removed—

and withdraws them from NDB (steps 1,2). If the operation were to stop here, there would likely be many strings besides x that are inadvertently deleted from NDB. To avoid this, the function determines for every string in D_x which strings it matches, other than x, and reinserts them into NDB (steps 4,5,6,7).

Figure 8 gives a general algorithm for this task.

Theorem 7 *Delete(x, NDB) outputs a negative database that exactly matches $U - (DB \cup \{x\})$.*

Proof Lines 1–2 identify the subset, D_x, of NDB that matches x and removes it from NDB. Note that there is no string in $NDB - D_x$ that matches any binary string matched by x.

Lines 3–7 reinsert all the strings represented by D_x except x: For each string y in D_x and for each of its unspecified positions (don't care symbols) there is a string y_i created which differs from x in its i^{th} position (line 6) and inserted into NDB (see Theorem 6). None of the new strings y_i match x.

If a string $z \in \{0,1\}^l$ other than x is matched by some $y \in D_x$ then z must have the same specified positions as y. Given that z is different from x it follows that it must disagree with it in at least one bit, say bit k, z will be matched by y'_k. Therefore only x is eliminated from NDB. Finally, observe that since y subsumes each new entry y'_i (see Definition 3) no unwanted strings are included by the operation.

```
Delete($x$, $NDB$)
1. Let $D_x$ be all the strings in $NDB$ that match $x$.
2. Remove $D_x$ from $NDB$.
3. for all $y \in D_x$
4.     for each unspecified position $q_i$ of $y$
5.         if the $i^{th}$ bit of $x$ is specified
6.             Create a new string $y_i$ using the specified bits of $y$ and the
               complement of the $i^{th}$ bit of $x$.
7.             Insert($y_i$, $NDB$.)
```

Fig. 8 Delete from NDB.

One important effect of the Insert and Delete operations is that they both cause NDB to grow, especially in the latter case when the number of new entries in NDB is a function of the number of entries matched by the strings to be deleted. To address this problem we introduce a clean-up operation designed to reduce the size of the negative database and thus reduce the number of entries expected to match any binary string.

5.2.3 Clean-up

The operation presented here (Fig. 9) takes as input a negative database NDB and outputs a negative database NDB' that represents exactly the same set of binary strings, and therefore, matches exactly those strings not in DB. The function includes a parameter τ (line 4) which is meant to drive the size of the resulting database. If the Insert operation introduces fewer than τ entries per call then Clean-up will not increase the size of NDB and will likely reduce it.

```
Clean-up($NDB$, $\tau$)
1. Randomly select a string $x$ from $NDB$.
2.     Find a subpattern $K$ of $x$ not found in any $DB$ string [a].
3.     Let $D_K$ be all strings in $NDB$ that have $K$.
4.     if $|D_K| > \tau$
5.         Remove $D_K$ from $NDB$;
6          Create a string $V_K$ of length $l$ with $K$ as
           a subpattern and the remaining positions set to *.
7.         Insert($V_K$, $NDB$)
```

[a] According to lines 1–6 of Fig.6 or lines 1–6 of Fig. 5.

Fig. 9 Clean-up. Outputs a negative database that represents the same strings as its input NDB with equal or fewer entries.

Theorem 8 *The output of Clean-up is a negative database that represents the same set of binary strings as its input NDB.*

Proof Lines 1–2 find a subpattern K of a string in NDB, such that no string in DB has that pattern (see Definition 4 Lemma 3) i.e. every string in $\{0,1\}^l$ with such a pattern must be represented in NDB. Line 3 finds all NDB entries D_K that exhibit this pattern, line 5 removes them. Only strings in $\{0,1\}^l$ that have K stop being represented in NDB, for if a string y is matched by D_K then it must also be matched by K. Therefore, the removal of D_K causes only strings with K as a subpattern to be excluded. Line 6–7 reinsert every string and only strings with K as a subpattern into NDB (see Theorem 6).

5.2.4 Properties

It was previously mentioned that Pattern_Generate could be used in place of Negative_Pattern_ Generate within the Insert and Delete operations. In the case of Clean-up, extraction of a minimal pattern (line 2) can be achieved with lines 1–6 of Pattern_Generate or Negative_Pattern_Generate depending on the availability of DB. If the former is used, it is easy to see that the resulting negative database preserves the properties of the $RNDB$ algorithm's output outlined in Section 5.1.2. On the other hand, if the latter is applied then it is not feasible to determine if a pattern constitutes a c-key, and therefore, the number of clauses that can possibly be generated will be restricted.

An important property of the Insert, Delete and Clean-up operations is that, in general, their application does not make the problem of reversing a given NDB any easier. Consider the following problem:

Definition 5 Self-Recognition-Pair (SR-Pair)

INSTANCE: (ϕ, S, ϕ', S') where ϕ is a SAT instance, S a set of assignments to ϕ, ϕ' is a SAT instance obtained by inserting or deleting an arbitrary assignment x and only x from ϕ by means of any polynomial time algorithm $\mathcal{A}$. S' is obtained by inserting or deleting x from S accordingly.

QUESTION: Is ϕ' exactly satisfied by S'?

Theorem 9 *SR-Pair is $\mathcal{NP}$-hard*

Proof We prove the theorem by reducing SAT to SR-pair. The proof is divided into the case in which $\mathcal{A}$ is used to insert a satisfying assignment x to ϕ and the case in which it is used to delete a satisfying assignment x from ϕ.

1. Insertion version of SR-Pair is $\mathcal{NP}$-hard.
   Given instance ϕ of SAT. Pick any assignment x. If x satisfies ϕ answer YES to instance ϕ of SAT. If x does not satisfy ϕ use $\mathcal{A}$ to create ϕ' that is exactly satisfied by the assignments which satisfy ϕ, union $\{x\}$. Then $(\phi, \{\}, \phi', \{x\})$ is a valid instance of the insertion version of SR-Pair and:
   ϕ is a NO instance of SAT $\iff$ $(\phi, \{\}, \phi', \{x\})$ is a YES instance of SR-Pair.
2. Deletion version of SR-Pair is $\mathcal{NP}$-hard.
   Given an instance ϕ of SAT. Pick any assignment x. If x satisfies ϕ answer YES to instance ϕ of SAT. Otherwise, x does not satisfy ϕ and use $\mathcal{A}$ to create ϕ' that is exactly satisfied by the assignments which satisfy ϕ, minus $\{x\}$ (note ϕ is logically equivalent to ϕ'.) Then $(\phi, \{\}, \phi', \{\})$ is a valid instance of the deletion version of SR-Pair and:
   ϕ is NO instance of SAT $\iff$ $(\phi, \{\}, \phi', \{\})$ is a YES instance of SR-Pair.
   We conclude by stating that there is a polynomial time reduction from SAT to SR-Pair and hence that SR-Pair is $\mathcal{NP}$-hard.

It follows that the application of the Insert, Delete and Clean-up operations doesn't make a difficult instance any easier to reverse. However, we emphasize that the practical reversal difficulty of a specific NDB depends on the heuristics used to solve it, and hence these operations can decrease or increase the actual time required by a given heuristic.

The complexity of the algorithms is as follows: Negative _Pattern _Generate runs in time $O(l \cdot |NDB|)$. Insert takes $O(l^3|NDB|)$ time if Negative _Pattern _Generate is used, or $O(l^3|DB|)$ if Pattern_Generate is employed and inserts $O(l^2)$ strings per call into NDB. The Delete operation runs in $O(l^4|NDB|^2)$ or $O(l^4|NDB||DB|)$ time depending on whether the negative or positive pattern generate procedures are used. Delete causes the addition of $O(l^2|NDB|)$ entries in NDB. The Clean-up time complexity is dominated by its call to Insert and has the same complexity. Note that these bounds are due, in great part, to the generality that the algorithms afford. It is expected that the production of hard NDB instances will require limiting some parameters which will, in turn, reduce the complexity of the operations.

Randomize_NDB({},4)	**Delete**(1111,NDB)	**Insert**(1111,NDB)
000*	000*	000*
001*	001*	001*
01*0	01*0	01*0
01*1	01*1	01*1
10*0	10*0	10*0
10*1	10*1	10*1
111*	110*	110*
110*	11*0	11*0
	*110	*110
		11

Fig. 10 Possible states of NDB after successive initialization, deletion and insertion of a string.

6 Related work

The algorithms presented in this paper are concerned with exact representations of $U - DB$—everything except the database. There is, however, a growing body of work dealing with representing data negatively in an imprecise way [24,40,20], where the negative image of DB might not represent $U - DB$ entirely. This representations have slightly different properties and are useful in some scenarios. An alternative representation for negative databases is investigated in [15]. The cited scheme relies on cryptographic guarantees for the data security and yields compact representations; however, it limits some of the manipulations on the data that the present proposal allows. More operations on negative databases are presented in [25].

There are several other areas of research that are potentially relevant to the ideas discussed in this paper. These include: encryption, zero-knowledge sets, privacy-preserving databases, privacy-preserving data-mining, query restriction, multi-party computation and negative data.

An obvious starting point for protecting sensitive data is the large body of work on cryptographic methods, e.g., as described in [53]. Some researchers have investigated how to combine cryptographic methods with databases [28, 27,6,56], for example, by encrypting each record with its own key.

Zero-knowledge sets were recently introduced in [44] and provide a primitive for constructing databases that have many of the same properties as negative databases, namely, the restriction of queries to simple membership. There are several differences between the two approaches. First, zero-knowledge sets are based on widely believed cryptographically secure methods. Second, zero-knowledge sets require a controlling entity for answering queries. The relaxation of this requirement allows negative databases to perform operations such as set intersection privately and efficiently. Finally, to date, there is no efficient way of updating a zero-knowledge set, while Section 5.2 gives efficient algorithms for on-line operations on negative databases. A similar construction to zero-knowledge sets is presented in [50] in which range queries such as "Are there any keys in $[a, b]$" are possible.

Cryptosystems founded on $\mathcal{NP}$-complete problems [26] have been proposed such as the Merkle-Hellman cryptosystem [43], which is based on the general knapsack problem. These systems rely on a series of tricks to conceal the existence of a "trapdoor" that permits retrieving the hidden information efficiently. However, almost all knapsack cryptosystems have been broken [49], and it has been shown [8,9] that if breaking such a cryptosystem is $\mathcal{NP}$-hard then $\mathcal{NP}$=$Co\mathcal{NP}$. In general, if a scheme based on a $\mathcal{NP}$-hard result, such as the one proposed here, is to be used in a privacy setting it will be necessary to study under what situations it does indeed produce hard to reverse instances and if these instances can be readily obtained. There is a large body of work regarding the issues and techniques involved in generating hard-to-solve $\mathcal{NP}$-complete problems [35,34,49,43] and in particular of SAT instances [45,13]. Much of this work is focused on the case where instances are generated without knowledge of their specific solutions. Efforts concerned with the generation of hard instances possessing some specific solution, or solutions with some specific property include [29,1]. Finally, the problem of learning a distribution, whether by evaluation or generation [38, 47], is also closely related to constructing the sort of databases in which we are interested.

Of particular relevance are one-way functions [31,48]—functions that are easy to compute but hard to reverse— and one-way accumulators [5,11] which are similar to one-way hash functions but with the additional property of being commutative. One key distinction between these methods and negative databases is that the output of a one-way function is usually compact, and the message it encodes typically has a unique representation. By representing data negatively, as described here, a single message has many possible encodings, an idea that is exploited in probabilistic encryption [33,7].

Multi-party computation schemes [57,32], in which complex operations across databases can be performed privately are relevant to our discussion, in particular, when they involve applications such as set intersection. Other approaches to set intersection include [39,55,46] where several protocols and data structures are introduced to perform the operation securely and efficiently.

In privacy-preserving data mining, the goal is to protect the confidentiality of individual data while still supporting certain data-mining operations, for example, the computation of aggregate statistical properties [4,3,2,16, 19,56,54]. In one example of this approach (ref. [4]), relevant statistical distributions are preserved, but the details of individual records are obscured. Negative databases are roughly the reverse of this approach, in that they support simple membership queries efficiently but higher-level queries may be expensive.

Negative databases are also related to query restriction [41,12,16,17,54], where the query language is designed to support only the desired classes of queries. Although query restriction controls access to the data by outside users, it cannot protect an insider with full privileges from inspecting individual records to retrieve information.

The term "negative data" sounds similar to our method, but is actually quite different. The deductive database model (e.g., [30] presents an excellent survey of the foundations of the model) supports in the intensional database (IDB) the negative representation of data. The objectives, mechanisms, and consequences here are quite different from our scheme. In a deductive database, traditional motivations for "negative data" include reducing space utilization, speeding query processing, and the specification and enforcement of integrity constraints.

There is a large body of work in finding compact representations of a set of binary strings or functions (for example, [37,51,42,10]). Our work differs in its need to obtain a compact representation of the complement of the input set without explicitly calculating it, for it may be exponentially larger than its counterpart. Also, the nature of our representation makes many operations, such as comparing whether two functions are equivalent, potentially difficult. This is in contrast to techniques whose objective is to find compact representations of Boolean formulas, while preserving the ability to perform a wide range of operations on their representations. However, some of the compaction schemes may be useful in future work for exploiting other properties of negative representations.

To summarize, the existence of sensitive data requires some method for controlling access to individual records. The overall goal is that the contents of a database be available for appropriate analysis and consultation without revealing information inappropriately. Satisfying both requirements usually entails some compromise, such as degrading the detail of the stored information, limiting the power of queries, or database encryption.

7 Discussion

In this paper we have established the feasibility of a new approach to representing information. Specifically, we have shown that negative representations are computationally feasible, that they can be difficult to reverse, and that some interesting operations can be performed on them. However, there are many important questions and issues remaining. Which classes of queries can be computed efficiently and which cannot? Our initial results address two extremes—the case of testing simple membership for a specific, single record and the case of reconstructing the entire positive database. We would like to understand the computational complexity at points across the spectrum between these two extremes, as well as understanding what computational properties are desirable in a privacy-protecting context. A related question involves the costs of database updates under our representation. We have investigated algorithms that perform inserts and deletes in polynomial time, and we showed theoretically what their impact is on the complexity of the resulting negative database. We also introduced an operation that takes as input a negative database NDB and outputs a negative database NDB' which matches exactly the same set of binary strings as NDB. We would like to investigate ways in which this operation can be used to explore other potentially hard instances. And, we believe that more efficient algorithms might be designed to make the method practical for large-scale databases.

Are there other useful representations of NDB? Once we understand more completely the computational properties of our current representations, we may be able to devise other representations whose properties are more appropriate for some applications.

In this paper we emphasized the irreversibility properties of negative databases, as a means of protecting the privacy of individual records and as a method for privately computing the intersection of sets owned by different parties. There are additional characteristics and applications which we intend to investigate in our future work, such as the properties of a negative database when it is partitioned into several fragments and the qualities of the operations afforded by it.

Finally, we are interested in inexact representations. The NDB representation is closely related to partial match detection [23] which has many applications in anomaly detection. We are interested in studying how those methods might be combined with NDB either for designing an adaptive query mechanism or for approximate databases.

8 Conclusion

In this paper we introduced the concept of negative representations of information and presented a specific instantiation of this idea called negative databases. We established that a negative database can be constructed in time polynomial in the size of its positive counterpart. We presented algorithms for creating and maintaining such a database and offered an analysis of their properties and the properties of the negative databases they produce. Further, we investigated one characteristic of negative databases, namely that given a negative database it is an $\mathcal{NP}$-hard problem to recover its positive image. We also showed that, even though reversing a negative database is hard, there are certain types of queries that can be carried out efficiently, and discussed how this property can be exploited to privately compute the intersection of two sets. In current work we are exploring how to make the representations and algorithms more practical, and we are exploring several applications that seem well-suited to negative representations.

In conclusion, although we have shown that negative representations of data are computationally feasible, and in some cases difficult to reverse, there are many possible avenues for future work. We are optimistic that by tailoring a negative representation to particular requirements we can address at least some of the problems presented by large collections of sensitive data.

9 Acknowledgments

The authors gratefully acknowledge the support of the National Science Foundation (CCR-0331580, CCR-0311686, and DBI-0309147), Defense Advanced Research Projects Agency (grant AGR F30602-00-2-0584), the Intel Corporation, and the Santa Fe Institute. F.E. also thanks Consejo Nacional de Ciencia y Tecnología (México) grant No. 116691/131686 for its financial support.

References

1. Achlioptas, D., Gomes, C., Kautz, H., Selman, B.: Generating satisfiable problem instances. In: Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI-00), pp. 256–261. AAAI Press, Menlo Park, CA (2000)
2. Adam, N.R., Wortman, J.C.: Security-control methods for statistical databases. ACM Computing Surveys **21(4)**, 515–556 (1989)
3. Agrawal, D., Aggarwal, C.C.: On the design and quantification of privacy preserving data mining algorithms. In: Symposium on Principles of Database Systems, pp. 247–255 (2001)
4. Agrawal, R., Srikant, R.: Privacy-preserving data mining. In: Proc. of the ACM SIGMOD Conference on Management of Data, pp. 439–450. ACM Press (2000). URL citeseer.nj.nec.com/agrawal00privacypreserving.html
5. Benaloh, J.C., de Mare, M.: One-way accumulators: A decentralized alternative to digital signatures. In: Advances in Cryptology—EUROCRYPT '93, pp. 274–285 (1994). URL citeseer.nj.nec.com/article/benaloh93oneway.html
6. Blakley, G.R., Meadows, C.: A database encryption scheme which allows the computation of statistics using encrypted data. In: Proceedings of the IEEE Symposium on Research in Security and Privacy, pp. 116–122. IEEE CS Press (1985)
7. Blum, M., Goldwasser, S.: An efficient probabilistic public-key encryption scheme which hides all partial information. In: G.R. Blakely, D. Chaum (eds.) Advances in Cryptology: proceedings of CRYPTO 84, *Lecture Notes in Computer Science*, vol. 196, pp. 289–302. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc. (1985)
8. Brassard, G.: A note on the complexity of cryptography. IEEE Transactions on Information Theory **25**(2), 232—233 (1979)
9. Brassard, G., Fortune, S., Hopcroft, J.E.: A note on cryptography and NP∩ coNP-P. Technical Report TR78-338, Cornell University, Computer Science Department (1978)
10. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24**(3), 293–318 (1992). URL citeseer.nj.nec.com/bryant92symbolic.html
11. Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: M. Yung (ed.) Advances in Cryptology – CRYPTO ' 2002, *Lecture Notes in Computer Science*, vol. 2442, pp. 61–76. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany (2002). URL http://www.springerlink.com/link.asp?id=yklb7xdvbbc0wwgy
12. Chin, F.: Security problems on inference control for sum, max, and min queries. J. ACM **33**(3), 451–464 (1986). DOI http://doi.acm.org/10.1145/5925.5928
13. Cook, S.A., Mitchell, D.G.: Finding hard instances of the satisfiability problem: A survey. In: Du, Gu, Pardalos (eds.) Satisfiability Problem: Theory and Applications, *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, vol. 35, pp. 1–17. American Mathematical Society (1997)
14. Crawford, J.M., Anton, L.D.: Experimental results on the crossover point in satisfiability problems. In: R. Fikes, W. Lehnert (eds.) Proceedings of the Eleventh National Conference on Artificial Intelligence, pp. 21–27. American Association for Artificial Intelligence, AAAI Press, Menlo Park, California (1993)
15. Danezis, G., Diaz, G., Faust, S., Käsper, E., C., C.T., Preneel, B.: Efficient negative databases from cryptographic hash functions. In: S. LNCS (ed.) Information Security Conference, vol. 4779, pp. 423–436 (2007)
16. Denning, D.: Cryptography and Data Security. AddisonWesley, Reading, MA (1982)
17. Denning, D., Schlorer, J.: Inference controls for statistical databases. Computer **16(7)**, 69–82 (1983)
18. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The second-generation onion router. In: In Proceedings of the 13th USENIX Security Symposium (2004)

19. Dobkin, D., Jones, A., Lipton, R.: Secure databases: Protection against user influence. ACM Transactions on Database Systems **4(1)**, 97–106 (1979)
20. Esponda, F.: Hiding a needle in a haystack using negative databases. In: Proceedings of the 10th Information Hiding Conference (2008)
21. Esponda, F., Ackley, E.S., Forrest, S., Helman, P.: On-line negative databases (with experimental results). International Journal of Unconventional Computing **1**(3), 201–220 (2005)
22. Esponda, F., Forrest, S., Helman, P.: Enhancing privacy through negative representations of data. Technical report, University of New Mexico (2004)
23. Esponda, F., Forrest, S., Helman, P.: A formal framework for positive and negative detection schemes. IEEE Transactions on Systems, Man and Cybernetics Part B: Cybernetics **34**(1), 357–373 (2004)
24. Esponda, F., Forrest, S., Helman, P.: Protecting data privacy through hard-to-reverse negative databases. International Journal of Information Security **6**(6), 403–415 (2007)
25. Esponda, F., Trias, E., Ackley, E., Forrest, S.: A relational algebra for negative databases. Tech. Rep. TR-CS-2007-18, University of New Mexico (2007)
26. Even, S., Yacobi, Y.: Cryptography and np-completeness. In: Proc. 7th Colloq. Automata, Languages, and Programming (Lecture Notes in Computer Science), vol. 85, pp. 195–207. Springer-Verlag (1980)
27. Feigenbaum, J., Grosse, E., Reeds, J.A.: Cryptographic protection of membership lists **9**(1), 16–20 (1992). URL ftp://cm.bell-labs.com/cm/cs/doc/91/4-12.ps.gz
28. Feigenbaum, J., Liberman, M.Y., Wright, R.N.: Cryptographic protection of databases and software. In: Distributed Computing and Cryptography, pp. 161–172. American Mathematical Society (1991)
29. Fiorini, C., Martinelli, E., Massacci, F.: How to fake an RSA signature by encoding modular root finding as a SAT problem. Discrete Appl. Math. **130**(2), 101–127 (2003)
30. Gallaire, H., Minker, J., Nicolas, J.: Logic and databases: a deductive approach. Computing Surveys **16:1**, 154–185 (1984)
31. Goldreich, O.: Foundations of Cryptography: Basic Tools. Cambridge University Press (2000)
32. Goldwasser, S.: Multi party computations: past and present. In: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing, pp. 1–6. ACM Press (1997). DOI http://doi.acm.org/10.1145/259380.259405
33. Goldwasser, S., Micali, S.: Probabilistic encryption. Journal of Computer and System Sciences **28**(2), 270–299 (1984). See also preliminary version in 14th STOC, 1982.
34. Impagliazzo, R., Levin, L.A., Luby, M.: Pseudo-random generation from one-way functions. In: Proceedings of the twenty-first annual ACM symposium on Theory of computing, pp. 12–24. ACM Press (1989). DOI http://doi.acm.org/10.1145/73007.73009
35. Impagliazzo, R., Naor, M.: Efficient cryptographic schemes provably as secure as subset sum. In: IEEE (ed.) 30th annual Symposium on Foundations of Computer Science, October 30–November 1, 1989, Research Triangle Park, NC, pp. 236–241. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA (1989)
36. Jia, H., Moore, C., Strain, D.: Generating hard satisfiable formulas by hiding solutions deceptively. In: AAAI (2005)
37. Karnaugh, M.: The map method for synthesis of combinational logic circuits. Trans. AIEE pp. 593–598 (1953)
38. Kearns, M., Mansour, Y., Ron, D., Rubinfeld, R., Schapire, R.E., Sellie, L.: On the learnability of discrete distributions. In: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing, pp. 273–282. ACM Press (1994). DOI http://doi.acm.org/10.1145/195058.195155
39. M. Freedman, K.N., Pinkas, B.: Efficient private matching and set intersection. In: Advances in Cryptology – Eurocrypt '2004 Proceedings, LNCS 3027, pp. 1–19. Springer-Verlag (2004)

40. de Mare, M., Secure, R.W.: Set membership using 3sat. In: Proceedings of the Eighth International Conference on Information and Communication Security (ICICS '06) (2006)
41. Matloff, N.S.: Inference control via query restriction vs. data modification: a perspective. In: on Database Security: Status and Prospects, pp. 159–166. North-Holland Publishing Co. (1988)
42. McCluskey, E.: Minimization of boolean functions. Bell System Technical Journal, pp. 1417–1444 (1956)
43. Merkle, R.C., Hellman, M.E.: Hiding information and signatures in trapdoor knapsacks **IT-24**, 525–530 (1978)
44. Micali, S., Rabin, M., Kilian, J.: Zero-knowledge sets. In: Proc. FOCS 2003., p. 80 (2003)
45. Mitchell, D., Selman, B., Levesque, H.: Problem solving: Hardness and easiness - hard and easy distributions of SAT problems. In: Proceeding of the 10th National Conference on Artificial Intelligence (AAAI-92), San Jose, California, pp. 459–465. AAAI Press, Menlo Park, California, USA (1992)
46. Morselli, R., Bhattacharjee, S., Katz, J., Keleher, P.: Trust preserving set operations. Tech. rep.
47. Naor, M.: Evaluation may be easier than generation (extended abstract). In: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, pp. 74–83. ACM Press (1996). DOI http://doi.acm.org/10.1145/237814.237833
48. Naor, M., Yung, M.: Universal one-way hash functions and their cryptographic applications. In: Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing: Seattle, Washington, May 15–17, 1989, pp. 33–43. ACM Press, New York, NY 10036, USA (1989)
49. Odlyzko, A.M.: The rise and fall of knapsack cryptosystems. In: C. Pomerance, S. Goldwasser (eds.) Cryptology and Computational Number Theory, *Proceedings of symposia in applied mathematics. AMS short course lecture notes*, vol. 42, pp. 75–88. pub-AMS (1990)
50. Ostrovsky, R., Rackoff, C., Smith, A.: Efficient consistency proofs for generalized queries on a committed database. In: ICALP: Annual International Colloquium on Automata, Languages and Programming, pp. 1041—1053 (2004)
51. Quine, W.V.: A way to simplify truth functions. American Mathematical Monthly pp. 627–631 (1955)
52. Reed, M.G., Syverson, P.F., Goldschlag, D.M.: Anonymous connections and onion routing. IEEE Journal on Selected Areas in Communications **16**(4), 482–494 (1998)
53. Schneier, B.: Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley and Sons, Inc., New York, NY, USA (1994)
54. Tendick, P., Matloff, N.: A modified random perturbation method for database security. ACM Trans. Database Syst. **19**(1), 47–63 (1994). DOI http://doi.acm.org/10.1145/174638.174641
55. Vaidya, J., Clifton, C.: Secure set intersection cardinality with application to association rule mining. To appear in Journal of Computer Security (2004)
56. Wayner, P.: Translucent Databases. Flyzone Press (2002)
57. Yao, A.: Protocols for secure computation. In: IEEE (ed.) 23rd annual Symposium on Foundations of Computer Science, November 3–5, 1982, Chicago, IL, pp. 160–164. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA (1982)