Hiding a Needle in a Haystack Using Negative Databases

Fernando Esponda

Department of Computer Science
Instituto Tecnológico Autónomo de México
Mexico City, Mexico
fernando.esponda@itam.mx

Abstract. In this paper we present a method for hiding a list of data by mixing it with a large amount of superfluous items. The technique uses a device known as a negative database which stores the complement of a set rather that the set itself to include an arbitrary number of garbage entries efficiently. The resulting structure effectively hides the data, without encrypting it, and obfuscates the number of data items hidden; it prevents arbitrary data lookups, while supporting simple membership queries; and can be manipulated to reflect relational algebra operations on the original data.

1 Introduction

Data confidentiality has always been a primary concern of individuals and organizations. Before the computer era, important documents were hidden or kept under lock and key. Getting access to them implied a guardian "server", a key, or knowing the document's whereabouts. Many of the same techniques apply today to information kept in a digital form; however, the variety and volume of today's data, together with the number of users and applications it must service, require greater robustness and flexibility in how data may be manipulated.

Consider, for instance, a list of credit-card numbers that is made available to several independent entities; for example, the list records numbers presumably involved in "suspicious" activities or, alternatively, the list of winners of some draw. The numbers in the list should remain confidential, but the presence or absence of a given number must be readily verified (in order to call the authorities, deny service, or present a prize).

Cryptographic hash functions are effective for creating a list of data items whose true identity cannot be easily determined but whose presence can be verified by anyone. Suppose, however, that learning the number of entries in the list is useful for an adversary, or that the data needs to be augmented with the card holders name, or the list restricted to only those cards issued by a specific financial institution (the first digits of the card indicate its issuer). A hashed list fails to conceal the number of items it contains and makes meaningful manipulations difficult, e.g., selecting a subset of the contents according to some

criterion. Alternatively, consider hiding the data by mixing it in with items that are structurally similar but meaningless—hiding a needle in a haystack. Finding a datum requires sifting through a large amount of chaff; the size of the data set is concealed; and the resulting set can be handled to reflect some manipulations on the hidden data. Verifying the presence of an item, however, requires special, secret knowledge of where to look. The quality of the concealment strongly depends on the nature and size of the set in which the items are hidden.

In this paper we propose a scheme that resembles a combination of both methods discussed above. It efficiently supports membership queries while making arbitrary fishing expeditions hard, it obfuscates the cardinality of the hidden set by including "garbage" entries, and the hidden data can be manipulated using some relational operations. The main difference is that the data is not hashed and can still be manipulated meaningfully, and that the amount of chaff that can be efficiently included with the data is very large. The data-structure we employ is known a negative database—a compressed version of the list containing all of the elements *not* in the original, positive database or list. Storing the complement of the list of interest allows us to include a large amount of superfluous data within it; intuitively, the more chaff the smaller the negative database.

Previous work on negative databases relied on the theoretical difficulty of "reversing" a negative database (deciding whether a negative database is empty or not is NP-complete) and on finding suitably hard instances for its security [12]. Our current proposal does not lean on this property, as it may very well be easy to find entries not included in the negative database; its security relies on the number of superfluous entries included alongside the data and on the infeasibility of retrieving only the valid items. Furthermore, the size of the representations presented here are dramatically smaller that those used in the cited work.

Sections 2 and 3 describe our proposal in detail and give it a theoretical treatment. In section 4 we outline a possible implementation and present the output of several experiments. We discuss our results and give some concluding remarks is Sect.6.

2 Description

The original data is a subset of the set of all binary strings of length t, U_d. Strings belonging to U_d are referred to as *text* strings throughout the document. Our strategy is to embed U_d within a larger universe U, such that U can be partitioned into a set of valid strings and a set of invalid strings denoted U_V and U_I respectively. The universe U is the set $U_d \times \{0,1\}^c$ of strings of length $l = t+c$; the additional c bits are referred to as the *code* and are used to distinguish valid from invalid strings. Accordingly, U_V contains only those text-code combinations that are deemed to be valid in a particular context (and U_I contains the rest of U).

The data to be concealed, the positive database DB, is a subset of U_V. The negative database, NDB, is a compact representation of $U - (DB \cup G)$, where G (the chaff) is a subset of $U - DB$ and includes strings that are not in DB but

that are nevertheless included in the positive image of NDB. The positive image of NDB, i.e., the binary strings not represented in DB, is denoted as DB'. Negative databases should meet the following desiderata:

- NDB must exclude all of DB
- NDB must be created efficiently
- The membership of any specific string in the set characterized by NDB should be easily determined
- G must contain an intractable number of different text strings and an intractable number of different code strings
- The number of strings in G from U_V must be marginal
- The number of strings in DB' within a Hamming Distance S_H from U_V must be insignificant. S_H is a security parameter that discourages using strings that are farther than S_H as starting points for exhaustive searches
- There must be no easy way to enumerate the valid strings NDB negatively represents without also enumerating an intractable number of invalid strings
- All strings in the universe U should be readily classifiable as valid or invalid

In the next section we describe an algorithm for creating negative databases and investigate the properties of the resulting NDBs. We then layout the characteristics the code should have to complete our scheme.

3 Generating a Negative Database

In this section we present an algorithm that outputs a negative database, NDB, when given as input the set of strings DB. First, a few definitions:

Positive Database: A positive database is a set of binary strings of length $l = t + c$—the original data, the data to be concealed, has length t and a code of length c is used to augment it

Negative record: Let $\mathbb{Z}_n$ be the set of non-negative integers less than n, e.g., $\mathbb{Z}_2 = \{0, 1\}$. A negative record is a k-tuple of pairs, (position,value), defined over $\mathbb{Z}_l \times \mathbb{Z}_2$. For l=10 and k=2 the 2-tuple $< (5, 0), (7, 1) >$ is a negative record with a 0 at position 5 and a 1 at position 7

Negative database: A negative database, denoted NDB, is a set of negative records

Matching: A negative record N_r is said to match a binary string x if and only if for every pair p in the k-tuple, $x[p.position] = p.value$, where $x[i]$ denotes the value of string x projected onto position i. We write $N_r \mathrm{M} x$ for a match and N_r DNM x for a mismatch

Membership: A string x is in NDB if and only if it is matched by at least one negative record

A negative database for positive database DB is such that no negative record in NDB matches a string in DB. There might be, however, some binary strings not in DB that are not matched by NDB; we denote the set of all strings not matched by any NDB entry as DB'.

There are a several algorithms in the literature for creating negative databases [8, 12, 9]. Moreover, since it was shown in [9] that negative databases are linked by a simple transformation to boolean satisfiability formulas, algorithms for generating formulas can be adapted to generate $NDBs$.

The algorithm presented in Fig. 1 was chosen for its simplicity and ease of analysis. One of our priorities is to have an algorithm with as few biases as possible that could potentially be used by an adversary. The current version is straight forward enough to avoid most of these concerns. The algorithm is similar to methods for generating SAT formulae, to techniques for intrusion detection systems [14], and to algorithms for creating digital credentials [8].

Input: $Size$, k, l, DB
Output: NDB
$NDB \leftarrow \emptyset$
while($|NDB| < Size$)
    Create a negative record N_r by selecting k distinct
    pairs from $\mathbb{Z}_b \times \mathbb{Z}_2$ uniformly at random, where $b = \lceil \log_2(l) \rceil$
    if $((N_r \notin NDB) \wedge (\forall x \in DB,\ N_r\ DNM\ x))$
        $NDB \leftarrow NDB \cup N_r$
Sort NDB

Fig. 1. Negative Database Generation Algorithm

The algorithm creates a negative database by selecting negative records uniformly at random from the space of possible records, keeping only those that do not match any DB entry. Each record, in turn, has the same number of position-value pairs (k, sometimes referred to as specified positions). The desired size of NDB is given as a parameter and plays an important role for our scheme. The last line of the algorithm requires sorting NDB; the details are purposely left unspecified as any deterministic ordering will suffice. It is important, however, that both the record order within NDB and the tuple order within each record be specified in order to erase the relative order in which they where created. We assume the use of a suitable pseudo random number generator with a large seed space and cycle.

We wish to create just enough negative records so as to match all valid strings not in DB ($U_V - DB$) and at the same time leave a large subset of U_I unmatched. If too many negative records are created DB' will be very close to DB and the task of retrieving an original data record simplified; on the other hand, if NDB is too small, DB' will include a large number of valid strings and the demarcation of DB will be lost—we aim for the inclusion of strings from $U_V - DB$ to be marginal. This requires not only having an NDB of the proper

size but for valid strings to be well distributed throughout the space. The code attached to each string, discussed in Sect. 3.2, accomplishes this.

The number of iterations of the algorithm's main loop is roughly $|NDB|e^{|DB|2^{-k}}$ (see Sect. 3.1, eq. 3). A more mindful version of the algorithm postpones eliminating repeated NDB entries until after the main loop (creating a repeated entry is unlikely) and completes NDB as needed. Taking this into account, the main effort in each iteration is searching through DB. The algorithm's asymptotic time complexity is $O(|NDB| \cdot e^{|DB|2^{-k}} \cdot SearchCost(DB))$, where $SearchCost(DB)$ is the cost of determining whether a potential NDB entry matches DB.

3.1 Properties

Assume that U_V is a set of strings selected independently and uniformly at random from U, that the strings in DB are selected independently and uniformly at random from U_V, and that NDB records are independent of each other. We analyze the properties of our scheme under this circumstances and in Sect. 3.2 discuss the properties the code should have to approximate them in a real scenario. In Sect. 4 we test a particular implementation and examine its results.

The probability of a set of independent detectors, NDB, not matching a particular string in $U - DB$, i.e., the coverage of NDB, is approximated by:

$$P_e = (1 - 2^{-k})^{|NDB|} \simeq e^{-|NDB|2^{-k}} \tag{1}$$

The number of entries in a negative database so as to achieve P_e is:

$$|NDB| \simeq -\ln(P_e)2^k \tag{2}$$

Notice how the number of entries in NDB does not depend on the length of DB strings. The number of bits per entry, however, does. Each NDB record requires $k\lceil \log_2(l) \rceil + k$ bits and the total number of bits in a NDB is $|NDB|(k\lceil \log_2(l) \rceil + k)$.

The size of NDB grows exponentially with the number of specified bits, k, per entry (see eq. 2); k, in turn, determines how easy it will be to generate a record that does not match any DB string. We define P_k as the probability that a randomly chosen negative record is a valid NDB entry:

$$P_k = (1 - 2^{-k})^{|DB|} \simeq e^{-|DB|2^{-k}} \tag{3}$$

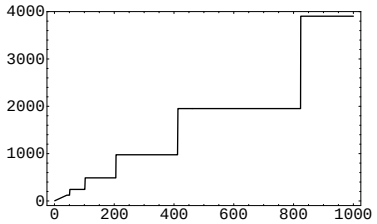
the value of k is given by:

$$k \simeq \frac{1}{\ln(2)}(\ln(|DB|) - \ln^2(P_k^{-1})) \tag{4}$$

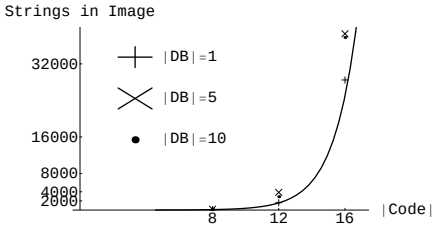
Along with P_e, k determines what size NDB will have and sets an upper bound on how big a DB can be depicted by as many NDB entries. DBs of any size up to this one, can be represented by NDBs with the same number of records and the same number of specified bits per record. The size of NDB leaks only an upper bound on the size of DB.

For fixed values of P_e and P_k the number of NDB entries grows linearly with the size of the positive database (see Fig. 2(a)):

$$|NDB| = \frac{\ln(P_e)}{\ln(P_k)}|DB| \tag{5}$$



(a) $|NDB|$ growth with respect to $|DB|$. The x axis represents $|DB|$ and the y axis $|NDB|$. The plot was made for a P_e=2^{-10} and a P_k=0.04. The resulting k value is rounded up yielding a stepwise growth in NDB



(b) The number of strings in DB' for a text of length 4, code lengths of 8, 12 and 16, and DBs of size 1, 5, and 10. The corresponding NDBs are of size 28, 28 and 56 (P_m = 0.5, k=3,3,4). The theoretical results are plotted as a solid line

Fig. 2. Growth behavior of DB, NDB, and DB'

The expected number of strings in NDB's positive image (denoted DB') is $|DB|$ plus the additional strings in $U - DB$ that are not matched by any NDB record:

$$\begin{aligned} |DB'| &= |DB| + P_e|U - DB| \\ &= |DB| + P_e|U_V - DB| + P_e|U_I| \end{aligned} \tag{6}$$

The expected number of superfluous valid strings included in the positive image is estimated as being $P_e(|U_V - DB|)$ and the expected number of invalid strings as $P_e(|U - U_V|)$ (see Fig.2(b) for an example). Notice that the former does not depend on the size of U while the latter increases as U grows for a given U_V. In what follows we describe a scheme that allows us to control the expected number of false positives by setting P_e, and the expected number of invalid strings by choosing the relative size between U and U_V.

3.2 The Code

Before creating a negative database, the original data must be augmented with a code so as to create a distinction between valid and invalid strings, and to disperse valid strings throughout the space from which DB' is drawn. Consider

attaching a unique, uniformly distributed random string to every string in U_d—the set from which the original data is drawn—and creating a NDB with the data to be hidden augmented with its corresponding code (DB). The resulting construction satisfies the first three points laid out in Sect. 2: NDB excludes all of DB; it is created efficiently; and verifying if a data point is in NDB is done by looking up its code and determining if the augmented string is matched by any of NDB's records.

Each negative record matches a subset of U—a hyper-sphere in hamming space—containing all binary strings with the given k positions set to the specified values (a total of 2^{l-k} strings). The task of the NDB generation algorithm is to randomly generates those spheres discarding the ones which include any string from DB. If the attached code is sufficiently long, the strings in U_V will be well distributed throughout U making it unlikely that a discarded ball includes a DB entry as well as a $U_V - DB$ string (see Fig. 3). If the algorithm generates an appropriate amount of records (Sect. 3.1 examines this issue), the total number of strings in G will be large with a small number of strings from U_V satisfying the next three properties from Sect. 2 (Sect. 3.2 considers this points). Finally, since the code bears no straightforward relation to the text, there is no efficient way of restricting the retrieval of strings from DB'—the reverse of NDB—to only strings in U_V.

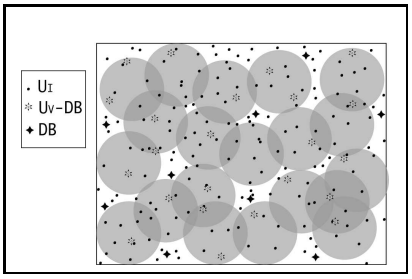


Fig. 3. Graphical depiction of our scheme. The gray circles symbolize the area covered (strings matched) by each NDB entry. The small dark dots are invalid strings, all the rest are valid. DB entries are shown as big dark diamonds. NDB matches most strings in the universe, leaving only DB and an intractable amount of invalid strings unmatched

The obvious drawback of this design is the intractable amount of bookkeeping required to distinguish valid from invalid strings (a independent randomly generated code must kept for each DB string). This idealized code, however, brings out the characteristics we want from our code and code generating function g:

- g is computed efficiently. g may be easily invertible

– A small change in the text leads to an arbitrarily large change in the code. For any two distinct strings x and y that are arbitrarily close (in Hamming distance) $g(x)$ and $g(y)$ are arbitrarily far apart
– There is a low probability of code collisions
– It is infeasible to distinguish a valid code from an invalid one without having its corresponding text. It is hard to determine the code having only a small subset of the bits in the text
– The probability of randomly generating a NDB entry for any string in $U_V - DB$ increases with the length of the code
– The resulting code is sufficiently large as to make it hard to recover valid strings from NDB.

An important point is that g may be easy to invert, as the security of the scheme relies on the challenge of finding a valid text-code combination, rather than on the difficulty of recovering the text given the code or vice versa. Finally, note that if a NDB of only codes is created, forgetting about the text altogether, we must additionally ensure that several codes do not decode to the same text, in order to avoid including unwanted valid strings in DB'.

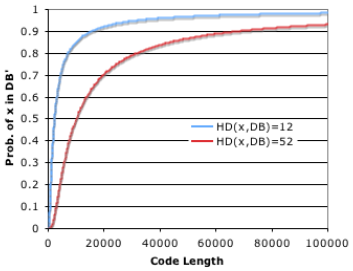
Code Length The length of the code influences the security of the scheme in two ways: First it determines the proportion of valid to invalid strings in DB'; the longer the code the more invalid strings are included and the more likely to retrieve an invalid string than a valid one in a single try. Second, the length of the code prescribes how many distinct text strings (and code strings) will be included in DB'. This speaks to how easy it is to obtain a valid string given a retrieved entry from DB'. An intractable amount of strings in DB' is not enough to guarantee security unless the number of distinct texts (and codes) is also intractable (it is easy to guide the search away from a few selected strings by including them in NDB (see[10])). The security of the present scheme relies on including an intractable number of strings with distinct texts and codes.

The amount of distinct texts is computed by first estimating the probability of including in DB' a particular string x that is a Hamming distance of h away from the closest string in DB. Let $HD(x, DB)$ be the smallest Hamming distance between a string x and the set of strings DB, and D_h be the number of strings in DB at a distance of h from x.

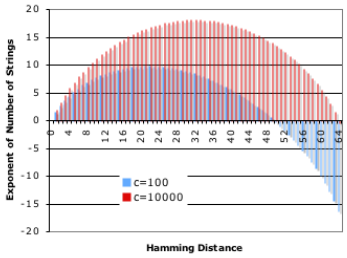
$$P(x \in DB' | HD(x, DB) = h) = (1 - 2^{-k}(1 - \frac{\binom{l-h}{k}}{\binom{l}{k}})^{D_h})^{|NDB|} \tag{7}$$

The probability that a DB' includes a given string that is a distance of h away from DB increases as the length of the code increases (see Fig. 4(a)). The expected number of distinct texts in DB' is given by:

$$\sum_{h=0}^{t} \binom{t}{h} P(x \in DB' | HD(x, DB) = h) \tag{8}$$



(a) Probability that text x is included in DB' as its attached code increases in length. $|DB| = 1$, x is 64 bits long, k=3, and $|NDB| = 366$.



(b) The logarithm (base 10) of the expected number of distinct texts in DB' for codes of length 100 and 10000. $|DB| = 1$, x is 64 bits long, k=3, and $|NDB| = 366$.

As can be seen from this analysis the number of texts in DB' that are close to DB are few in comparison to the total number of distinct texts included in DB' (see Fig. 4(b)). This makes finding a string close to DB increasingly unlikely as the code grows and prevents exhaustive searches from using retrieved strings as a starting point. Section 4 shows how close eq. 4(b) resembles the experimental results for DBs of size one and one hundred.

4 Implementation

This section presents a possible implementation of our scheme and explores its characteristics experimentally. The first step is to choose a code generating function that satisfies the requirements laid out in the previous sections. A straight forward option is to use a hash function and apply it repeatedly to achieve the desired code length. We chose MD5 [21] since it satisfies all of our desiderata, even though we do not require that it be difficult to recover the text using only the code. In particular, we use MD5 to ensure that a few bits of the code cannot be used to determine the text and vice versa.
In order the generate codes of the desired size MD5 is applied repeatedly to a string $x \in U_d$ as described in Fig. 4.

We conducted experiments for positive databases with one and one hundred elements each, with text strings of 64 bits, and codes of size 128 and 1024. Longer strings and larger databases are possible, but it becomes increasingly harder to handle for the retrieval algorithm (we currently use ZChaff, see below), hindering the ability to collect statistics. The parameter P_e (see Sect. 3.1) was chosen to marginalize the probability of including unwanted valid strings, and k was set to make the creation of NDB agile; however, it could just as easily taken a number of other values yielding different NDB sizes—with $k = 4$ a NDB of size 716 can be readily created for databases of up to 100 elements.

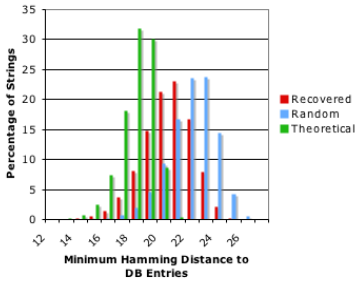
Input: x, c
Output: A code of length c for string x
$o \leftarrow MD5(x)$
while($length(o) < \lceil c/128 \rceil$) *//the length of the MD5 code is 128 bits*
    $o \leftarrow o \cdot MD5(x \cdot o)$
return the c most significant bits of o

Fig. 4. Code Generation Algorithm

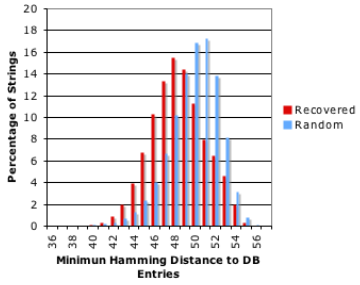
The details of each experiment are shown in the corresponding captions. Each run (an experiment has 10 runs) consists of generating a negative database for the input DB using algorithm 1 and recovering 100,000 strings from the resulting NDB using zChaff [2]—a complete solver for satisfiability formulas. Using zChaff entails transforming NDB into its equivalent boolean formula (for more on the transformations see [9]) as an intermediate step and converting the solution back to a string in our representation. Each run invokes zChaff 10^5 times with different random seeds to obtain a variety of DB' entries. zChaff uses several advanced heuristics and it's therefore not expected to select solutions uniformly at random; nevertheless, the experimental results (see Fig's. 5 and 6) do not diverge significantly from the theoretical analysis. Indeed, part of our security relies on the infeasibility of leveraging the search towards valid solutions.

The experiments report the percentage of recovered strings (divided into text and code) at a given Hamming Distance from DB (when DB has more than one element we report the distance of the recovered string to the closest DB entry) and show that, for proper parameter settings, it is unlikely to recover a string in DB' that is close enough to DB to use as a starting point for an exhaustive search. No experiment yielded any strings in DB and no valid strings where discovered (recall that there is a slight probability of including valid strings in DB' that are not in DB).

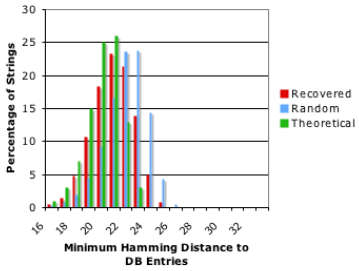
The experiments show that the distance distribution of recovered texts to texts in DB is increasingly similar to the distance distribution of texts guessed at random, and suggest that the likelihood of retrieving a text within a certain distance of the hidden data diminishes as the code grows. Likewise, the experiments also show that the distance of strings recovered to strings concealed increases with their length and, therefore, that the likelihood of retrieving a code (or text) close to DB also decreases. This properties together with the amount of distinct texts and codes that are included in DB' computationally bound the number of strings that can be recovered using NDB before a DB entry can be found using zChaff. Notice that no string can be discarded from the search, given that a large number of text bits are required for a reasonable guess of what its corresponding code will be (the composite MD5 of Fig. 4) and vice versa. Finally, note that our algorithm (Fig. 1) can produce the same



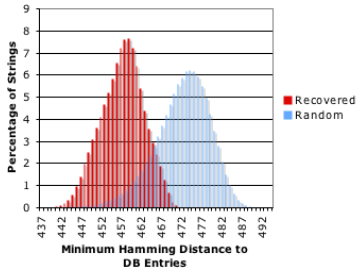
(a) Minimum distance of the recovered texts to the texts in DB (t=64,c=128)



(b) Minimum distance of the recovered codes to the codes in DB (t=64,c=128)

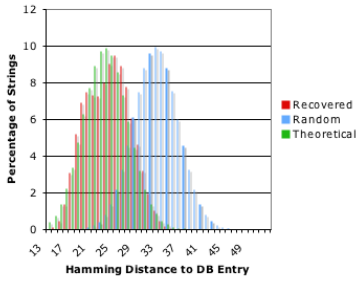


(c) Minimum distance of the recovered texts to the texts in DB (t=64,c=1024)

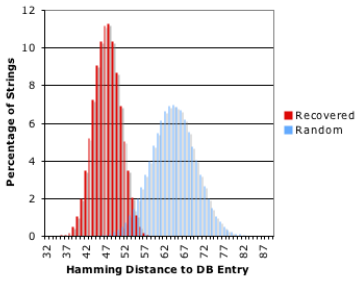


(d) Minimum distance of the recovered codes to the codes in DB (t=64,c=1024)

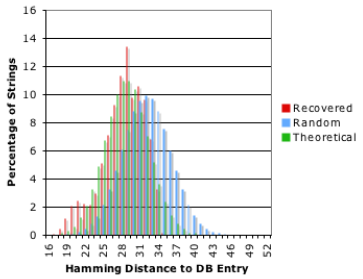
Fig. 5. Each figure displays the results of an experiment involving ten distinct DBs of size 100 with t=64 and k=7; the DBs of figures (a) and (b) have a code size of 128 and those of (c) and (d) a size of 1024. A NDB of size 5768 was created for each DB and 100,000 "positive" strings recovered from the corresponding $DB's$. The percentage of distinct entries recovered from each DB' for each Hamming Distance was computed and averaged over the ten experiments. Each figure also shows the minimum distance to DB of 50,000 randomly generated strings averaged over the ten DBs. Figures (a) and (c) additionally show the value predicted by using eq. 7 (the recovered and predicted values are very close together, the distance of the random strings are the rightmost values in each figure). Algorithm 1 iterated 12800 times in the worst case



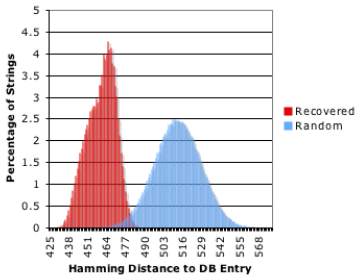
(a) Distance of the recovered texts to the text in DB (t=64, c=128)



(b) Distance of the recovered codes to the code in DB (t=64, c=128)



(c) Distance of the recovered texts to the text in DB (t=64, c=1024)



(d) Distance of the recovered codes to the code in DB (t=64, c=1024)

Fig. 6. Each figure displays the results of an experiment involving ten distinct DBs of size one with t=64 and k=3; the DBs of figures (a) and (b) have a code size of 128 and those of (c) and (d) a size of 1024. A NDB of size 366 was created for each DB and 10^5 "positive" strings recovered from the corresponding $DB's$. The percentage of distinct entries recovered from each DB' for each Hamming Distance was computed and averaged over the ten experiments. Each figure also shows the distance of 50,000 randomly generated strings to DB averaged over the ten DBs and figures (a) and (c) show the value predicted using eq. 7. Algorithm 1 iterated 420 times in the worst case.

output NDB for a large number of different input DBs, since any subset of DB' can possibly cause NDB. This handicaps the ability of analyzing NDB to deduce DB. For appropriate text and code lengths the data is safely hidden by a negative database.

5 Related Work

There are many areas that are of interest and influence to this work. First, is the previous work on negative databases: In [11] the negative database representation is introduced; [8] uses a similar algorithm for generating NDBs and introduces some interesting applications, but relies on NP-hardness for its security. In [12] the issue of attaching a code is proposed, albeit it is used with a

different purpose, and [7] is concerned with generating NDBs efficiently. Both consider ways of creating secure NDBs. In contrast to [7], our proposal does not rely on cryptographic primitives for the security of the system, and its construction allows for the stored data to be manipulated via a special algebra (see [13]). Reference [10] discusses several algorithms for updating a negative database once it has been created.

The technique of winnowing and chaffing presented in [20] and further studied in [16] has many similarities with our proposal as does the method presented in [5] (Danezis et.al [7] provide another related example); primarily the notion of hiding valid strings amid invalid ones. The main distinctions are that our technique does not require the use of any shared secrets, anybody with a text can verify if it is in DB, and that the amount of chaff that can be efficiently included alongside DB is far greater (this is one of the main advantages of negative databases). Neither method requires encryption.

The work in [18] has similar goals as our own. Here, queries to an encrypted database are restricted by type and "mass harvesting" queries are computationally inefficient. Our technique impedes mass harvesting without the use of encryption by virtue of the representation, but only processes simple membership queries efficiently. It allows, however, the use of many relational algebra operations on the data without the need for keys or secrets. Other techniques using different principles for restricting the types of information that can be learned from a database include [1, 17].

On the topic of representation there are several techniques for creating compact depictions of binary sets, of special interest are Bloom Filters [4] and Ordered Binary Decision Diagrams [6]. The primary differences between these and our scheme is the need of negative databases to always obtain a compact depiction of the complement of a set without explicitly calculating it, and the difficulty with which the data can be retrieved.

Information Hiding techniques [3, 19] like water-marking and steganography [15] focus on concealing data items within some stream of information (while, in a matter of speaking, ours hides information in the garbage). Our approach is similar to these techniques in that, given a negative database, it is infeasible to determine whether it represents any valid string (message) at all, i.e., if the corresponding positive database is non-empty; and in that knowledge of the existence of a message does not facilitate its retrieval. However, the referred techniques allow retrieving the message with special knowledge of its whereabouts, while our present proposal does not.

6 Discussion

In this paper we presented a method for hiding data by placing it amidst garbage. The novelty of our scheme is that instead of storing data plus garbage, we store the complement of this set—the negative database. This enables us to compactly keep an arbitrary amount of invalid strings alongside the data. Other work on negative databases rely on NP-hardness principles or on cryptographic primitives

to safeguard the confidentiality of the original data. Our scheme relies on neither, but rather on the amount of garbage and the difficulty of systematically telling it apart from the valuable data. Further, the present scheme surpasses some of its predecessors in keeping the size of the data structure manageable; for instance, a negative database for 100 strings can be maintained with roughly four times as many bits as the positive version ($k = 4$, $P_e = 2^{-(t+1)}$ eq. 2).

Our scheme allows any entity to efficiently verify the presence of a given data item in the structure, while preventing massive data harvesting. The most important differences with methods that offer this same functionality are that negative databases don't use encryption; that they obfuscate the size of the hidden set; that a single dataset has a very large amount of different negative database representations and no easy way to test their equivalence; and that they can be manipulated to express transformations on the hidden set. As an example of the latter consider a list of credit-card numbers (CCN) and balances, and its corresponding NDB; suppose the CCN occupies the 54 most significant bits of each record and the balance the following 20. Appending entry <(0,55)(0,56)(0,57)> to NDB will restrict the list's contents to only those with balance less than 131072 dlls. Further, using the operations described in [13] the NDB can be joined with a negative database of names and CCNs to produce one of names, CCNs, and balances. All without knowledge of the actual contents of the actual lists or their sizes.

In this paper we focused on creating NDBs for text and codes because of the functionality added by the relational algebra operations mentioned above. Possible variations of our work include keeping only the code (loosing said functionality) or having secret codes or code generations mechanisms that allow privileged parties to efficiently limit the search and recover the original data.

The results and analysis presented in this paper strongly suggest that negative database can be used to safely keep data hidden; there is, however, more work to be done. Along with more extensive experimentation, better data structures for representing DB and NDB need be explored (to enable more efficient searches) and the statistical properties of NDBs studied in more detail.

Acknowledgments

The author wishes to thank Elena S. Ackley, Stephanie Forrest, and Joan Feigenbaum for their help and insights; and gratefully acknowledge the support of the PORTIA project (NSF grant 0331548) for partially funding this research.

References

1. R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 439–450. ACM Press, May 2000.
2. Boolean Satisfability Research Group at Princeton. zChaff. http://ee.princeton.edu/~chaff/zchaff.php, 2004.

3. W. Bender, D. Gruhl, N. Morimoto, and A. Lu. Techniques for data hiding. *IBM Systems Journal*, Sept-Dec 1996.
4. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, jul 1970.
5. R. Brinkman, S. Maubach, and w. Jonker. A lucky dip as a secure data store. In *Proceedings of Workshop on Information and System Security*, 2006.
6. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.
7. G. Danezis, G. Diaz, S. Faust, E. Käsper, C. Troncoso C., and B. Preneel. Efficient negative databases from cryptographic hash functions. In Springer LNCS, editor, *Information Security Conference*, volume 4779, pages 423–436, 2007.
8. M. de Mare and R. Wright Secure. Set membership using 3sat. In *Proceedings of the Eighth International Conference on Information and Communication Security (ICICS '06)*, 2006.
9. F. Esponda. *Negative Representations of Information*. PhD thesis, University of New Mexico, 2005.
10. F. Esponda, E. S. Ackley, S. Forrest, and P. Helman. On-line negative databases. In Giuseppe Nicosia, Vincenzo Cutello, Peter J. Bentley, and Jon Timmis, editors, *Proceedings of the 3rd International Conference on Artificial Immune Systems (ICARIS)*, pages 175–188, Catania, Sicily, Italy, Sep 2004. Springer-Verlag.
11. F. Esponda, S. Forrest, and P. Helman. Enhancing privacy through negative representations of data. *Technical report, University of New Mexico*, 2004.
12. F. Esponda, S. Forrest, and P. Helman. Protecting data privacy through hard-to-reverse negative databases. *International Journal of Information Security*, 6(6):403–415, Oct. 2007.
13. F. Esponda, E. Trias, E.S. Ackley, and S. Forrest. A relational algebra for negative databases. Technical Report TR-CS-2007-18, University of New Mexico, 2007.
14. S. Hofmeyr and S. Forrest. Architecture for an artificial immune system. *Evolutionary Computation Journal*, 8(4):443–473, 2000.
15. S. Katzenbeisser and F. A. P. Petitcolas, editors. *Information hiding techniques for steganography and digital watermarking*. Artech House computer security series. Artech House Inc., Norwood, MA, USA, 2000.
16. J. McHugh. Chaffing at the bit: Thoughts on a note by ronald rivest. In *Proceedings of the Workshop on Information Hiding*, pages 395–404, 1999.
17. S. Micali, M. Rabin, and J. Kilian. Zero-knowledge sets. In *Proc. FOCS 2003.*, page 80, 2003.
18. A. Narayanan and V. Shmatikov. Obfuscated databases and group privacy. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 102–111, New York, NY, USA, 2005. ACM.
19. F.A.P. Petitcolas, R.J. Anderson, and M.G. Kuhn. Information hiding-a survey. *Proceedings of the IEEE special issue on protection of multimedia content*, 87(7):1062–1078, 1999.
20. R. L. Rivest. Chaffing and winnowing: Confidentiality without encryption. MIT Lab for Computer Science, http://theory.lcs.mit.edu/~rivest/chaffing.txt, March 1998.
21. R.L. Rivest. The md5 message-digest algorithm. 1992.