Cryptographic Protection of Databases and Software

Joan Feigenbaum[*] Mark Y. Liberman[†] Rebecca N. Wright[‡]

August 15, 1989

Abstract

We describe experimental work on *cryptographic protection* of databases and software. The database in our experiment is a *natural language dictionary* of over 4000 Spanish verbs. Our tentative conclusion is that the overhead cost of computing with encrypted data is fairly small.

1 Introduction

It is often desirable to allow a user to access a database D via some program P while preventing her from obtaining a copy of the entire contents of the database. For example, let D be a table of pairs (l_i, r_i), $1 \leq i \leq n$, where each l_i is the name of a person and $P(l_i, D) = r_i$ is the phone number of l_i. A typical user of D should be able to obtain the number r_i if she knows l_i but should not be able to obtain the names and numbers of the people she does not know.

More formally, given a database D and program P that accesses it, we wish to construct an encrypted database D' and a corresponding program P' with the following properties.

- For any query q, $P(q, D) = P'(q, D')$.
- Given P' and D', it is computationally infeasible to reconstruct D.

This report describes:

- A simple cryptographic solution for protecting table-structured databases.
- Experimental work applying this technique to certain natural-language databases and the automata-based programs that access them.
- Potential alternative approaches and open questions.

[*]AT&T Bell Laboratories, Room 2C473, Murray Hill, NJ 07974 USA, jf@research.att.com.
[†]AT&T Bell Laboratories, Room 2D446, Murray Hill, NJ 07974 USA, myl@research.att.com.
[‡]Yale University, Comp. Sci. Dept., New Haven, CT 06520 USA, wright-rebecca@cs.yale.edu.

Various theoretical problems concerning computation with encrypted data have appeared in the cryptologic literature. We mention two of them and then explain how they differ from the problem that we study here.

Yao [30] posed the problem of which distributed computation problems have *secure multiparty protocols*. That is, when is it possible for equally-powerful parties P_i, $1 \leq i \leq n$, each holding a private input x_i, to cooperate in the computation of $y = f(x_1, \ldots, x_n)$ in such a way that each P_i learns y and no P_i learns anything about x_j, $j \neq i$, except what is implied by y and x_i? Secure multiparty protocols in which all of the parties are limited to polynomial-time computation are discussed in [10, 13]. Those in which all of the parties are computationally unlimited are discussed in [3, 4].

Abadi, Feigenbaum, and Kilian [1] posed the problem of which functions have *instance-hiding schemes*. In their scenario, one party, say P_0, has a private input x, but lacks the computational resources to compute f. Party P_1 has the power to compute f but cannot guarantee the privacy of inputs it receives. For which functions f can P_0 cooperate with P_1 to obtain $f(x)$ without revealing x? Instance-hiding schemes in which P_0 cooperates with multiple powerful parties $P_1, \ldots, P_m$ are discussed in [2].

In this paper, we take an experimental approach to computing with encrypted data, and we set a more modest goal than that of the previous authors. We wish to allow users to learn the answers $P(q, D)$ to specific queries q, but we wish to make it difficult for them to incorporate the entire database D into another program of their own design. We do this by providing only an encrypted database D'. The "information content" of D is certainly present in D', but we can make it difficult to extract by using a good encryption function. Note that access to D' via the program P' *does* allow a user to recover the input-output relation of P by computing $P'(q, D')$ for all possible queries q. This does not mean that we have not gained anything in encrypting D. First, the value of D may be in the way it is structured, not in the "raw data" that it contains; indeed, the raw data of the input-output relation is often publicly available and is, in fact, the starting point for both the database designer and the potential thief. If the user were willing to structure D from the raw data, then he would not need D' at all. Second, the set of all *syntactically valid* queries is usually a proper superset of the set of queries that yield meaningful output, and it may be far too large to generate by exhaustive search.

The natural language software that we seek to protect is a good example of *leaky technology*, as defined by Ouchi [27]. Once developed and sold by one vendor, it can easily be modified and resold by someone who has not borne any of the development costs. Traditional patent and copyright laws may not be applicable, and litigation is a costly and cumberson source of protection in any case. Ouchi suggests joint research and development by large companies all of whom stand to gain from the advancement of a leaky technology. Clearly a technical solution to the problem of leakage would be more satisfying and, in the software industry, where there are many potential vendors most of whom have no research and development efforts, it may be the only possible type of solution.

The rest of this paper is organized as follows. Section 2 describes a method of encrypting table-structured databases such as the phone database described above. Section 3 gives the structure of the natural language databases that we seek to protect and describes our experiment with the Spanish-verb dictionary. Finally, Section 4 states our conclusions and open questions.

2 A Method for Encrypting Tables

Suppose that D consists of a table of pairs (l_i, r_i), $1 \leq i \leq n$, and that P is a simple lookup program. That is, for any l that is equal to l_i, for some $1 \leq i \leq n$, $P(l, D) = r_i$, and, for any other l, $P(l, D) = \text{NIL}$. The l_i's are referred to as *keys* and the r_i's as *values*. We construct the corresponding D' and P' as follows.

Let L be the set of all syntactically valid keys; in particular, $L_{\mathsf{in}} \subseteq L$, where $L_{\mathsf{in}} = \{l_1, \ldots, l_n\}$. Let h be a one-way hash function whose domain is L. Ideally, such a function h would be a polynomial-time computable, one-to-one mapping from L onto the integers 1 through $|L|$ with the property that, given $j \in \{1, \ldots, |L|\}$, it is computationally infeasible to find $h^{-1}(j) \in L$. For a more thorough discussion of one-way hash functions, refer to [24, 25].

Let Enc and Dec be encryption and decryption functions such that, for all x and k, $Dec(Enc(x, k), k) = x$. Then the encrypted database D' is derived from D as follows: for every (l_i, r_i) pair in D, a pair $(h(l_i), Enc(r_i, l_i))$ is inserted into D'. The modified lookup program P' uses the algorithm:

$P'(l, D')$:

```
{
    l' ← h(l)
    r' ← P(l', D')
    if r' = NIL, return(NIL).
    else return(Dec(r',l)).
}
```

This table-encryption scheme is an enhancement of a standard password-storage technique first developed by Needham (see Denning's book [7, Section 3.6] for a thorough discussion). Normally such a scheme is used only when $|L|$ is too large for h to be inverted by brute force and L_{in} is a subset that is infeasible to generate.

3 An Application to Dictionaries

3.1 Why Natural Languages?

Natural-language dictionaries as used by computer programs are good candidates for the kind of protection discussed in this paper. Information about the form and usage of words is not inherently secret – thus the fact that *conozcamos* is the **first person plural present**

subjunctive form of the Spanish verb *conocer* (to know) is public knowledge. Such facts can certainly not be patented, and copyright can only protect the form of their expression, which can easily be changed without altering the content. Nevertheless, a significant investment of skilled labor goes into arranging such facts for computer applications – for instance, it takes a good fraction of a year's work to create a database that correctly relates all possible Spanish orthographic forms with their dictionary head-words and inflectional categories.

There are many other types of information that might be contained in such linguistic databases: the pronunciation of English words, the relation between words and probability distributions over grammatical categories, the relation between Japanese and English terms of art in Electrical Engineering, and so forth. These databases may have hundreds of thousands or even millions of entries, which usually have to be created or checked by human labor. For convenience in exposition, we will use the word "dictionary" to mean natural language databases that require skilled labor and linguistic knowledge to design and assemble from the raw data.

Anyone who creates software that includes such a dictionary must face the fact that its information content is likely to be stolen, if it is not somehow protected. That is, dictionary software is leaky. Our aim in this section is to present a general method for protecting dictionaries, along with an experimental implementation.

3.2 Structuring and Storing the Cleartext Dictionary

Sometimes a dictionary is naturally expressed as a simple key-value table, and in this case, the method sketched in Section 2 can be applied directly. Often, however, the forms to which the dictionary applies arise from the rule-governed combination of parts, and the resulting relation may be large enough that we would rather generate it on-line than store it. Thus regular plural and possessive forms in English would triple the size of the noun list in a pronouncing dictionary, but can be derived in a simple way from the singular forms. For each Spanish verb stem, regular inflection and the attachment of enclitic pronouns can produce almost 100 forms. In some cases, such as the treatment of compound words in German, the full table is not bounded in any non-arbitrary way, and a decomposition in terms of simpler parts is forced.

One simple but useful way to treat such computations is as **transductions**, that is, n-ary word relations definable by the component-wise concatenation of n-tuple labels along paths in a finite directed labeled graph.[1] Define an *n-ary nondeterministic finite automaton* as a 5-tuple

$$\mathcal{A} = (Q, q_1, F, \Sigma, H)$$

where Q is a finite non-empty set of states, q_1 is a designated start state, F is a set of designated final states, Σ is a finite non-empty alphabet, and H is a finite subset of $\{Q \times (\Sigma^*)^n \times Q\}$, where $(\Sigma^*)^n$ is the set of n-tuples of (possibly empty) words over Σ. Thus $\mathcal{A}$ defines a labeled directed graph, whose nodes are elements of Q, and whose edges are elements

[1] See Rabin-Scott [28] and Elgot-Mezei [8].

of H. Each edge is labeled with an n-tuple of (possibly empty) words. The component-wise concatenation of labels along every path that begins in q_1 and ends in an element of F defines a set of n-tuples, $R \subseteq (\Sigma^*)^n$, which is the relation transduced by $\mathcal{A}$.

The information content of most dictionaries (in the sense in which we are using that term) can be comfortably expressed by such a graph.[2] In order to use the dictionary, we need to run a program that searches this graph in order to find all the n-tuples in R meeting some condition, say those for which the "spelling" component of the relation is the string w.

Figure 1 shows a sample graph, labelled by pairs of strings, which expresses the regular aspects of Spanish verb inflection. As a cleartext database D we represent such a graph by an *arcfile* as follows. We assume, without loss of generality, that the start state has a designated name $START$ and that there is a single final state with the designated name $FINAL$. Every other node is the graph is assigned some unique string as an identifier. Then the information content of the graph is given by a set of 4-tuples of strings

```
OriginState DestinationState InString OutString
```

3.3 A Program for Accessing the Cleartext Dictionary

The lookup program P searches the graph represented by such an arcfile for all paths consistent with a given input string; that is, for all paths such that the input string matches the concatenation of *InStrings* along the path. Obviously, the labeling of tuple components as input and output is arbitrary – the same arclist can be used for computations in either direction. In the Spanish verb dictionary described in Section 3.5 below, this means that the same graph can be used to map verbs to morphological descriptions (e.g., *conozcamos* to **first person plural present subjunctive** of *conocer*) and to map morphological descriptions to verbs (e.g., **first person plural present subjunctive** of *conocer* to *conozcamos*); for the first mapping, we search the graph in one direction, and for the second, we search it in the opposite direction.

A recursive algorithm for P running on a given user input w is shown below. It assumes a function $find(D, OriginState, InString)$ that returns the (possibly null) list of $(DestinationState, OutString)$ pairs from all the arcs in D with that OriginState and InString. The notation `string1 - string2`, where string2 is an initial substring of string1, denotes the remaining characters of string1 if string2 is removed from its beginning. The notation `string1 + string2`, where string1 and string2 are arbitrary strings, denotes concatenation. `NULL` denotes the null string. *Process* uses the Boolean variable *accept* to keep track of whether there has been any accepting path.

[2]Many researchers have based dictionary-access programs on non-deterministic finite transducers, although little of this work has been documented (see [15, 16, 17, 20, 21]). Previous work of this type is based on architectures in which the relations defined by multiple automata are intersected or composed, with an implementation that either simulates the multiple automata directly or (where this is possible) first constructs an equivalent single automaton.

$P(w, D)$:

```
Initialize:
    {
        state ← START                  /* current state */
        in ← w                         /* input not yet used up */
        out ← NULL                     /* output seen so far */
        accept ← FALSE
        accept ← process(state, in, out, accept)
        if accept = FALSE, print(NIL)
    }
process(state, in, out, accept):
    {
        if state = FINAL and in = NULL {
            print(out)
            accept ← TRUE
            return(accept)
        }
        /* otherwise continue */
        for i ← 0 to length(in) {
            s ← initial substring of in of length i
            for (dstate, outstr) in find(D, state, s) {
                accept ← process(dstate, in-s, out+outstr, accept)
            }
        }
    return(accept)
    }
```

3.4 The Encrypted Dictionary and Modified Lookup Program

In order to create the corresponding encrypted database D', the arcfile can be treated as a table D of pairs (l_i, r_i), $1 \leq i \leq n$ by taking, for each arc, the concatenation of the OriginState and InString as l_i, and the concatenation of the DestinationState and OutString as r_i. Additional complexity can be given to D' by renaming the states to long random strings before encryption.

The program P' will then search the graph encrypted as D'. The algorithm is similar to P, with additional steps to deal with the encryption. It assumes a program *xfind(D', hashvalue)* which returns the (possibly null) list of encrypted r values corresponding to arcs with that specified hashvalue in D', and a program *separate(string)*, which determines whether its input string is a valid DestinationState-OutString concatenation, returns the separated DestinationState and OutString if valid, and returns FALSE if its input is not valid.[3] As before, `string1 - string2` denotes the remaining characters of string1 after removing string2 from the beginning, `string1 + string2` denotes concatenation, and `NULL` denotes the null string.

$P'(w, D')$:

```
    Initialize:
        {
            state ← START                /* current state */
            in ← w                       /* input not yet used up */
            out ← NULL                   /* output seen so far */
            accept ← FALSE
            accept ← process(state, in, out, accept)
            if accept = FALSE, print(NIL)
        }
    process(state, in, out, accept):
        {
            if state = FINAL and in = NULL {
                print(out)
                accept ← TRUE
                exit
            }
            /* otherwise continue */
            for i ← 0 to length(in) {
                s ← initial substring of in of length i
                for enctext in xfind(DB, hash(state+s)) {
                    dectext ← Dec(enctext, state+s)
                    if separate(dectext) = NIL, return(accept)
                    /* otherwise continue */
                    (dstate, outstr) ← separate(dectext)
                    accept ← process(dstate, in-s, out+outstr, accept)
                }
            }
            return(accept)
        }
```

As noted in Section 1, the complete input/output relation of a program accessing the database does not always provide enough information to reconstruct the database. In the case of an automaton simulator, for example, reconstructing the sequential transducer would require an exhaustive search of the space of possible origin-instring pairs.

[3] If we could guarantee the absence of hash collisions, this check on the validity of the encrypted form of r given l would not be necessary.

3.5 Our Implementation and the Spanish-verb Analyzer

In the implementation described in this section, the hash function $hash(x)$ was a modified version of the hash function used by the UNIX spelling checker *spell* [23]. For the encryption-decryption pair $Enc(x, k)$ and $Dec(y, k)$, a modification of Don Mitchell's DES implementation was used.

The programs are called *dictionary*, *datacrypt*, and *lookup*. As a test case, the spanish regular verb analyzer shown in Figure 1 was used.[4] It contains 4417 regular Spanish verbs, and recognizes 48 of the inflected forms of each verb: the infinitive, the present and past participles, the imperatives, the indicative present, imperfect, preterite, future and conditional, and the subjunctive present and imperfect of each verb for all applicable combinations of person and number. The size in bytes of the cleartext arcfile and the encrypted files are as follows:

cleartext arcfile	94704
encrypted arcfile	108682
encrypted reverse arcfile	95057

Thus the space overhead for encryption was not very great. Of course, the cleartext arcfile is quite redundant, and could be represented much more compactly. Significant compression (e.g. by a factor of 3 to 4) could also easily be achieved in the encrypted version, although some techniques (such as those that use shared structure) would not work, and it would therefore be hard to match the ultimate level of compression achievable in the clear. Still, the size of dictionary databases encrypted by this method would be acceptable for many applications.

We implemented both P and P', as described above, in C, aiming more at clarity and simplicity than at optimal performance. Our implementation of the P algorithm runs about twice as fast as our implementation of the P' algorithm, which nevertheless is able to process more than 100 words per second on a 10-MIP machine. The P' implementation spends a good deal of its time on a rather inefficient conversion of character strings into DES keys, and caching of previously-obtained results would give a big performance boost given the usual Zipf's-law distribution of input forms. Even without any tuning, however, the performance is acceptable for many applications.

We should note that there are some efficient implementation techniques that would not be usable with this approach to database protection. An example is the use of a letter trie[5] associated with each state to encode the set of *instrings* that can be matched at that state, thus avoiding the need to test all possible substrings.

[4] Following this experiment, the same framework was applied to a complete implementation of the large Collins Spanish/English dictionary [6]. This database covers 55,000 base forms and the full set of their inflections and regular derivations.

[5] Refer to Knuth [19] for a detailed description of tries and their use in data retrieval.

4 Conclusion and Open Questions

Our tentative conclusion is that it is feasible, both in terms of storage-space overhead and in terms of running-time overhead, to compute with encrypted versions of natural-language databases encoded as automata. Two types of open questions remain and invite further work.

First, there are open questions about our implementation. How close to one-way is the *spell* hash function? Our software can be modified easily to use any hash function that is fast enough to render the resulting *lookup* program practical. In our implementation of *lookup*, the following "false" matches are detected: $h(OriginState + InString)$, for some invalid available substring, matches the key of an entry in the encrypted automaton, but the decrypted value of the same entry is not a valid $h(\text{destination} - \text{state} + \text{outstring})$ pair. The applicable notion of "validity" need not require the algorithm to know the set of possible state names, but could (for instance) be determined by a redundancy check character inserted into the decrypted string after the character that delimits the boundary between *destination-state* and *outstring*. It seems unlikely, but is obviously not impossible, that an undetectable false match could occur, i.e., an invalid *instring* choice could cause the automaton to "hit" a valid $h(\text{destination} - \text{state} + \text{outstring})$ pair and to continue through an erroneous path of the simulation. Is this really unlikely enough in practice to be ignored? Note that for output actually to be produced in such a case, the nonsensical instring would have to produce false matches along an entire path from start state to final state.

Next, we mention two open questions about the relationship of this work to more general questions in software protection and software engineering.

If the hashing, encryption, and decryption algorithms used in our scheme are good enough, then a computationally-bounded adversary cannot compute D given P' and D'. However, a trace of the execution of P' on a particular input would allow an adversary to learn one path through the automaton D. To recover the entire automaton, the adversary would have to generate inputs which "cover" all of D. The following observation is due to R. J. Lipton: the consensus in the program-testing community is that "coverage," i.e., generating a set of test inputs that cause every statement in a program to be executed, is very hard to achieve. Is it possible to formalize the apparent "reduction" from coverage to decryption? That is, can we say precisely that, to recover D, or even a significant portion of it, the adversary would have to generate a covering set of inputs, which appears to be hard in practice?

Finally, we draw the reader's attention to the Goldreich software protection model [12]; see also the recent work of Ostrovsky (to appear in [26]; abstract in these proceedings [9]). We do not review the details of the model here, but merely point out that our database protection goal is a special case of the goal of the software vendor in the Goldreich model. In [12], the end user feeds an encrypted program π' and a cleartext input x to a computer that contains a nonstandard piece of hardware, called a "cryptographic cpu." The software vendor wants the user not to be able to infer *anything* significant about the original program π from the memory-access pattern of the computer running π' on input x. In our case, the original program is the pair P, D, and the input is the query q. When we feed the encrypted pair P', D' to the computer, we do not care whether the user learns P. For the automaton case that we have already implemented, P is just a certain kind of graph search program, which the user could presumably construct on his own, and in general, P is some sort of database lookup program; all of the valuable information is in the encrypted data D'. Are there inexpensive versions of the solutions of Goldreich or Ostrovsky that work for the type of programs that we are interested in but may not work in general?

5 Acknowledgements

It is a pleasure to acknowledge the help of Doug Blewett and Don Mitchell.

References

[1] M. Abadi, J. Feigenbaum, and J. Kilian. On Hiding Information from an Oracle, *J. Comput. and System Sci.* 39(1):21–50, 1989.

[2] D. Beaver and J. Feigenbaum. Hiding Instances in Multioracle Queries, *Proceedings of the 7th STACS* (1990), Springer Verlag, 37–48.

[3] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation, *Proceedings of the 20th STOC* (1988), ACM, 1–10.

[4] D. Chaum, C. Crépeau, and I. Damgaard. Multiparty Unconditionally Secure Protocols, *Proceedings of the 20th STOC* (1988), ACM, 11–19.

[5] N. Chomsky. Context-free Grammars and Pushdown Storage, MIT Research Lab of Electronics Quarterly Progress Report, 1962.

[6] *Collins Spanish Dictionary: Spanish-English.* Collins Publishers, Glasgow, 1989.

[7] D. Denning. *Cryptography and Data Security*, Addison-Wesley, Reading, 1982.

[8] C. C. Elgot and J.E. Mezei. On Relations Defined by Generalized Finite Automata, *IBM Journal Res.* 9:47-68, 1965.

[9] J. Feigenbaum and M. Merritt. Workshop Summary, these proceedings.

[10] Z. Galil, S. Haber, and M. Yung. Cryptographic Computation: Secure fault-tolerant protocols and the public-key model, *Proceedings of the 7th CRYPTO* (1987), Springer Verlag, 135–155.

[11] S. Ginsburg. *An Introduction to Mathematical Machine Theory*, Addison-Wesley, Reading, 1966.

[12] O. Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs, *Proceedings of the 19th STOC* (1987), ACM, 182–194.

[13] O. Goldreich, S. Micali, and A. Wigderson. How to Play ANY Mental Game, *Proceedings of the 19th STOC* (1987), ACM, 218–229.

[14] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, 1979.

[15] M. Kay. When Meta-rules are not Meta-rules, in *Automatic Natural Language Processing*, Spark-Jones and Wilks (eds.). University of Essex, Cognitive Studies Center (CSM-10), 1982.

[16] L. Kartunnen. KIMMO: A general morphological processor, *Texas Linguistic Forum* 22:165–186, 1983.

[17] L. Kartunnen, K. Koskenniemi, and R. Kaplan. A Compiler for Two-level Phonological Rules, manuscript, Xerox Palo Alto Research Center, 1987.

[18] R. Khan. A two-level morphological analysis of Roumanian, *Texas Linguistic Forum* 22:253–270, 1983.

[19] D. Knuth. *The Art of Computer Programming – Vol. 3: Searching and Sorting*, Addison-Wesley, Reading, 1973.

[20] K. Koskenniemi. Two-level morphology: A General Computational Model for Word-Form Recognition and Production, University of Helsinki, Dept. of General Linguistics, Publications No. 11, 1983.

[21] K. Koskenniemi and K. W. Church. Complexity, Two-level Morphology, and Finnish, *Proceedings of the 12th International Conference on Computational Linguistics*, Budapest, Hungary, 1988.

[22] S. Lun. A two-level morphological analysis of French, *Texas Linguistic Forum* 22:271-277, 1983.

[23] M. D. McIlroy. Development of a Spelling List, *IEEE Transactions on Communications* COM-30(1):91–99, 1982.

[24] R. Merkle. One-Way Hash Functions and DES, *Proceedings of the 9th CRYPTO* (1989), Springer-Verlag, to appear.

[25] M. Naor and M. Yung. Universal One-Way Hash Functions and their Cryptographic Applications, *Proceedings of the 21st STOC* (1989), ACM, 33–43.

[26] R. Ostrovsky. Efficient Computation on Oblivious RAMs, *Proceedings of the 22nd STOC* (1990), ACM, to appear.

[27] W. G. Ouchi. The New Joint R&D, *Proceedings of the IEEE*, 77:1318–1326, 1989.

[28] M. O. Rabin and D. Scott. Finite Automata and their Decision Problems, *IBM J. Res.* 3:114-125, 1959.

[29] M. P. Schützenberger. A Remark on Finite Transducers, *Information and Control* 4:185-196, 1961.

[30] A. C. Yao. Protocols for Secure Computations, *Proceedings of the 23rd FOCS* (1982), IEEE, 160–164.