HOW TO ENCRYPT/USR/DICT/WORDS IN ABOUT A SECOND

Peter Su
Matt Bishop

Technical Report PCS-TR92-182

How to Encrypt /usr/dict/words in About a Second

Peter Su
Matt Bishop
Dartmouth College

Abstract

We present an implementation of the Data Encryption Standard on the Connection Machine architecture. The DES encryption algorithm is ideally suited to the Connection Machine because it consists of bit serial operations, and thousands of encryptions can be done in parallel, independently of one another. Thus, our code encrypts passwords about ten times faster than the fastest competition that we know about.

In addition, the nature of the Connection Machine's architecture is such that some of the optimizations that make DES run much faster on conventional architectures have no effect on the performance of the Connection Machine. Our comparison of a simple implementation along with one that uses many "optimizations" illustrates this fact.

1. How the DES Works

The Data Encryption Standard was designated as a federal standard for protecting unclassified but sensitive information [1]. It uses an iterated series of expansions, permutations, and substitutions to transform one block of 64 bits into another of 64 bits. Although the cryptographic key is usually given as 64 bits, in fact only 56 bits are actually used.

From the 64 bit key, a set of 16 distinct *round keys* are generated using a sequence of permutations and shifts. Then the bits of the input block are permuted using the IP permutation (defined in [1]), and the result is split into two halves. The right half is run through a function *f*, which expands it to 48 bits using a table called the E table, xors it with the appropriate round key, and then replaces each set of 6 bits with a set of 4 bits as indicated by a series of substitutions known as S-boxes. The result is then permuted using the P table. This result is xor'ed with the left half, and the new left half and the original right half are exchanged. This is repeated 15 more times, except that no exchange occurs in the last round. Instead, the left and right halves are concatenated and run through the inverse of the IP permutation. The resulting 64 bits are the output block (see figure 1).

The UNIX® operating system uses the DES as the basis for a one-way hashing function designed to protect passwords. The password is never stored on the system; instead, its hash is. To authenticate the user, the system recomputes the hash using the password provided by the user; if that hash matches the stored hash, the authentication succeeds.

The hash function consists of taking the message of all zero bits and encrypting it using the password as the key and iterating the DES 25 times; as the DES has not yet been broken using a known plaintext attack, the password is safe even though both the plaintext (the zero block) and the ciphertext (the stored hash) are available. However, this scheme is vulnerable to a second type of attack, called a *dictionary attack.* This attack requires guessing a potential password, hashing it, and comparing the results to the stored hash. If the computed hash and the stored hash match, the password has been found. If not, the procedure is repeated. This attack is very effective because people tend to pick passwords from a small set of characters [8,7]

The effectiveness of a dictionary attack is dependent on the speed of a particular implementation of the DES algorithm. The faster the implementation, the more effective the attack. The encryption is clearly a bit-oriented transformation. Because accessing individual bits is so expensive, various techniques have been used to speed up software implementations of the Data Encryption Standard [4,2], usually by grouping bits into (small) blocks of bits and precomputing permutations for those blocks.

The speed of a dictionary attack can also be increased by precomputing the set of hashes corresponding to a set of possible passwords; then, all that needs to be done is to collect the set of hashes for a system, and compare. To prevent this latter speedup, the UNIX password hashing algorithm perturbs one part of the DES algorithm. Specifically, when a user selects a password, a number (called the *salt*) between 0 and 2^{12}-1 (inclusive) is generated, and based on that salt the E table is permuted. The salt is stored with the hash, so the authentication procedures can use the same salt to generate the hash. Now, rather than being able to precompute one set of hashes and to compare it to hashes from any system, an attacker would have to precompute $2^{12} = 4096$ distinct sets. This requires a large amount of space, and until very recently has been considered impractical.

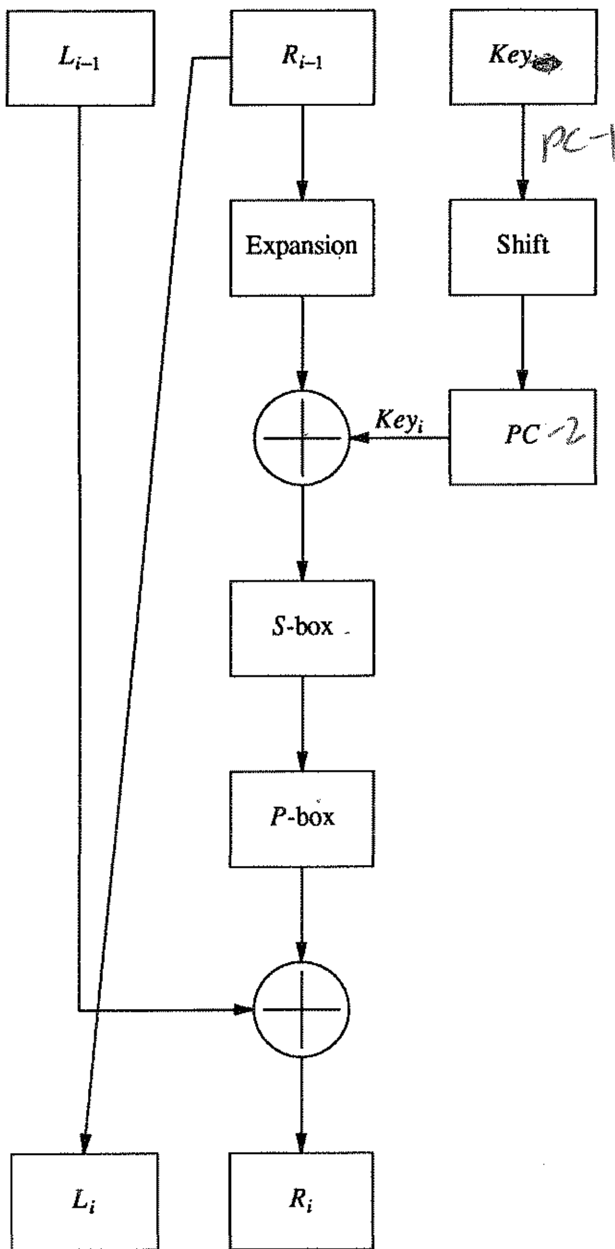

Figure 1: One cycle of the DES. Note that each of the keys Key_i for $0 \leq i \leq 16$ are normally pre-computed.

Because of the structure of the DES, certain additional speedups are available for the UNIX password hashing function. For example, because the initial message is 0, the first application of the permutation IP may be ignored; similarly, as the last step of the DES is to apply the inverse of the initial permutation, both can be omitted from all but the last iteration of the DES. More discussion can be found in [2, 6].

The work cited above concentrates on improving the performance of the DES algorithm on conventional architectures (i.e. Cray, Vax, Sun, etc). However, in recent years many commercial architectures based on bit-serial ALUs have become available. Typically, such a machine would consist of thousands of processors, each of which is bit-serial Since the DES algorithm is also bit oriented, it is natural to examine the performance of the algorithm on these architectures. Therefore, we set out to see how well a Connection Machine would help the average password cracker.

2. The Connection Machine

The Connection Machine is a massively parallel fine grain SIMD architecture employing up to sixty four thousand simple processors, each with 64K bits of memory. Each processor in the Connection Machine has a one bit ALU, and all the processors are connected in a communications network that allows any processor to communicate with any other in the machine. In addition, every 32 processors share a floating point accelerator that can perform single precision calculations on behalf of the processors sharing it. The Connection Machine operates through a front end computer (in our case, a Sun-4) which sends instructions to it through a special interface. A microsequencer decodes the instructions, and then broadcasts the resulting instruction stream to the processors for execution [5].

One can program the Connection Machine at many levels, but the most widely used language is probably PARIS [3]. PARIS stands for "PARallel InStruction" set. The PARIS language is implemented as a library of subroutines which supplements a conventional language such as LISP or C. For our implementation work, we have been using LISP/PARIS.

PARIS allows the programmer to work with an abstract machine with a fixed number of "virtual processors." The system maps these virtual processors, or VPs onto the physical machine. The ratio of virtual processors to physical processors is called the VP ratio. The abstract machine is called a "virtual processor set," or "VP set." One can work with multiple VP sets at once, so the programming model is relatively high level, and flexible.

Within a VP set, a programmer allocated parallel data structures using blocks of memory called *fields*. One copy of each field is allocated on every VP, so multiple copies may be allocated on each physical processor.

PARIS instructions resemble a high level assembly language, and specify operations to be performed by every processor in a VP set. There are instructions for arithmetic and logical operations, memory management, communication between processors, and communication between the Connection Machine and the front end computer. Communication between processors is supported by a hypercube network, and some sophisticated routing

hardware and microcode. PARIS allows the programmer to specify either global communication operations, or specialized grid like communication patterns. In addition, there are mechanisms in the instruction set for conditional execution, and local indirect addressing.

3. Implementing DES in Parallel

Our approach to running DES in parallel is very simple. We simply have each bit serial processor in the CM process a separate key and text block. Because the DES algorithm is bit serial, we can use a straightforward representation of the message and keys. Most of the bit transformations that the DES algorithm uses are independent of the key and the message. Therefore, these transformations can be stored in the front end and broadcast as necessary. We will describe our algorithm using a parallel pseudocode. In the code, parallel data structures are denoted in upper case and scalar variables are lower case.

3.1. Computing the Key Schedule

The key schedule is computed by first permuting the bits of the key, and then iteratively shifting and permuting the result to generate each key. The operations performed are independent of the key and the message, so in parallel the algorithm is pure SIMD.

```
forall processors do
 for (i = 0; i < 28; i++)
  {
   C[i] = KEY[pc1_c[i] - 1];
   D[i] = KEY[pc1_d[i] - 1];
  }
 for (i = 0; i < 16; i++)
  {

   for (k = 0; k < shifts[i]; k++)
      {
       shift(C,1);
       shift(D,1);
      }
   for (j = 0; j < 24; j++)
      {
       KS[i][j] = C[pc2_c[j] - 1];
       KS[i][j+24] = C[pc2_d[j] - 1];
      }
  }
```

Figure 2: Code for the first part of the DES

Since each processor accesses the permutation tables `pc1_c`, `pc1_d`, `pc2_c` and `pc2_d` in the same way, we store these in the front end and broadcast the values as they are needed.

3.2. Encrypting the Message

Encrypting the message is again just a matter of performing bit permutations in each processor. The only tricky part of the process is applying the final selection function. This is the only permutation in the DES algorithm that is accessed in a data dependent way. In order to implement this step, we take advantage of the indirect addressing mechanism in PARIS.

Recall that each group of 32 processors in the CM shares one floating point unit. In addition, there is a hardware interface between the bit serial processors and the FPU called a *transposer*. Each group of 32 bit serial processors, FPU and transposer is called a *Sprint* node. PARIS contains microcode subroutines which allow the user to store an array in the Sprint node and have all 32 bit serial processors share its contents. In addition, each processor can access the array independently of all the others.

Using this mechanism, we store one copy of the selection permutation at each Sprint node and use indirect addressing to fetch data from it. Aside from this small complication, the rest of the encryption algorithm is easily coded in PARIS. As before, we store all the other permutation tables in the front end and simply execute the DES algorithm in a bit-serial fashion.

3.3. Performance

To test the performance of our parallel algorithm, we implemented a version of the UNIX password hashing function that uses our parallel code rather than the normal encryption code. Recall that the UNIX password hashing scheme encrypts a password by running the DES algorithm 25 times. Figure 3 shows the performance of our simple algorithm. The x-axis records the VP ratio, and the y-axis records the number of encryptions per second that the algorithm achieved on an 8192 processor Connection Machine. We tested the code on 64 bit keys, and used the Connection Machine timing system to record the run times.

Notice that the algorithm becomes more efficient at high VP ratios. This reflects the fact that instruction issue has a high overhead on the CM, and at high VP ratios this cost is amortized over the number of VPs that each physical processor must emulate. The performance curve peaks at about 15K passwords per second at a VP ratio of 32 (256K key total). Because the algorithm will scale perfectly with machine size, going to a full 64K CM will provide encryption rates of about 120K passwords per second on two million keys. Thus, on a full Connection Machine, we could check 100 encrypted passwords against a two million word dictionary in about half an hour. On a smaller scale, it would take this code about a second to encrypt `/usr/dict/words` on a 32K CM.

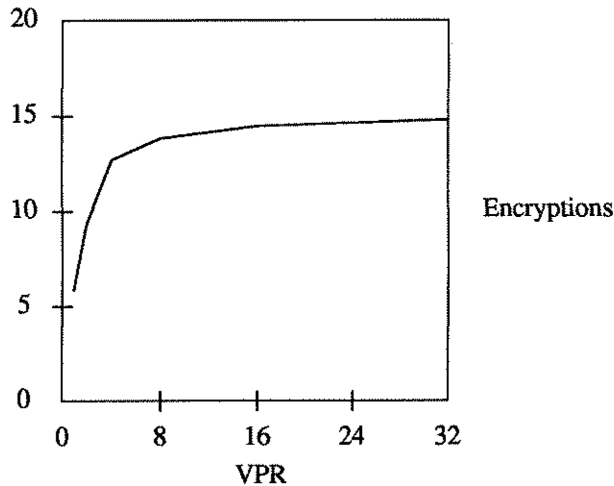

Figure 3: Performance of the simple algorithm.

4. A "Faster" Implementation

In a recent paper [2], Bishop explains how to speed up the DES algorithm on conventional architectures by precomputing many of the permutations, and working with blocks of bits rather than bit serially. On conventional machines, these techniques sped up the algorithm by as much as twenty times. On the Connection Machine, these techniques should only have a minor effect, since the architecture is bit serial. To test this hypothesis, we implemented Bishop's method on the CM.

The main problem with the new implementation is that processors must perform many memory fetches that are data dependent. Thus, in our implementation, we store the precomputed permutation tables in shared arrays and use indirect addressing to do the fetches. The rest of Bishop's method is straightforward to implement, we simply translated his C code into PARIS by hand line by line. This was tedious work, but not difficult.

We measured the performance of Bishop's algorithm the same way we measured the original. However, we could only run the algorithm at a VP ratio of 2 because of the space overhead that the large tables introduced. At VP ratio 2 on an 8K CM, the new code encrypted 16K passwords per second. Thus, the new code has an advantage at low VP ratios, but is not better than the old code running at high VP ratios.

This behavior is consistent with the structure of the CM architecture. By working with blocks of bits rather than single bit operations, Bishop's fast DES code is effectively reducing the effect of the various kinds of overhead that we mentioned in the previous section. This keeps the CM processors busier at low VP ratios and improves the performance of the algorithm.

However, as we said before, PARIS provides us with a way to achieve the same effect with no work at all. We simply run our old code at a higher VP ratio, and our performance increases the same way. Since our original implementation was much simpler than Bishop's method, and dictionaries tend to be large, we feel that the straightforward algorithm is the one to use.

5. Conclusions

If your password is in the dictionary, fix it! It would not take long for an enterprising cracker, with a large government grant, to use a CM to crack every password in the country that is in `/usr/dict/words`.

Our simple experiment with massive parallelism illustrates, to an even greater extent than earlier papers [2], that one cannot depend on security through computational complexity. Each processor on the Connection Machine takes a long time to perform one encryption. Unfortunately, 64K of them encrypting at once is a formidable army of password crackers.

Our code also illustrates the fact that the designers of the DES could not have developed a better algorithm for the Connection Machine to execute. The DES algorithm and the Connection Machine architecture are a match made in heaven. As long as large arrays of bit serial processors exist (Connection Machine, MPP, Blitzen, MasPar), no UNIX password is safe.

6. Acknowledgements

None of this would have been possible without the use of the Connection Machine Network Service, made available to us by Thinking Machines Corporation through the Connection Machine Network Server Pilot Facility, supported under terms of DARPA contract number DACA76-88-C-0012. In addition, the code never would have worked without the help of the technical staff at Thinking Machines, who handled our never ending stream of problems and questions.

References

1. "Data Encryption Standard," Federal Information Processing Standards, 46, National Bureau of Standards (Jan. 1977).
2. M. Bishop, "An Application of a Fast Data Encryption Standard Implementation," *Computing Systems Journal*, 1, 3, pp. 221-254 (1988).
3. T. M. Corporation, "Paris Reference Manual," Tech. Report (1989).
4. M. Davio, Y. Desmedt, M. Fosseprea, R. Govaerts, J. Hulsbosch, P. Neutjens, P. Piret, J.-J. Quisquater, J. Vandewalle, and P. Wouters, "Analytical Characteristics of the DES" in *Advances in Cryptology: Proceedings of Crypto 84*, ed. G. Blakely and D. Chaum, Lecture Notes in Computer Science, 186,

Springer-Verlag, New York, NY (Aug. 1984).

5. D. Douglas, B. Kahle, and A. Vasilevsky, "The Architecture of the CM-2 Data Processor," Tech. Report, Thinking Machines Corporation (1988).
6. D. Feldmeier, "A High-Speed Crypt Implementation," *Advances in Cryptology: Proceedings of Crypto 90* (to appear).
7. D. Klein, "Foiling the Cracker: A Survey of, and Improvements to, Password Security," *Proceedings of the UNIX Security Workshop II*, pp. 5-14 (Aug. 1990).
8. R. Morris and K. Thompson, "Password Security: A Case History," *CACM*, 22, 11, pp. 594-597 (Nov. 1979).