UNIX Password Security - Ten Years Later*

David C. Feldmeier and Philip R. Karn
Bellcore
445 South Street
Morristown, NJ 07960

Abstract

Passwords in the UNIX operating system are encrypted with the *crypt* algorithm and kept in the publicly-readable file /etc/passwd. This paper examines the vulnerability of UNIX to attacks on its password system. Over the past 10 years, improvements in hardware and software have increased the crypts/second/dollar ratio by five orders of magnitude. We reexamine the UNIX password system in light of these advances and point out possible solutions to the problem of easily found passwords. The paper discusses how the authors built some high-speed tools for password cracking and what elements were necessary for their success. These elements are examined to determine if any of them can be removed from the hands of a possible system infiltrator, and thus increase the security of the system. We conclude that the single most important step that can be taken to improve password security is to increase password entropy.

1 Introduction

Ten years ago, Robert Morris and Ken Thompson wrote the standard paper on UNIX password security [9]. It described a new one-way function to encrypt UNIX passwords for storage in the publicly-readable file /etc/passwd. This *crypt* function, based on the NBS Data Encryption Standard (DES) algorithm, remains the standard in almost every version of UNIX.

Crypt uses the resistance of DES to known plain text attack to make it computationally infeasible to determine the original password that produced a given encrypted password by exhaustive search. The only publicly-known technique that may reveal certain passwords is password guessing: passing large word lists through the crypt function to see if any match the encrypted password entries in an /etc/passwd file. Our experience is that this type of attack is successful unless explicit steps have been taken to thwart it. Generally we find over 30% of the passwords on previously unsecured systems.

Recent well-publicized intrusions into UNIX systems prompted another look at the security of the UNIX password algorithm. In certain cases, intruders are using password-guessing attacks much like those described by Morris and Thompson. One such attack was contained in the ARPA Internet Worm of November 1988[12].

Experiments by the authors demonstrate that the rapid improvements in computer price/performance ratios over the past decade call into question the adequacy of the present UNIX password algorithm. By careful optimization and the liberal use of space/time trade-offs, one of us (Feldmeier) has developed a version of the present standard UNIX crypt

*The title refers to the paper by Morris and Thompson printed in Communications of the ACM in 1979[9] function that executes in 0.92 ms on the latest generation of RISC workstations. The old Version 6 UNIX crypt function (based on the M-209 rotor cipher) executed in 1.25 ms on the PDP-11/70s that were current in the late 1970s when the crypt algorithm was changed. It is interesting to note that the main reason given by Morris and Thompson for abandoning the Version 6 algorithm was that it executed too quickly.

This paper discusses how passwords in the UNIX operating system can be found using high-speed versions of the UNIX crypt algorithm and pre-encrypting large dictionaries. Given that such tools are available to crack passwords, the elements necessary to allow such cracking are examined to determine how some of these necessary elements can be eliminated to improve security.

2 Fast Crypt Implementations

The crypt implementation that is included with UNIX distributions (such as BSD 4.2) is not optimized for speed because it already allows logins in a reasonable amount of time. Several techniques can be used to speed up an implementation. One technique is to alter the crypt algorithm so that it is easier to compute but still produces the same results. Another technique is to take advantage of the architectural features of the computer that runs the algorithm. Space-time tradeoffs are used to minimize the number of table lookups at the expense of table size and carefully designed data structures minimize the manipulation of individual bits.

The result of applying these methods to increase the performance of the crypt implementation leads to a 102.9 times speedup over the crypt implementation in 4.2 Berkeley UNIX on a Sun 3/50 and a top speed of 1092.8 crypts per second on a Sun SPARCStation. A more complete description of these techniques may be found in the appendix.

Using the speeds of several fast crypt implementations[1] and the prices of several computers (adjusted for inflation to 1989 dollars) produces the graph in figure 1 that shows the increase in crypts/second/dollar over the last 15 years. The graph shows both increases in hardware speed (□) and the best combinations of hardware and software speeds ($\times$). At the left side of the graph is the speed of the Version 6 crypt ($\bullet$). Crypts/second/dollar is the correct metric because password cracking is an easily segmented problem. The speedup is nearly linear with the number of machines, so the best performance overall is obtained by using machines with the best price/performance ratio. The computers shown are the DEC PDP 11/70 (1975), the DEC VAX 11/780 (1978), the Sun 3/50 (1986), the Sun 4/280 (1987), the DEC 3100 (1989) and the Sun SPARCStation (1989).

Table 1 shows how much CPU time on the DEC 3100 is required for exhaustive search of various password spaces. Note that these numbers are for a single DEC 3100, but exhaustive searches are easy to parallelize and many workstations could be used at night and on weekends when they are otherwise idle. Given 20 machines and the numbers above, it is probably reasonable to do an exhaustive search of passwords of length 7-8 lower-case letters, 7 lower-case letters and numbers, 6 alpha-numeric characters, 5-6 printable character or 5 ASCII characters. The moral is keep your passwords 8 characters long or use lots of unusual characters, but in no circumstance use less than 6 characters. Of course, if the crypt/second/dollar ratio increases by another five orders of magnitude in the next decade, only eight-character passwords that utilize the entire ASCII character set will be immune from brute-force cracking!

[1] See table 6 in the appendix.

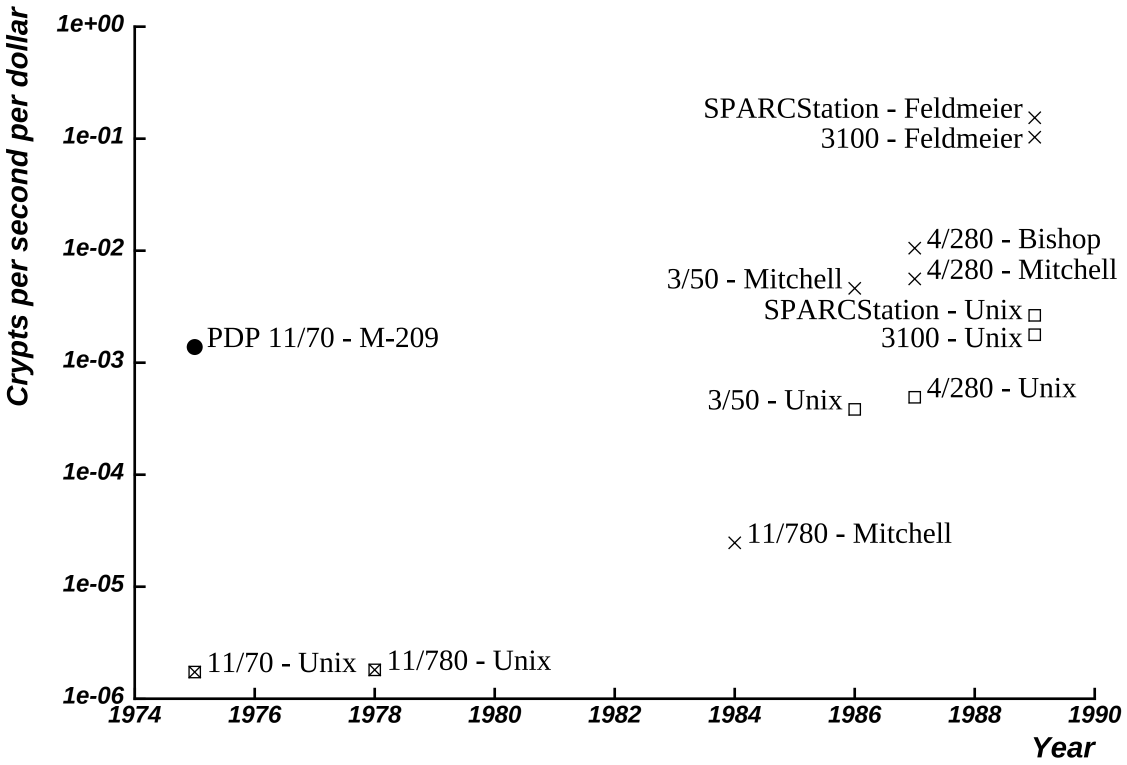

Figure 1: Crypts/Second/Dollar versus Time

n	26 lower-case letters	36 lower-case letters & digits	62 alpha-numeric characters	95 printable characters	128 ASCII characters
4	419 sec	25.7 min	226 min	20.8 hours	68.4 hours
5	182 min	15.4 hours	234 hours	82.2 days	365 days
6	78.8 hours	23.1 days	603 days	21.4 years	128 years
7	85.3 days	832 days	102 years	2.03 kyears	16.4 kyears
8	6.07 years	82.1 years	6.35 kyears	193 kyears	2.03 myears

Table 1: Brute Force Cracking Times of Speed-Crypt on the DEC 3100

Exhaustive search in hardware is possible[5]. One of the aims of including the salt in the crypt algorithm was to remove the threat of using DES hardware to find passwords. However, it would not be too expensive to build VLSI chips that compute the crypt function and run 1000 times faster than these software implementations, not to mention the possible gain due to parallelism and pipelining.

The ultimate size of the key space allowed by the UNIX crypt program is very large: 2^{56} or about 7.2×10^{16} possible keys. Even with only 95 printable characters on a keyboard, there are still 95^8 or about 6.6×10^{15} possible keys. This is large enough to resist brute-force attacks in software, yet most of the passwords selected by users are in a very small part of this total space. It is impossible to use exhaustive search over the large space, but it is possible over the smaller space. What is needed is a list of words that have a high probability of being chosen as a password. Such a list can be derived from dictionaries, telephone directories, etc.

3 Precomputed Encrypted Dictionaries

A fast way of cracking large batches of passwords on a routine basis is to first encrypt a list of likely passwords and then compare each new batch of encrypted passwords against this pre-encrypted list.

Salting was specifically designed to hinder this approach. Because the specific salt values are not known in advance, the pre-encrypted dictionary must encrypt each trial password with all possible salts, increasing storage requirements considerably. However, bulk storage is now far cheaper than it was ten years ago. In 1979, the densest form of bulk storage on the market was 6250 bpi magnetic tape and the corresponding tape drives were expensive. The consumer video cassette recorder boom of the 1980s has produced a spinoff in the form of inexpensive digital cassette drives that can store about 2 gigabytes on a standard 8mm video cassette. These cassettes are about the same size as audio cassettes, yet they can hold the equivalent of fifteen 2400' reels of 6250 bpi tape.

Using a collection of Sun-3 and Sun-4 processors, the authors built a pre-encrypted dictionary from a list of about 107,000 trial passwords. Encrypting each trial password 4,096 times (once for each possible salt value) took several CPU-weeks on these machines; the results fit onto two 8mm cassettes. (Recomputing the dictionary would now take less time, since the fast crypt algorithm was still being tuned while we were generating the dictionary.)

Each encrypted password is stored as an 8-byte value; the plain text is not stored on the tape. Not only does this reduce the amount of tape necessary, but the tapes alone are enough to determine whether an encrypted password is in the password list without revealing the plain text password. This is ideal for improving system security without the possibility of the tapes being used to infiltrate other systems.

The cassettes can be replayed repeatedly and checked against lines from the /etc/passwd file. At a tape transfer rate of 250 kilobytes/second, the CPU can keep up easily with the comparisons; thus the system checks about 30,000 trial passwords/second, faster than the fast crypt code runs in real time. The tapes also can be supplemented with tapes produced from new passwords and with the fast crypt program to check words of local interest, such as names of employees or projects.

The precomputed dictionary approach is therefore 28.6 times faster than real time encryption on a DEC 3100. Twelve tapes containing the encrypted versions of about 732,000 trial passwords would take a day to read on one drive; the crypt implementation running on a single DEC 3100 would take a month, assuming that the password file contains at least one entry for each possible salt. Even so, as long as users change their password less often than once a month, many passwords still can be found. The precomputed dictionary is helpful but not essential for password cracking.

4 Improving Password Security

Since this paper discusses the security of password systems, other possible human-to-machine authentication systems (such as retinal scans or fingerprints) are not considered here. The purpose of this section is to examine the necessary elements for successful password cracking and to suggest which elements might be changed to increase the difficulty of cracking passwords. The elements required to crack passwords as discussed in this paper are:

- High performance/price ratio computers
- Large on-line word lists (dictionaries, etc.)
- A known password encryption algorithm
- A constraint on the acceptable running times for the login program
- A publicly-readable password file
- Passwords with a significant probability of being in the word list

The existence of high performance/price ratio computers and on-line word lists cannot be controlled, so if the difficulty of cracking passwords is to be increased, then one of the remaining four conditions must be changed.

4.1 Known Encryption Algorithm

We consider it a given, as did Morris and Thompson, that the encryption algorithm used for the one-way password crypt function must be published and subjected to public scrutiny. As in cryptography, it is neither practical nor necessary to base the security of a password algorithm on its secrecy. The storm of protest in response to the NSA's recent attempt to replace DES with a secret cipher of its own design indicates the importance of this principle. Furthermore, the enormous success of the UNIX operating system is based largely on the openness of its design and the availability of its algorithms and source code. Assuming that the basic algorithm has not been compromised, there is no real reason to change it.

4.2 Acceptable Running Times

Software de-facto standards, such as the UNIX password algorithm, tend to outlive their original underlying hardware. UNIX now runs on a wide range of processors, from IBM PC/XTs to Cray IIs. A crypt function that is slow enough to thwart password attacks on a Cray II would be intolerably slow to a user logging into an IBM PC, and of course what is slow but tolerable on today's machines will become unacceptably fast on tomorrow's. It is ironic that current fast crypt implementations on 1989 hardware run faster than the old (and "unacceptably fast") Version 6 crypt ran on 1979 hardware.

Also, a crypt routine written specifically for password cracking runs orders of magnitude faster than a version built into a login command. Many of the performance enhancements (e.g., precomputation and the factoring of permutations) are meaningful only for a large scale password cracker that is trying many more keys than encrypted passwords. Also, the amount of memory required for the large tables in fast crypt implementations is not acceptable for a utility program that is to be a standard part of the system. The adversary can afford to dedicate an entire system to password cracking; the "good guy" is unlikely to be willing to dedicate his entire system to the login program.

Morris and Thompson purposely slowed the execution speed of the UNIX crypt implementation in an attempt to hinder password cracking. Adjusting crypt's speed so that the slowest processor running UNIX still has an acceptable login delay thwarts attack only slightly; the fundamental problems with current passwords are not addressed. Other approaches are more likely to be successful. We believe the solution to the password cracking problem lies *not* in penalizing both the good guy and the adversary by deliberately slowing the algorithm. A better approach lies instead in "driving a wedge" between them by increasing the entropy of user passwords. This forces the adversary to search such a large key space as to be impracticable even with a fast algorithm running on the fastest available computer for the foreseeable future.

4.3 Encrypted Password Availability

A resource available to the adversary that is removable is the existence of a publicly-readable encrypted password file (/etc/passwd). For the purposes of this paper, it is assumed that physical access to the machine alone is enough to subvert it and it is assumed that the machine itself is physically secured according to the desired level of security. Many machines can be rebooted into privileged mode with physical access, so that physical access implies that a password-based attack is really unnecessary for system access.

One method of restricting availability of the encrypted passwords is the *shadow password* file. A shadow password file is a file that contains the encrypted passwords and is not publicly accessible. This prevents the average user from accessing the password file. One problem with this approach is that a system administrator has access to the file. This is not a problem while the person remains an administrator, since presumably he has access to everything anyway. The problem is when a system administrator leaves. He may have copied encrypted passwords while he was an administrator and can now use these passwords to infiltrate the system. Also, if an error in access permission is made, the encrypted password file may become available to all users of a system. Shadow password files cannot hurt, but it seems unwise to count on them alone for system security.

A more promising, if more complex, method is to provide each user with a *smart card* that is used to authenticate the user to the system. A smart card in this case is a small computer with a keypad and a general purpose processor. The smart card should be able to communicate directly with the login system (e.g., by electrical or infrared link) rather than require manual I/O. The smart card is more than a physical key; the human authenticates himself to the smart card using a password, so the card alone is worthless. Note that the encrypted password resides in the card and cannot be obtained without possession of the card. Once the human authenticates himself to the card, the card authenticates itself to the computer by some other method not discussed here. Some possibilities include zero-knowledge proofs and public key systems. The main problem would seem to be the rather limited processing capacity of a smart card. However, there are ways of utilizing computing power in untrusted machines to execute complicated secret computations efficiently[7].

4.4 Decreasing Password Guessability

The main weakness in any password system is that users often choose easily guessable passwords: English words, names, trivial extensions to English words, etc., because they are easy to remember. It is important that passwords be difficult to guess. One way to decrease password guessability is to eliminate common passwords from /etc/passwd. At Bellcore, we are using our high-speed password cracking system and a large dictionary to find easily-guessed passwords on systems. Since we have a faster crypt and more CPU cycles than potential intruders, this allows us to use a larger dictionary; thus any passwords that survive our scrutiny are unlikely to be found by an intruder.

Another possibility is to restrict the passwords accepted from the user with a system that filters out easily guessed passwords. Different schemes of filtering are possible and one such method was described by Morris and Thompson[9]. This system acts as a password advisor that indicates insecure passwords, but it does not force the user to accept its recommendation. In some Applied Research laboratories in Bellcore, passwords must have certain characteristics before they are accepted by the system when the user changes his password. In addition, all existing passwords are periodically checked against a large dictionary and anyone whose password is found is forced to change it.

The most drastic solution is to have the system assign an arbitrary password. The problem is that such a password is hard to remember, so the temptation to write it down is strong. A written password is like a physical key, and can be used by anyone who obtains it. A slightly friendlier version of this system assigns a password, but allows the user to rearrange it to make it easier to remember. Then the new password is checked; if it is still acceptable, it becomes the user's new password.

A fundamental problem is that passwords typed by the user are truncated to 8 characters in length. Easily remembered passwords that are this short almost inevitably have much less than the 56 bits of entropy allowed by the crypt algorithm, making them easier to find by exhaustive search. All of the techniques just described attempt to increase entropy in the users' passwords, but they do it in a way that ignores human factors considerations. Almost anyone can remember 56 bits of arbitrary information, but he must be allowed to do it in a way that is suited to human, not computer, memory. The way to do this is by extending the present algorithm to allow *pass phrases*[10]. A pass phrase is simply a longer version of a password that includes several words. According to Shannon [11], English text has a lower bound of 1-2 bits of entropy per character. Therefore an ordinary English phrase of 5-10 words (assuming 5-6 characters/word and no unusual punctuation or capitalization) has sufficient entropy as a pass phrase.

To accommodate this in the UNIX crypt algorithm, a hash function is needed to fold the typed pass phrase into 56 bits, with each input character affecting the result. This function should be backward compatible with the existing UNIX password algorithm for pass phrases of 8 characters or less. One possibility is to treat the first 8 characters as before, exclusive-ORing into it each successive 8-character block from the pass phrase (if the phrase is not a multiple of 8 characters, it is null-padded on the right).

Users might still object to pass phrases if they were required to type them too frequently (e.g., when they must repeatedly log into a several different systems, each for short intervals). A solution to this problem lies in the use of an distributed authentication system such as Kerberos, in which the user need type his password only once to obtain a set of "tickets" that can be used to access other systems repeatedly without having to retype the pass phrase each time[13].

Not only is the entropy of the password important, but also the amount of time available to the cracker to find it. The more infrequently that a user changes his password, the more vulnerable it is to a cracker. Thus it is important that the user occasionally change his password. On System V UNIX, this is accomplished by including an *aging field* for each line of the /etc/passwd file. When a password is sufficiently old, then the user is prompted for a new one. Of course, the system should check that the new password is different from previous ones.

4.5 Other Approaches

Two suggested solutions to the problem of easily cracked passwords are to increase the size of the salt or to change the constant that is encrypted by crypt. Neither of these seems to be particularly helpful.

Increasing the size of the salt does not help prevent attack on an individual password, but it does help defeat checking multiple passwords simultaneously and pre-encrypted wordlist attacks by increasing the time and space required, respectively. The current salt is large enough that few of the lines in a typical /etc/passwd file share the same salt. The only remaining reason to increase the size of the salt is to reduce the number of pre-encrypted passwords that can fit onto a fixed amount of tape. But as shown above, pre-encryption decreases the cracking time by a factor of 30, so this is the maximum penalty that could be exacted by even a large increase in the salt size.

Making the starting constant used by crypt a system configuration variable instead of all zeros was mentioned by Morris and Thompson[9]. This has the advantage of making it harder to use the pre-encrypted wordlist, but unless the constant is kept secret it is still possible to attack individual passwords with a fast crypt. Depending on the implementation, a user might be able to learn the constant by decompiling the login program. If a password/encrypted password pair is known, then the starting constant can be determined simply by reversing the internal crypt operations. The encrypted passwords are known if the /etc/passwd file is publicly readable; a plain text password that matches an encrypted password is available to any user (his own password) and some passwords (such as for UUCP) may be widely known. An adversary with access to the /etc/passwd file alone but no knowledge of the constant could take advantage of the fact that many passwords are in the dictionary. As before, crypt can be run backwards for a dictionary of words on several passwords and the corresponding input constants checked for a match. Once a matching pair of constants is found (there are 2^{64} or about 1.8×10^{19} possibilities for the constant, so a match is unlikely unless both are the hidden constant), the number can be verified simply by using one of the plain text keys that produced the match to attempt to log in. If the attempt is successful, the hidden constant is known and cracking continues as usual. The only cost above the usual cracking technique is the maintenance of the list of constants until a match is found. Table 2 shows the probability of finding the unknown constant after checking n encrypted passwords, for 10%, 20%, and 30% probabilities of finding a password in the dictionary.

Once chosen, the constant could not be changed easily (e.g., in event of compromise) without having every user re-enter his or her password. Another disadvantage of the system-configurable constant is that /etc/password files would no longer be portable across systems with different encryption constants, although it might be argued that such portability de-

Number of Passwords	Success Probability (10%)	Success Probability (20%)	Success Probability (30%)
5	8%	26%	47%
10	26%	62%	85%
15	45%	83%	96%
20	61%	93%	99%

Table 2: Probability of Cracking Two or More Passwords

creases system security by allowing arbitrary (and perhaps easily guessed) passwords to be put in the /etc/password file.

To keep the constant secret, it is necessary to couple the secret constant technique with shadow password files. Since this technique must be implemented in conjunction with shadow password files, has no advantages over that of shadow passwords alone, and has several disadvantages, it is better to implement password file shadowing only.

5 Conclusion

The current UNIX password system is not always sufficient to prevent unauthorized entry because it is fairly easy to crack passwords. An important point is that although the crypt algorithm is a good one, the password system as a whole is weak. Six factors contribute to the ease of cracking passwords: high performance/price ratio computers, large on-line word lists, a known password encryption algorithm, a maximum acceptable running time for the login program, a publicly readable password file, and easily guessable passwords.

Nothing can be done about large on-line dictionaries or high performance/price ratio computers. In fact, the password system should take the exponential speed increase of computers into account. It is argued that the password encryption algorithm must be known to be trusted and that there is a range of acceptable running times for the algorithm which sets an upper limit on the amount of computation that the password encryption algorithm may use. Unfortunately, the computation limit is small enough to allow faster machines to use a dictionary-based attack. It is also argued that Morris and Thompson's assertion that slowing down the implementation of the crypt function improves security does not address the large range of processors that run UNIX.

Two of the main problems with the current system are that users choose easily guessable passwords and that the encrypted password file is publicly readable. A dual approach is suggested. One part is to make passwords less predictable by allowing pass phrases and restricting passwords accepted by the system. This effectively increases the entropy of a password, making wordlist attacks less successful. The other approach is to make the encrypted password file less accessible. How exactly this is done depends on the desired level of security and includes shadow password files and smart cards.

References

[1] Robert W. Baldwin. MIT fdes 5 (crypt) source code.

[2] Matt Bishop. An application of a fast data encryption standard implementation. *Computing Systems*, 1(3):221–254, Summer 1988.

[3] Marc Davio, Yvo Desmedt, Marc Fosseprez, Rene Govaerts, Jan Hulsbosch, Patrik Neutjens, Philippe Piret, Jean-Jacques Quisquater, Joos Vandewalle, and Pascal Wouters. Analytical characteristics of the DES. In *Proceedings of Crypto '83*, pages 171–202, August 1983.

[4] Marc Davio, Yvo Desmedt, Jo Goubert, Frank Hoornaert, and Jean-Jacques Quisquater. Efficient hardware and software implementations for the DES. In *Proceedings of Crypto '84*, pages 144–146, August 1984.

[5] W. Diffie and M. E. Hellman. Exhaustive cryptanalysis of the NBS data encryption standard. *Computer*, 10(6):74–84, June 1977.

[6] Alan G. Konheim. *Cryptography: A Primer.* John Wiley & Sons, 1981.

[7] T. Matsumoto, K. Kato, and H. Imai. Speeding up secret computations with insecure auxiliary devices. In *Proceedings of Crypto '88*, August 1988.

[8] Donald Mitchell. AT&T Questor (crypt) source code.

[9] Robert Morris and Ken Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, November 1979.

[10] Charles P. Pfleeger. *Security in Computing.* Prentice Hall, 1989.

[11] Claude Shannon. Prediction and entropy of printed english. *Bell System Technical Journal*, 30(1):50–64, January 1951.

[12] Eugene H. Spafford. The internet worm program: An analysis. *Computer Communication Review*, 19(1):17–57, January 1989.

[13] J.G. Steiner, C. Neuman, and J.I. Schiller. Kerberos: An authentication service for open network systems. In *USENIX Conference Proceedings*, pages 191–202, Dallas, Texas, February 1988.

A A High-Speed Crypt Implementation

This appendix[2] describes a high-speed software implementation of the UNIX crypt algorithm. This new crypt is 102.9 times faster than the crypt in 4.2 Berkeley UNIX on a Sun 3/50. Many of the results are also applicable to software *Data Encryption Standard* (DES) and other product cipher implementations.

Several techniques are used to increase the program speed. One technique is to alter the crypt algorithm so that it is easier to compute but still produces the same results. Another technique is to take advantage of the architectural features of the computer that will run

[2] This appendix was originally a paper entitled *A High-Speed Crypt Implementation* by David C. Feldmeier.

the algorithm. It also is important for high performance to minimize the manipulation of individual bits. A data representation is described that allows the E expansion of DES to be accomplished by a simple register copy and yet allows a fast implementation of the S function of DES without bit manipulation. This appendix assumes that the reader is familiar with the DES [6].

A.1 Overview of Crypt

Crypt is a program used by the UNIX operating system to encrypt user passwords and is based on the DES encryption algorithm. Crypt is designed to be a one-way function; given an input, it is easy to compute the output, but given an output, it is impossible to determine the corresponding input except by guessing. DES can be used to realize a one-way function, so it is a good choice to use for crypt. In particular DES is resistant to a known plain text attack, which means given the plain text and the cipher text, the key can be found only by exhaustive search. The crypt algorithm uses 25 successive DES encryptions of a constant (64 zeros) to produce the encrypted password. The key for all of the DES encryptions is derived from the user's password. The first eight characters of the password are used as a 56-bit key (7 bits for each ASCII character, 8 characters). If the password is less than 8 characters, the password is padded with zeros to the full 56-bit length.

Actually, crypt does not use pure DES. To prevent use of off-the-shelf high-speed DES hardware to crack passwords, crypt modifies the DES algorithm slightly. A randomly generated *salt* is included with each entry in the /etc/passwd file. The 12-bit salt ranges from zero to 4095. Think of the salt as a permutation that immediately follows the expansion function E in DES. If bit 1 of the salt is a 1, then the salt permutation swaps bits 1 and 25 of the 48-bit block generated by E. If bit 2 is a 1, then bit 2 and 26 are swapped and so on. Since there are 12 possible swaps and any combination of these swaps may occur, this produces 4096 possible variations of DES (a salt of zero corresponds to pure DES). More details on the UNIX password mechanism are found in a paper by Morris and Thompson[9].

A.2 The Speed-Crypt Implementation

This section describes some of the ideas behind the implementation of the crypt algorithm written by the author (*speed-crypt*) and why it is fast. The implementation is designed for 32-bit machines; it is possible to run it on other size machines with minor modifications. The DES algorithm has 48-bit, 56-bit and 64-bit wide paths. Since even 48 bits is wider than the expected data path, each operation requires two word manipulations; thus the DES data path is broken into high and low pieces of 32 bits each.

A.2.1 Algorithm Modifications

The basic crypt algorithm can be modified in a number of ways that do not change the function computed by the algorithm. Each DES encryption begins with the *initial permutation* (IP) and ends with the *inverse initial permutation* (IP^{-1}). These two permutations are inverses of each other, so when two DES encryptions are concatenated, the IP of the second encryption immediately follows the IP^{-1} of the first encryption. There is no reason to do either of these permutations, since the net result is no permutation at all. Therefore, any IP^{-1}-IP pairs can be factored out of the algorithm and only the first IP and the last IP^{-1} ever need be done. In fact, because of how crypt works, these remaining permutations can be factored out as well.

Inside of DES, the 64-bit input block is broken into two pieces of 32 bits each called *Left* (L) and *Right* (R). Within each DES encryption there are 16 product-transformation/block-transformation pairs. The block transformation is simply the swapping of the R 32 bits with the L 32 bits. To avoid the block transformation at the end of each product-transformation, speed-crypt uses two different product transformations: one works the usual way and the other operates on L as if it were R and vice-versa. Using these two product transformations alternately eliminates the need for a block transformation between product transformations. The only problem with this scheme is that after the last product-transformation, the L and R blocks are swapped. This reversal is incorporated into the final IP^{-1} because a swap and a permutation is just a permutation. Let ϵ represent the E expansion, σ represent the S function and π represent the P permutation. Let K_i be the i^{th} subkey, R_i be the i^{th} value of Right and L_i be the i^{th} value of Left. By definition:

$$L_i \equiv R_{i-1}$$

$$R_i \equiv L_{i-1} \oplus f_i R_{i-1}$$

$$f_i R_{i-1} \equiv \pi\sigma(K_i \oplus \epsilon R_{i-1})$$

where f_i is the product transformation that uses subkey K_i. After two rounds:

$$L_{i+1} = L_{i-1} \oplus f_i R_{i-1}$$

$$R_{i+1} = R_{i-1} \oplus f_{i+1}(L_{i-1} \oplus f_i R_{i-1})$$

Since there are 16 rounds in DES, 8 double-rounds of the following form can be used instead:

$$L_{i+1} \equiv L_{i-1} \oplus f_i R_{i-1}$$

$$R_{i+1} \equiv R_{i-1} \oplus f_{i+1} L_{i+1}$$

Notice that no intermediate values of R_i and L_i need be retained. After one double-round:

$$L_{i+1} = L_{i-1} \oplus f_i R_{i-1}$$

$$R_{i+1} = R_{i-1} \oplus f_{i+1}(L_{i-1} \oplus f_i R_{i-1})$$

which is the same as before. In effect, the swap has been built into the iteration.

Because the crypt program begins by encrypting all zeros, the first IP permutation and the first E expansion can be factored out because any permutation or expansion of zero is still zero. The first salting operation can be factored out for the same reason.

Another method of increasing speed depends upon the assumption that there are fewer encrypted passwords to be checked than words to be tried. Under these circumstances, it makes sense to do as many operations as possible on the encrypted passwords if operations can be avoided on the words. The encrypted passwords should be operated on to allow their comparison with the results of the crypt program as early as possible, since a single backward step on the password saves as many forward steps as there are words. The final IP^{-1} can be skipped if you are checking to see whether a password is in a wordlist. Instead of doing the IP^{-1} permutation for each word that is being tested as the password, a better way to do this is to take the encrypted password from the passwd file and permute it with IP. The comparison is now done between the output of the last DES round and the permuted encrypted password.

A.2.2 Subkey Generation

Subkey generation means taking a 64-bit password key K and generating 16 48-bit DES subkeys K_i. The generation of subkeys involves taking a 64-bit plain text password and applying the reduction/permutation Permuted Choice 1. Permuted Choice 1 reduces the password from 64 bits to 56 bits by eliminating the parity bits and then permutes the result. The 56-bit result is then divided into low and high 28-bit halves and each half is left-circular shifted by an amount that depends on the particular subkey being generated. The two 28-bit halves are joined and the permutation/reduction Permuted Choice 2 is applied. Permuted Choice 2 permutes the 56-bit result of the rotation and then selects 48 of these bits for the subkey. The combination of the permutations and reductions for each subkey is combined into a single permutation/reduction, for a total of 16 subkey generation functions. Let α represent Permuted Choice 1, β represent Permuted Choice 2, and ρ_i represent the rotations for the i^{th} subkey. Let K_i be the i^{th} subkey and K be the key that the subkey is generated from. Then:

$$K_i \equiv \kappa_i K$$

$$\kappa_i \equiv \beta \rho_i \alpha$$

where κ_i is the permutation/reduction that generates the i^{th} subkey from the original password. To limit the table size, subkey lookups take seven key bits at a time (each ASCII password character is represented by 7 bits). The lookup could take any number of bits at a time because all are independent, but doing lookups a character at a time is convenient and two characters at a time makes the tables too large.

For each character in the password, the partial subkey is found and logical-ORed with the partial subkeys for the other characters in the password. After 16 passes (each requiring two lookups) for each character in the password (a maximum of 128 passes), all 16 subkeys have been generated. At most 256 table lookups are needed to generate all subkeys. A nice side-effect is that the time required for this method of subkey generation is proportional to the password length. Since passwords of less than eight characters are padded with zeros and permutations of zero are also zero, the subkeys will not be changed by these additional zeros, so there is no need to bother with them. The total table size is 2^3 character positions in each password, 2^7 possible characters in each position, 2^4 subkeys per password and 2^3 bytes to hold each subkey for a total table size of 2^{17} or 131,072 bytes.

A.2.3 Table Lookup

The program gets a lot of its speed from a space-for-time tradeoff - almost everything in the program is done by table lookup. Ideally, a table lookup would take the entire input (up to 64 bits) and return the entire output (up to 64 bits). Of course, the maximum size of the input and output of a table are limited by the virtual memory size and the bus width. Thus, on a 32-bit bus, a table with a 64-bit output requires that two lookups be done, one for the low 32 bits and one for the high 32 bits. As for the input to the table, even a 32-bit input would be completely impractical because it is desirable to keep all of the tables in main memory for fast access time and to prevent paging.

To replicate the effect of a single large lookup table with several small lookup tables presents a problem. The problem is that groups of input bits may exist such that all the bits must be read simultaneously to produce a result. An example of this is the S boxes in DES. Each S box takes a 6-bit input and produces a 4-bit output. Because all 6 bits are simultaneously necessary (because the S box is non-affine), all S box lookups must be done in multiples of 6 bits.

However, if groups are independent, then the table lookup can be broken into manageable pieces (ideally of equal size to minimize the total table size), each of which can be manipulated independently. The results can then be logical-ORed together to produce the appropriate output. Because the physical memory available to an application in most workstations is only a few megabytes, this limits the size of the tables.

Changing one lookup to two produces a substantial change in total table size. The ratio is $2^{(n/2)+1}/2^n$ or $2^{1-(n/2)}$, where n is the number of bits used for the table index. The fewer the table lookups, the higher the speed, but also the more memory that is used. The crypt implementations should use as few table lookups as possible given the memory constraints.

The IP^{-1} table takes the 64-bit input a byte at a time and produces two 32-bit outputs that are ORed together in the usual way. The IP table is used only for permuting encrypted passwords that are being searched for to avoid computing the IP^{-1} for each word in the dictionary (remember that IP and IP^{-1} are inverse permutations). Since IP is used on password entries, the password entries must be converted from ASCII form to a 64-bit form. Speed-crypt uses a special version of IP that converts directly from the 11 ASCII character format of the /etc/passwd file to the 64-bit format after the IP permutation. IP does lookups a character at a time and produces two 32-bit outputs that are ORed together.

The S boxes and the P permutation are combined into a single lookup table. The expansion function E is done with a simple register copy. The details of the E expansion will be explained later. Table 3 shows the table sizes for speed-crypt.

Table	Bytes
SP	65,536
key	131,072
IP^{-1}	16,384
total	212,992

Table 3: Table Size in Speed-Crypt

A.2.4 Data Representation

A representation is devised that does not require a table lookup for the expansion function E, allows the S/P function to be implemented as 4 table lookups without the manipulation of individual bits, and allows fast salting. Such a representation is possible, but requires a strange bit order.

Consider the mapping from R (32 bits) to ϵR (48 bits) as shown in table 4. The first thing to notice is that the table is presented in groups of 6. The reason for this is that the eight S boxes each use six bits for their lookup. Because the S boxes are non-affine, the groups of six cannot be broken up. Therefore, they define the granularity of the table lookups (multiples of six bits along the boundaries shown).

An important aspect of the E expansion is that no input bit of R ever becomes more than two output bits in ϵR. This suggests that simply copying R into a second register will give all 48 bits necessary for the expansion E in two 32-bit words. Designate the two copies of R as A and B.

ϵR (48 bit)	1	2	3	4	5	6
R (32 bit)	32	1	2	3	4	5
ϵR (48 bit)	7	8	9	10	11	12
R (32 bit)	4	5	6	7	8	9
ϵR (48 bit)	13	14	15	16	17	18
R (32 bit)	8	9	10	11	12	13
ϵR (48 bit)	19	20	21	22	23	24
R (32 bit)	12	13	14	15	16	17
ϵR (48 bit)	25	26	27	28	29	30
R (32 bit)	16	17	18	19	20	21
ϵR (48 bit)	31	32	33	34	35	36
R (32 bit)	20	21	22	23	24	25
ϵR (48 bit)	37	38	39	40	41	42
R (32 bit)	24	25	26	27	28	29
ϵR (48 bit)	43	44	45	46	47	48
R (32 bit)	28	29	30	31	32	1

Table 4: Expansion Function E (mapping from R to ϵR)

Since as little manipulation as possible of the 32-bit quantities before the SP lookup is desired, the bit order of the 32-bit R (and symmetrically L) is critical. The aim is to do four lookups, two S boxes at a time. Notice that there is a circular structure to the E expansion and that bits from R that occur on line n also occur on lines $(n+1)$ and $(n-1)$ mod 8. This means that the lookup of all odd lines must occur in one copy of R (say A) and the lookup of even lines must occur in the other (B) so that overlapping bits of R may be salted differently. Salting constrains the data representation in yet another way. Because salt exchanges bits between lines 1 & 5, and lines 2 & 6, lines 1 & 5 must be read in a single lookup, as must lines 2 & 6. This also implies that lines 3 & 7 must be a single lookup, as must lines 4 & 8.

To minimize table size, lines 1 & 5, 2 & 6, 3 & 7, and 4 & 8 must be organized as blocks of 12 bits with no intervening bits. In addition, the circular structure of the E expansion requires that the bits for lines 8 and 2 be adjacent to those for line 1, lines 1 and 3 be adjacent to those for line 2, etc. One way to achieve this is to interleave the bits of the pairs of lines. Thus lines 1 & 5 are interleaved, lines 2 & 6 are interleaved, etc. Of course, interleaved lines 1 & 5 are adjacent to interleaved lines 4 & 8 and 2 & 6. Interleaving not only allows the correct adjacencies but also makes salting easy, since aligning the bits for comparison takes only a single shift.

Assume that the bits of a word are numbered such that 0 is the least significant bit and 31 is the most significant bit. The data representation for R starts with the first bit of line 1 in bit 2, the first bit of line 5 in bit 3 and so on, thus interleaving lines 1 and 5. This continues with lines 2 and 6. Notice that bit 10 represents not only the fifth bit of line 1 but also the first bit of line 2, and bit 11 represents the fifth bit of line 5 as well as the first bit of line 6. This pattern continues for lines 3 & 7 and 4 & 8. Notice that lines 4 & 8 wrap around the end of the word, and thus the fourth bit of line 4 is represented by bit 0. This is the 32-bit representation of R:

30	14	29	13	28	12	27	11	26	10	25	09	24	08	23	07	22	06	21	05	20	04	19	03	18	02	17	01	16	32	31	15

To get the 32-bit representation of L (which is not expanded), add 32 to each of the above numbers. Notice that lines 1 & 5 do not start immediately at the least significant bit; there are two extra bits in the least significant byte. This is because when doing pointer arithmetic, normally the value to be found in a table on a machine with a 32-bit (4 byte) word size would have to be multiplied by 4 before addition. In effect, the data representation above "premultiplies" by 4, thus saving an operation in a critical section of the code.

Now apply the mapping from R to ϵR for 1 & 5 and 3 & 7 (which have no overlapping bits in R) to obtain the representation of the 24 of the 48 bits in A (x denotes an unused position):

xx	xx	42	18	41	17	40	16	39	15	38	14	37	13	xx	xx	xx	xx	30	06	29	05	28	04	27	03	26	02	25	01	xx	xx

Now apply the mapping from R to ϵR for 2 & 6 and 4 & 8 (which have no overlapping bits in R) to obtain the representation of the 24 of the 48 bits in B:

45	21	44	20	43	19	xx	xx	xx	xx	36	12	35	11	34	10	33	09	32	08	31	07	xx	xx	xx	xx	24	48	23	47	46	22

A.2.5 Salting

Unlike other functions in crypt, such as E, S and P, the salt permutation is determined at runtime. Salting takes place in the 48-bit data path after the expansion function E and before the subkey is exclusive-ORed with the expanded R. The salt acts as an additional permutation after the E expansion, which can be either done separately or combined with other permutations. Because of the nature of the E expansion, complete salting cannot be done before E (although partial salting can). Let τ represent the salt permutation; then:

$$f_i R_{i-1} \equiv \pi\sigma(K_i \oplus \tau\epsilon R_{i-1})$$

If salting is done separately, then after the E expansion the appropriate bits are swapped according to the salt. For many keys to be encrypted with the same salt, *presalting* may be faster. Presalting involves combining the salt permutation with one or more of the table lookups at runtime before any encryption is done and then using these modified tables for encryption. Since in-line encryption uses one salt operation for each DES round, 400 salt operations are performed on each word. With enough encryptions, it is cheaper to use the salt to adjust each entry of the lookup tables appropriately.

The salting function is relatively fast. Because the salting operation is so regular, it is faster to compute it than to do a table lookup. Speed-crypt has a data representation such that the relative offsets of bits 1-12 is the same as that of bits 25-36, i.e. the distance between 1 and 2 is the same as the distance between 25 and 26, etc. This means that bits 1-12 can be aligned with bits 25-36 in a single operation.

The only bits that need be swapped are those that differ and this *delta bitmap* is computed by exclusive-ORing bits 1-12 with 25-36. Then the delta bitmap is logical-ANDed with the salt mask. The salt mask has a 1 in the positions where bits are to be salted and 0s elsewhere. This leaves a bitmap of those bits that differ and are supposed to be switched in the delta bitmap. This delta bitmap is then exclusive-ORed with bits 1-12 and 25-36, thus completing the salting operation.

Presalting is used exclusively for speed-crypt, since the crossover point for presalting versus in-line salting is only 10 words. Presalting involves altering lookup tables to eliminate the need to exchange bits after the E expansion. The only table lookup in speed-crypt is the SP table, which produces a 32-bit result. Sometimes bits that both can be salted after E appear as a single element in the 32-bit representation of R. This is a problem as the 32-bit element can salt the single bit one way, and perhaps have bits be swapped incorrectly for one of the two lookups. The solution is to reorder the entries of the SP table according to salt. By definition:

$$R_i \equiv L_{i-1} \oplus \pi\sigma(\kappa_i K \oplus \tau\epsilon R_{i-1})$$

Using the fact that $\tau\tau^{-1}$ is the identity permutation:

$$R_i = L_{i-1} \oplus \pi\sigma\tau\tau^{-1}(\kappa_i K \oplus \tau\epsilon R_{i-1})$$

Now distribute τ^{-1} across $\oplus$:

$$R_i = L_{i-1} \oplus \pi\sigma\tau(\tau^{-1}\kappa_i K \oplus \tau^{-1}\tau\epsilon R_{i-1})$$

Canceling $\tau^{-1}\tau$:

$$R_i = L_{i-1} \oplus \pi\sigma\tau(\tau^{-1}\kappa_i K \oplus \epsilon R_{i-1})$$

Because τ is its own inverse:

$$R_i = L_{i-1} \oplus \pi\sigma\tau(\tau\kappa_i K \oplus \epsilon R_{i-1})$$

The table entries have to be permuted or reordered, depending on whether the incorporated permutation is before or after the table.

$$\kappa_i(j) \leftarrow \tau(\kappa_i(j))$$

$$\pi\sigma_i \leftarrow \pi\sigma_{\tau(i)}$$

Thus all of the key table entries are salted with the usual salt function. SP table entries must be exchanged and this can be done quickly. If two bits to be salted are the same (when $i = \tau(i)$), then no exchange is necessary. Since $\tau = \tau^{-1}$, table entry $\pi\sigma_i$ can be swapped with entry $\pi\sigma_{\tau(i)}$ and no temporary storage is needed.

A.2.6 System Issues

Crypt runs DES 25 times and the basic round within DES is run 16 times for a total of 400 rounds. Anything that can be done to speed up this basic round will be multiplied by 400, so it is important that the rounds run efficiently.

In general it is best to keep the number of variables used by the crypt program small so that most of them can be kept in registers. On a fast processor, particularly those with caches, memory fetches slow down the system.

It is best to avoid instructions whenever possible. For example, with the SP table lookup, rather than logical-ORing the table entries together and then exclusive-ORing the result into the L register, it is faster to exclusive-OR the intermediate results into the L register directly, saving one instruction per round (400 instructions total).

For processors with a cache, the main DES rounds should be executed within a loop that is small enough to fit into a processor cache. The execution speedup achieved by the cache more than compensates for the running time of the extra loop instructions. Without a cache, the loop should be unwound so that the loop overhead is avoided. Speed-crypt has a compile-time option to structure the program appropriately depending on whether there is a cache.

Another important point is to take advantage of the processor instruction set, specifically whether the machine is a *Reduced Instruction Set Computer* (RISC) or a *Complex Instruction Set Computer* (CISC). Examples of RISC machines are the Sun 4 (SPARC processor) and the DEC 3100 (MIPS R2000 processor). Examples of CISC machines are the Sun 3 (68020 processor) and the VAX.

A useful feature of CISC processors is auto-increment mode. If possible, it is best to step through tables one element at a time so that auto-increment mode can be used. This is particularly useful for accessing the subkeys. The high and low words of the subkeys alternate and they are extracted one at a time with auto-increment mode. On a RISC machine, a separate addition must be done for each increment, so it does not matter what the step size is. Sometimes a different step size can lead to more efficient operation. Another feature of CISC processors are instructions specifically designed for efficient loops. In particular, it is faster to count down to zero with a single instruction that does the compare-decrement-branch function.

RISC machines generally contain a large number of registers and performance is enhanced if often-used constants are kept in registers. Large constants take two instructions to load into the processor rather than one (this is because the fixed-length RISC instructions can include only small constants).

Speed-crypt is written in C and should be portable with little trouble, although for some machines there are special cases inserted into the code. In particular, sometimes a compiler cannot be convinced to generate efficient assembly code for some portion of C code. An example of poorly generated code is the loop instructions. Compilers sometimes generate non-optimal loops, particularly for CISC processors, which often have good instructions for loops. One case where the compiler is not at fault is the lack of a bit rotation operator in C. Many processors have a bit rotation instruction, but the equivalent in C produces 3 assembly language instructions. In both of these cases there are compile time options to replace some portions of the C code directly with assemble language instructions for specific processors. Another possibility is to edit the assembly language that the compiler produces. Because the VAX compiler is reluctant to use all of the processor registers, speed-crypt on the VAX uses a SED (UNIX Stream EDitor) script that replaces certain memory references with register references in the assembly code.

A.3 Implementation Alternatives

Fast crypt implementations by Baldwin[1], Bishop[2] and Mitchell[8] each utilize most of the suggestions above. Fast DES implementations also use similar techniques[4]. Speed-crypt has a technique for the E expansion that can double the speed of an implementation on 32-bit RISC machines.

The *questor* code was written by Donald Mitchell at Bell Labs and is probably the most straight-forward of the fast crypt implementations[8]. Subkey generation uses two table lookups (Permuted Choice 1 and Permuted Choice 2) and two rotations per subkey. The E expansion is done as eight 64-bit table lookups (4 bits of R at a time). The S function and the P permutation are combined and eight 32-bit lookups are done (6 bits at a time). The swap of L and R occurs between DES rounds. The IP^{-1} lookup is also done 4 bits at a time. The version timed below has presalting, but the original version did in-line salting only. However, the code is well written and it runs quickly.

The implementations by both Baldwin and Bishop use a transformation of DES described by Davio[3]. The recurrence equation is rewritten so that the E expansion is combined with the S function and the P permutation; thus, only a single lookup table is needed for the DES rounds. An ϵ^{-1} function is needed at the end of the 25 DES encryptions, but this can be combined with the IP^{-1} table. The problem with this transformation is that the entire data path of the crypt function is 48-bits wide. This is not a problem on a machine that is 48-bits wide or wider, but it is on a machine with a smaller bus. This doubles the number of memory accesses for the SPE table because each 48-bit word requires two 32-bit fetches. The basic SPE table lookup requires 4 basic lookups, each of which has an 8 byte output, which means that each lookup requires two memory fetches on a 32-bit machine. Because memory fetches slow down the crypt program, speed-crypt executes faster because fewer (4) memory references are needed. The E expansion technique used for speed-crypt has little advantage over a combined SPE table for machines that are 64-bits wide or wider. For machines with smaller bus widths, the speed-crypt implementation should run about twice as fast.

Bob Baldwin wrote the *fdes* code at MIT[1]. The subkeys are computed in the standard way and he avoids the swap between DES rounds. His SPE lookups are done 6 bits at a time. Baldwin has other optimizations that are specific to his design and cannot be implemented in speed-crypt. The fdes program is designed to run well on a VAX. It runs very well on the VAX, but not as well on other machines.

Matt Bishop wrote his *deszip* code at Dartmouth College and the Research Institute for Advanced Computer Science[2]. The code has a variety of options to allow various speed/size tradeoffs. Keys can be computed either the standard way or with a permutation per subkey as speed-crypt does. The SPE lookups can be done 6 or 12 bits at a time. It is also sophisticated about taking advantage of the machine architecture to improve its speed.

A.4 Speed Measurements

Machine	User Time (Seconds)	System Time (Seconds)	Total Time (Seconds)	Crypts per Second	Milliseconds per Crypt
DEC 3100	96.5	1.4	97.9	1089.5	0.92
Sun 3/50	395.2	3.6	398.8	267.5	3.74
Sun 3/60	276.9	7.5	284.4	375.0	2.67
Sun 3/75	343.3	1.1	344.4	309.7	3.23
Sun 3/160	345.1	7.3	352.4	302.7	3.30
Sun 3/200	229.6	1.9	231.5	460.7	2.17
Sun 4/280	101.4	0.8	102.2	1043.6	0.96
VAX 8650	273.2	2.4	275.6	387.0	2.58

Table 5: Speed-Crypt Speeds on Various Machines

Person	Version	Year	DEC 3100	Sun 3/50	Sun 4/280	VAX 11/780
UNIX	bsd 4.2	?	18.8	2.6	16.7	1.4
Baldwin	fdes 5	?	176.9	23.8	123.0	38.2
Mitchell	questor	1984	238.5	31.3	190.3	19.1
Bishop	deszip	1987		71.5		23.9
Feldmeier	speed	1989	1089.5	267.5	1043.6	58.8

Table 6: Crypt Times of Various Implementations

The timings of the speed-crypt implementation are shown in table 5; the length of the test dictionary is 106,661 words. Table 6 shows the speed of the various crypt implementations on a variety of machines. The reason that implementations by others do not speed up as well when they are moved to a RISC machine is that they do many table lookups. The speed-crypt program is larger, but the number of table lookups is smaller, thus allowing it to run faster on a RISC machine. Also, Baldwin's fdes program is optimized for the VAX, not the Sun, so it may not have ported well. Notice that fdes runs 3.2 times faster on the Sun 4 than on the VAX 11/780, while speed-crypt runs 17.7 times faster.

A.5 Conclusion

This appendix describes the implementation of a high-speed crypt program written in C. It discusses how the crypt algorithm works and how it can be modified for higher speed. Implementation decisions and programming tricks for high speed are also discussed. It is worth pointing out that no real breakthroughs were required for the results obtained. What was required is a good understanding of the algorithm and of the computer systems on which it is implemented. The most unique part of the implementation is the unique bit order used in the 48-bit wide product transformation that allows fast E expansion, fast SP table lookup and salting without manipulation of individual bits. Of course, many of the ideas presented in this paper are applicable to software implementations of DES and other product ciphers.

The fastest crypt implementation is 102.9 times faster than the crypt in 4.2 Berkeley UNIX on a Sun 3/50. In absolute speed, the fastest crypt does 1089.5 crypt per second on a DEC 3100.