A Programmable Plaintext Recognizer

David A. Wagner *
Princeton University
Princeton, NJ
dawagner@princeton.edu

Steven M. Bellovin
AT&T Bell Laboratories
Murray Hill, NJ
smb@research.att.com
Phone: +1 908-582-5886
Fax: +1 908-582-3063

Abstract. Other researchers have studied the feasibility of a brute force attack on DES using several known plaintexts. In practice, known plaintext/ciphertext pairs may not be readily available, but statistical information about similar plaintexts is much more easily aquired. Accordingly, we design a statistical plaintext recognizer suitable for use in a ciphertext-only key search machine. Software simulations indicate that this design gives a powerful attack on the encryption of low-entropy data.

1 Introduction

Ever since the Data Encryption Standard (DES) was announced [1], researchers have worried about its relatively short key size. Its cryptographic strength notwithstanding, 56 bits seemed vulnerable to exhaustive search. Diffie and Hellman quickly concluded that a highly parallel custom-designed machine could indeed mount a successful brute force attack on DES with a price tag of about $20 million [2], though that estimate was disputed by some. Others have since refined the details of the design and found a more accurate cost estimate [3, 4, 5]; in one of the most recent papers, Wiener describes how to build a $1 million machine that can break DES in 3.5 hours with one or two known plaintexts [6]. On the other hand, the only other known attacks on DES, linear cryptanalysis [7] and differential cryptanalysis [8], require at least 2^{43} known plaintexts. This is simply too much data to compare favorably to brute force; thus, exhaustive key search is currently the most economical attack known.

In practice, known plaintext may not be readily available. We describe an improvement to the Wiener key search engine which eliminates the need for known plaintext/ciphertext pairs; instead, it uses statistical knowledge about plaintext to decide if it has found a possible key. Given a quantity of reference data, our procedure initializes the modified search engine with statistical information; when the plaintext is similar to the reference data, a few ciphertexts are enough to discover the enciphering key with high probability. We analyze the efficiency and cost of the modified key search machine and consider its implications on the strength of DES. Software simulations of our design confirm its performance.

---

* Work was done while at AT&T Bell Laboratories.

2 Overview

Wiener's design is based upon a machine that contains many copies of a cheap custom-made chip, each searching a portion of the key space. Given a plaintext/ciphertext pair (P, C) the heavily pipelined chip tests keys at the rate of 50 million per second to find K such that $K^{-1}[C] = P$. Each clock cycle, the output of one trial decryption is compared for equality to the known plaintext P. If there is a match, the key is saved; otherwise, the trial key is incremented and the search continues.

Our aim is to find a replacement for Wiener's simple equality test that does not require a known plaintext/ciphertext pair. In theory, trial decryption with the correct key should yield output which is statistically recognizable as similar to the reference data, while trial decryption with an incorrect key will produce uniformly random white noise. Thus, we need a test that can reliably discriminate between true plaintext blocks and random noise. Also, the test must be extremely fast—it should finish in one clock cycle to avoid delaying the decryption pipeline. (More precisely, the aggregate rate of the recognizer, which may itself be pipelined, must match or exceed that of the decryptor.) To do this, it must be composed of just a few simple logical operations, such as additions, table lookups, and comparisons; we want the plaintext recognition stage to be roughly comparable in cost to the DES pipeline. If these conditions can be achieved, then adding one plaintext recognizer to each DES pipeline will make possible a ciphertext-only key search machine which costs at most a small factor more than Wiener's known plaintext version.

Plaintext recognition has been well studied from the perspective of classical cryptography. Published work on automated cryptanalysis of the simple substitution cipher (and other classical cryptosystems) provides a wealth of ideas to examine [9, 10, 11, 12, 13]. Common methods included frequency statistics, digraph and trigraph statistics, the index of coincidence, vowel recognition, probable word guessing, exhaustive dictionary search, human interaction, and several exotic techniques. Only a few of these are useful in our situation. To meet our constraints, we must eliminate any classical methods which require large samples of plaintext or lengthy calculations. This leaves single character, digraph, and trigraph frequency statistics as a natural starting point.

Diffie and Hellman proposed a very simple decision strategy: if the high bit of all 8 bytes in the plaintext block is zero, save the key; otherwise, discard it [2]. This strategy has the advantage of being simple and quick. Unfortunately, this test only works for 7 bit text; we would like to handle other types of data, if possible. Furthermore, frequency statistics are a generalization of this strategy, and it seems reasonable to hope they will achieve more powerful plaintext recognition capabilities.

The efficiency of any proposed plaintext recognizer can be measured with several quantities. First of all, it is important for a good strategy to minimize the number of mistakenly characterized trial decryption keys: the quantities

$$\rho = \Pr(K_t^{-1}[C] \text{ saved} \,|\, K_t \neq K)$$
$$\hat{\rho} = \Pr(K_t^{-1}[C] \text{ discarded} \,|\, K_t = K)$$

Table 1. Summary of notation

Name	Meaning
K	The true DES key
K_t	Trial DES key
C, C_i	Ciphertext block
$K_t^{-1}[C]$	The result from DES decrypting C with key K_t
P	Plaintext block, often $K_t^{-1}[C]$
P_i	i-th byte of P, for $1 \leq i \leq 8$
ρ	Prob. that an incorrect plaintext passes the plaintext recognizer
$\hat{\rho}$	Prob. that a correct plaintext fails the plaintext recognizer
k	Number of ciphertext blocks used
$w(x, y)$	Weight of the (x, y) digraph
$w(P)$	Total weight of P
b	Precision of $w(x, y)$, in bits
c	Scaling factor used to calculate $w(x, y)$
t	Threshold

should be as small as possible, where C is a ciphertext block, K the true DES decryption key, and K_t a trial key. (See Table 1 for a short list of notation.) Furthermore, a good strategy must be fast enough to keep pace with the DES pipeline, and yet simple enough to implement on a fairly inexpensive chip. Finally, a good strategy must be general enough to work with many different types of reference data—once the design is finalized and the chip fabrication begun, it becomes terribly difficult to adopt a new decision strategy.

The operation of our key search machine is slightly more complicated than for other designs. The attacker must possess several ciphertext blocks and the ability to compile a large collection of data similar to the enciphered plaintext. The cryptanalyst must analyze the reference data, calculate optimal values for all the input parameters, and initialize the key search machine with the ciphertexts and statistics. Then the exhaustive search can begin.

Once a possible key is encountered, the rest of message can be decrypted with it and the results presented to the operator or to some sophisticated statistical analyzer for confirmation. So long as not too many key candidates are selected for further analysis, these steps should not place too much of a burden on the attacker.

3 Algorithm Possibilities

A simple first stab at a decision strategy is to compare the candidate decryption's byte probabilities with those of a sample of similar plaintext.

We save the key K_t if the probabilities exceed some threshold and discard it otherwise. The threshold chosen should be small enough that the true encryption key K isn't likely to be discarded, yet large enough so that not too many false trial keys $K_t \neq K$ are saved.

On the chip, this may be implemented with 8 lookups into a table containing the probabilities for all 256 bytes followed by 7 multiplications. Unfortunately, multiplication is typically quite slow in hardware. One well-known improvement is to replace the product by the sum of their logarithms [14, 11]. Small values of the sum correspond to plaintexts which are similar to the reference data, and large values correspond to plaintexts which resemble white noise. If we precompute the logarithms of the probabilities and the threshold and store those instead of the raw values, each multiplication can be replaced by an addition. This change greatly speeds up the save/discard decision.

Generally, just one ciphertext block will not suffice to narrow down the number of possible encryption keys enough, but with multiple ciphertext blocks, much more powerful results can be achieved. Suppose we have a plaintext recognizer which is fairly good at characterizing the output of one trial decryption $K_t^{-1}[C]$: it filters out all but ρ of the incorrect trial keys $K_t \neq K$, and discards the true decryption key K with probability $\hat{\rho}$. Given k ciphertext blocks $\{C_1, \ldots, C_k\}$, the improved decision strategy specifies that a trial key K_t is to be saved if and only if every plaintext passes the plaintext recognizer. The new strategy will discard all but ρ^k of the incorrect trial keys and throw out the true decryption key with probability $1-(1-\hat{\rho})^k \approx k\hat{\rho}$. The machine need not decrypt all available ciphertext blocks. Unless $K_t^{-1}[C_1]$ passes the filter, there is no need to try the other ciphertext blocks. Since almost all of the trial keys can be discarded immediately after the first ciphertext block is decrypted, the key search rate will not decrease noticeably.

Another possible improvement is to try using digraph statistics instead of (or in conjunction with) single character frequencies. This should work at least as well as the single-character model, though it does require a larger memory array (256×256 entries instead of just 256).

We considered using trigraphs, but a more sophisticated data structure would be required; a straight-forward approach would need a $256 \times 256 \times 256$ array. Since our preliminary measurements did not show a significant improvement compared with digraph statistics, we have not pursued that approach any further.

We must now consider how to store the values $p(x, y)$ in memory for a hardware implementation. First of all, only a limited number of bits are available for each array element; let b denote the width of each element. To store the $p(x, y)$ values in a b bit array element, scale them linearly by a fixed constant $c > 0$. Truncate $w(x, y)$ so it is in the range $0 \leq w(x, y) \leq (2^b - 1)/2^b$.

There are several parameters that may be varied to maximize efficiency. First, we must fix the precision b at design time; it should be small enough that we don't waste too much chip space on a huge memory array, yet large enough that the logarithms have enough precision to discriminate between random noise and true plaintexts. The rest of the parameters can vary from reference set to reference set. Of these, the scaling factor c

has perhaps the greatest effect. To avoid waste and maximize efficiency, we should pick an appropriate value so that the $w(x, y)$ weights occupy much of the available range between 0 and 1. More importantly, this parameter controls the tradeoff between ρ and $\hat{\rho}$: if c is too large the correct decryption key has a good chance of being missed, yet if c is too small, too many false trial keys are saved. The threshold t also adjusts the sensitivity of the plaintext recognizer, but in a slightly different way. For convenience, call digraphs with maximum weight *killer digraphs*; that is,

$$w(x, y) \geq (2^b - 1)/2^b \qquad \rightarrow \qquad (x, y) \text{ a killer digraph.}$$

Killer digraphs are the rarest of the rare; by definition, they are pairs that should never occur in correct plaintext. Then the quantity t roughly indicates how harshly plaintexts are judged: a plaintext is rejected immediately if t killer digraphs are encountered in it. If the threshold t is too small, plaintexts with just one or two rare digraphs will be discarded instantly. That is probably undesirable; a little lenience in allowing for input errors or just the occasional unusual character is preferred.

Finally, the number of ciphertext blocks k is the last input parameter which affects the efficiency of the decision strategy. Large values increase the probability $(k\hat{\rho})$ of discarding the true decryption key, so k should be no larger than necessary. On the other hand, the more ciphertext blocks we use, the better the chances of discarding incorrect trial keys (this probability is approximately ρ^k). Since reference data with less regularity yields flatter digraph statistics and less information per ciphertext block, more ciphertext blocks may be needed with these types of sources. Because approximately $2^{56}\rho^k$ trial keys will have been saved by the time the key space is exhausted if attacking DES, a good rule of thumb is to choose k so that $\rho^k \approx 2^{-50}$ or so, when feasible. (By contrast, the original Diffie-Hellman solution has $\rho = 2^{-8}$, necessitating a k of at least 6. On the other hand, their $\hat{\rho} = 0$.) All these parameters must be set through empirical observations.

We must also decide what to do with digraphs (or single characters) that never appear in the reference data; their frequency of occurrence is zero, but the logarithm of zero is undefined. The obvious solution is to consider the logarithm of zero to be some tremendously large negative number, and proceed as usual. There is another problem, though. A zero which appears in the frequency count of the reference data is not necessarily a "hard" zero. In other words, the difference between a frequency count of 0, 1, or 2 is not likely to be statistically significant. The reference data is only a finite random sample of plaintext, and this induces random fluctuations in the observed frequency counts. Unfortunately, the logarithm function amplifies the difference between a frequency count of 0 and 1 to a large extent. The fix is to perturb the zero frequency counts a little, replacing them by 1 or some other small number [15, 16].

4 Modes of Operation

So far, we have only considered ECB mode, but in fact by taking advantage of the XOR mask, most of the other standard modes of operation can be attacked as well. Let F stand for the initialization vector (IV) if C is the first block deciphered, or else a feedback variable if C is not the first block. Cipher block chaining (CBC) mode is defined by the equations

$$\begin{aligned} P &\leftarrow K^{-1}[C] \oplus F \\ F &\leftarrow C. \end{aligned}$$

If the value of F (which is just the previous ciphertext block in almost all cases) is known, then it can be used in the XOR mask. Thus, CBC mode can be attacked just as easily as ECB.

The cipher feedback (CFB) and output feedback (OFB) modes are variable-length modes. The full 64 bit CFB mode is easy to understand: decryption is accomplished with the formula

$$\begin{aligned} P &\leftarrow K[F] \oplus C \\ F &\leftarrow C. \end{aligned}$$

Again, the value of F is usually just a ciphertext block and hence will almost certainly be known; so 64 bit CFB mode can be attacked as easily as ECB mode can. In general, the m-bit CFB mode transforms m-bit plaintext blocks into m-bit ciphertext blocks according to the decryption rules

$$\begin{aligned} P &\leftarrow \mathrm{left}_m(K[F]) \oplus C \\ \mathrm{left}_{64-m}(F) &\leftarrow \mathrm{right}_{64-m}(F) \\ \mathrm{right}_m(F) &\leftarrow C, \end{aligned}$$

where $\mathrm{left}_j(R)$ denotes the leftmost j bits of a 64 bit register R, and $\mathrm{right}_j(R)$ is defined similarly. Multiple encryptions with the same key are required to obtain a full 64 bits of plaintext in the m-bit CFB mode. Attacking the m-bit variable-length modes requires additional logic, because obtaining a full 8 byte plaintext takes $64/m$ decryptions. Wiener discussed this issue and concluded that adding support for the variable-length modes increases the cost of a key search machine by 10% or so. Furthermore, in our design attacking the m-bit modes increases the search time by a factor of at most $64/m$ (though some savings are possible if most plaintext blocks can be discarded after examining just a few digraphs).

Observe that ECB and CBC modes require the ability to decrypt known ciphertext, while CFB mode (as well as OFB mode, we shall see) needs encryption. No major changes to the DES pipeline are necessary; the augmentation requires very little extra logic because of the similarity between DES's key scheduling in encryption and decryption modes.

The OFB mode is the trickiest mode to attack. The 64 bit version satisfies

$$\begin{aligned} P &\leftarrow K[F] \oplus C \\ F &\leftarrow K[F]. \end{aligned}$$

If the IV is known, then our probable plaintext attack applies to this mode just as easily as any other; similarly, the m-bit OFB mode can be broken just like the m-bit CFB mode if the IV is known. Unlike the other modes, however, there appears to be no way to extend our ciphertext-only attack to OFB if the IV is unknown. Some care is needed, though; if any plaintext is known, the corresponding ciphertext can serve as the IV for the following ciphertext block.

Deliberately and carefully hiding the IV seems like a useful technique for achieving extra security with OFB mode. Since both parties must know the hidden IV, it essentially serves as 64 more bits of "key." However, the operational characteristics of OFB mode render it unsuitable for many applications [17].

5 Hardware Issues

Our design goal is to maintain a search rate of 50 million decryptions per second with a cost roughly comparable to that of Wiener's machine. At 64 bits of output per decryption, this I/O rate is near the upper limit of current technology; accordingly, it is important to implement the decision strategy and the DES pipeline on the same chip to keep inter-chip I/O to a minimum. Since 1.0 square cm is the largest size chip that can be comfortably fabricated with current technology, and Wiener's DES pipeline occupies approximately 0.4 square cm, we have about 0.6 square cm to implement a decision strategy. A digraph lookup table with 256×256 elements and 8 bits per digraph requires about 500 Kbits of RAM, whereas commercially available memories pack up to 4 Mbits of RAM on board one chip. We can therefore fit up to four copies of an 8-bit digraph lookup table on the chip. Using multiple copies of the table permits lookups to be done in parallel, which speeds up the decision algorithm. The additional registers, masks, adders, and miscellaneous logic are not too complex, and should fit easily in the remaining space.

The implementation of the decision strategy must also be very fast—it must handle 50 million blocks per second to avoid stalling the DES pipeline. The stratagems outlined so far involve table lookups, additions, masking, and comparisons. Certainly 64 bit logical ANDs, XORs, and comparisons with a register can be performed without difficulty at a clock rate rate of 50 MHz. Today's RISC CPUs can perform a 32 bit integer addition in one clock cycle, so a 8 or 10 bit addition should be no problem for our custom chip.

An alternative strategy would be to build a pipelined recognizer. This would allow for slower steps, such as multiplication. The main restriction is the recognizer must be able to sustain throughput at a rate at least as great as that of the decryption engine. Care must be taken to avoid contention for shared resources, such as the digraph table; a pipeline where several different stages needed to access the same memory would not be likely to yield an overall speedup.

Sometimes one trial key must pass through the DES decryption pipeline several times, and this ability requires a slight hardware modification of Wiener's basic design. First of all, we need this modification to take advantage of multiple ciphertext blocks. When the first ciphertext block decrypts under K_t to data which passes the plaintext filter, we must re-insert K_t into the DES pipeline with a new ciphertext block. Furthermore, if the RAM is too slow to calculate the weights of all the digraphs in one clock cycle, we will sometimes need to stall the pipeline momentarily: we must re-insert a trial key K_t if it cannot be rejected after looking up just 4 plaintext digraphs. The two applications have very similar requirements, so both can share the circuitry needed for key feedback. In Appendix E of his paper, Wiener discusses the logic needed to re-insert trial keys at the head of the DES pipeline; fortunately this addition increases his cost estimate by only a small amount. The time penalty is even smaller in most cases, as the proportion of trial keys that require feedback is usually miniscule; however, some types of plaintext do require more samples, as is discussed below.

If trial keys are recirculated, it is advantageous to recirculate the current block and killer digraph counters as well. Otherwise, increasing the number of blocks examined *increases* the probability of rejecting the correct key, since an improbable result from any single block could cause a rejection. We therefore must select not just a single t, but a collection of thresholds t_i.

Memory lookups could be more of a problem. Even the fastest RAM has a delay time of 5 nanoseconds or so, and we need to calculate up to seven digraph weights in 20 nanoseconds, the time for one 50 MHz clock cycle to pass. The easiest way to achieve this would be to make seven copies of the digraph weight array, and do the seven lookups in parallel; unfortunately, there is not enough space on the chip for this solution. Fortunately, most of the time we don't need to look up all of the digraph weights, as the sum of the first few weights is often already larger than the threshold value. One solution, then, is to store four copies of the digraph table (or as many as space permits) in 2 Mbits of 20 nanosecond RAM so that the first four digraph weights can be found and summed in one clock cycle. If those four are not sufficient to discard the key, it can be reinserted at the top of the pipeline with the partial sum saved; then after the key makes its way through the pipeline again, we may look up its last three digraph weights and add them to obtain a total weight sum. The slowdown inherent in this scheme will depend upon the reference data, and could decrease the search rate to 25 MHz in the worst case. If the slowdown is predicted to be too severe, another possibility is incorporate faster RAM and add a second clock, so that all seven digraph weights can be retrieved from memory in 20 nanoseconds, letting the pipeline run at full speed. On the other hand, this alternative does have the disadvantage of requiring multiple clock rates and faster memory, which may be problematic. Fortunately, most types of reference data (such as text) require on average very few digraph lookups before the incorrect trial keys are discarded.

The primary factor affecting the capital cost of the new design is the extra chip area; that in turn drives the yield down. There will also be some additional control ciruitry needed to load the digraph tables. For a chip of this design, doubling the area will increase the net costs by a factor of $2.5 - 3$. Since the cost of Wiener's design is dominated by the key search chips, our modification will at most triple the price.

The run time, and hence the cost per message, is also affected, since we need to analyze more ciphertext blocks. Again, the increase is roughly threefold. Naturally, one can trade off capital costs for time.

6 Software Implementation

We have not actually implemented this design. We have, however, built software simulations to predict how well it would work in practice. The results are encouraging—we learned which decision strategies were useful, how to choose input parameters, and which reference data could be efficiently recognized.

The experiments spanned a wide range of input parameters and types of reference data. Some common classes of text were tried: generic English, Shakespeare's collected works, C source code, Postscript, TEX files, and even articles from a USENET news group. We also experimented with some less conventional types of data, such as executables from different architectures, compressed files, and (as a control) DES ciphertext. Given a set of reference files and a decision strategy, the software simulation measured ρ and $\hat{\rho}$. It also calculated the percentage of blocks which required more than four digraph weight lookups. See Tables 2 and 3 in the Appendix for a brief summary of our measurements.

These trials suggest how to choose values for the many input parameters. First of all, note that a minor adjustment of the scaling factor c can greatly affect the number of characterization mistakes. This parameter is delicate enough that often it must be adjusted by hand; perhaps some clever software could successfully automate this process, though. Also, the value of b (the number of bits of precision used in the digraph weight table) has a definite effect on efficiency. For text, $b = 6$ bits is reasonable, and 8 bits does not yield much improvement; on the other hand, executables need 8 bits of precision. As increasing the precision up to a full 10 bits gains nothing, $b = 8$ appears to be a reasonable design choice. On the other hand, the threshold t (the absolute minimum number of digraphs examined before a block can be discarded) does not have as much of an effect: a value of 1.5 or so seems reasonable across the board. Perturbing the zero frequency counts seems to be a fairly useful tactic. Both ECB and CBC modes were tried. Also, the use of several ciphertext blocks to improve the recognition of incorrect trial keys helps tremendously, as expected, though naturally this increases the runtime and hence the cost per message.

Not all of the proposed strategies were helpful, though. First of all, single character frequency statistics failed to provide any useful information over and above the digraph statistics, even when both were used together. In addition, calculating the weights with a non-linear function gave very little benefit in our trials.

Our design successfully recognizes almost all of the types of plaintext we examined. Most types of text can be efficiently attacked: typically $\rho = 2^{-20}$ and $\hat{\rho} = 1/30$ for text, so we can expect to find the decryption key 90% of the time by using three ciphertext blocks. Moreover, binary executables also fall to our ciphertext-only key search machine so long as the attacker knows for which architecture the binaries were compiled:

$\rho = 2^{-15}$ and $\hat{\rho} = 1/12$ are representative probabilities, which means that with four ciphertext blocks the attacker has a 70% chance of discovering the decryption key. Unfortunately, the search rate slows by a factor of two if only four digraph weights can be looked up per DES decryption cycle, though there is no penalty with faster RAM. In short, our design indicates that encrypting low entropy data with single DES is not secure, for it can be broken in four hours with a machine that costs just a few million dollars.

We had no success against compressed data, suggesting that compression before encryption is a useful security measure. The suggestion has been advanced many times in the past, but we know of no previous trials. Since it also accelerates the encryption of lengthy plaintexts, pre-compression may be a useful tactic in in many situations.

Of course, switching to an encryption algorithm with a longer key length remains the best way to thwart our attack and increase security. Triple DES is a good choice: van Oorschot and Wiener estimate that triple DES with two keys is about 10^{13} times stronger than single DES, and the three-key version is even better [18]. IDEA is another possibility, though systems which use single DES are more easily converted to triple DES. The extra key bits needed to avoid brute force attacks could be agreed upon in advance, or one could generate a large session key via Diffie-Hellman exponential key exchange, which generally yields hundreds of bits of shared secrets.

We conclude that single DES is disturbingly vulnerable to an exhaustive key search machine designed for a statistical ciphertext-only attack. Fortunately, there are several high quality cryptosystems available which render our attack infeasible. Accordingly, users of single DES ought to consider strongly upgrading to a more secure encryption algorithm.

References

1. NBS. Data encryption standard, January 1977. Federal Information Processing Standards Publication 46.
2. Whitfield Diffie and Martin E. Hellman. Exhaustive cryptanalysis of the NBS data encryption standard. *Computer*, 10(6):74–84, June 1977.
3. Gilles Garon and Richard Outerbridge. DES Watch: An examination of the sufficiency of the data encryption standard for financial institution information security in the 1990's. *Cryptologia*, XV(3):177–193, July 1991.
4. Robert McLaughlin. Yet another machine to break DES. *Cryptologia*, XVI(2):136–144, April 1992.
5. Peter C. Wayner. Content-addressable search engines and DES-like systems. In *Advances in Cryptology: Proceedings of CRYPTO '92*, pages 575–586. Springer-Verlag, 1993.
6. Michael J. Wiener. Efficient DES key search. Technical Report TR-244, School of Computer Science, Carleton University, Ottawa, Canada, May 1994. Presented at the Rump Session of Crypto '93.

7. Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology: Eurocrypt '93*, pages 386–397. Springer-Verlag, 1994.
8. Eli Biham and Adi Shamir. Differential cryptanalysis of the full 16-round DES. In *Advances in Cryptology: Proceedings of CRYPTO '92*, pages 487–496. Springer-Verlag, 1993.
9. Solomon Kullback. *Statistical Methods in Cryptanalysis*. Aegean Park Press, P.O. Box 2837, Laguna Hills, California 92653, 1976.
10. John M. Carroll and Lynda Robbins. The automated cryptanalysis of polyalphabetic ciphers. *Cryptologia*, XI(4):193–205, 1987.
11. Robert W. Baldwin and Alan T. Sherman. How we solved the $100,000 decipher puzzle (16 hours too late). *Cryptologia*, XIV(3):248–284, July 1990.
12. David Applegate and Guy Jacobson. Solving simple substitution ciphers by exhaustive dictionary search, November 1993.
13. Bruce R. Schatz. Automated analysis of cryptograms. *Cryptologia*, 1(2):116–142, April 1977.
14. Abraham Sinkov. *Elementary Cryptanalysis: A Mathematical Approach*. Random House, New York, 1968.
15. Jim Reeds, July 1994. Private conversation.
16. Caxton C. Foster. *Cryptanalysis for Microcomputers*. Hayden Book Co., Inc., Rochelle Park, NJ, 1982.
17. V. L. Voydock and S. T. Kent. Security mechanisms in high-level network protocols. *ACM Computing Surveys*, 15(2):135–171, June 1983.
18. P.C. van Oorschot and M.J. Wiener. A known-plaintext attack on two-key triple encryption. In *Advances in Cryptology: Proceedings of CRYPTO '90*, pages 318–325. Springer-Verlag, 1991.

A Summary of Simulation Results

Table 2. Performance with different reference data sets. Measurements are for the optimal value of the scaling factor c; here $b = 8$ bits, $t = 1.5$, and perturbation is used. Also, the Pr(pipeline stall) column refers to the extra time for chips equipped with slow RAM that can only look up 4 digraph weights per 20 nanosecond cycle time; chips with faster memory suffer no speed decrease.

Reference data	ρ	$\hat{\rho}$	Pr(pipeline stall)
Shakespeare	2.5×10^{-7}	0.021	0.1%
Generic English	7.5×10^{-7}	0.024	0.2%
PostScript	2.5×10^{-7}	0.047	0.2%
$\TeX$ documents	1.1×10^{-6}	0.035	0.4%
C programs	5.2×10^{-6}	0.043	0.8%
rec.art.poems	1.1×10^{-6}	0.043	0.5%
sci.crypt	2.3×10^{-6}	0.033	0.8%
SunOS executables	3.1×10^{-5}	0.081	99%
Compressed files	0.52	0.27	100%

Table 3. Performance with different input parameters

Reference data	b	t	Perturb	ρ	$\hat{\rho}$
Shakespeare	6	1.5	Yes	2.5×10^{-7}	0.021
Shakespeare	8	1.5	Yes	2.5×10^{-7}	0.021
Shakespeare	10	1.5	Yes	2.5×10^{-7}	0.021
Shakespeare	8	1.0	Yes	2.5×10^{-7}	0.022
Shakespeare	8	2.0	Yes	2.5×10^{-7}	0.019
Shakespeare	8	1.5	No	1.9×10^{-7}	0.026
SunOS executables	6	1.5	Yes	7.0×10^{-3}	0.051
SunOS executables	8	1.5	Yes	3.1×10^{-5}	0.081
SunOS executables	10	1.5	Yes	2.6×10^{-5}	0.083
SunOS executables	8	1.0	Yes	4.5×10^{-5}	0.068
SunOS executables	8	2.0	Yes	3.0×10^{-5}	0.081
SunOS executables	8	1.5	No	2.0×10^{-5}	0.130