A LIMITED-DIFFUSION ALGORITHM FOR BLIND SUBSTRING SEARCH

Matthew Skala
University of Victoria
Victoria, BC
(250) 472-7534
mskala@ansuz.sooke.bc.ca

Abstract

Applications are described for "blind substring search," where a program to search files for a substring is published without revealing the substring. The "limited diffusion" approach proposed in a previous article is described. Design criteria for a Boolean function to be used in the limited diffusion algorithm are stated, and a function meeting the criteria is proposed. An algorithm for blind substring search is developed and discussed.

A LIMITED-DIFFUSION ALGORITHM FOR BLIND SUBSTRING SEARCH

Introduction

Consider a binary string **S**, which may or may not contain a given substring **s**. A method will be developed for publishing a function that tests **S** for the presence of **s**, while cryptographically protecting **s** from attackers who reverse engineer the test function. This problem and some approaches to solving it were proposed in an article posted to Usenet by the author [6]; the present paper describes a detailed solution in more formal terms, with theoretical background.

One obvious application for this technique would be to "parental control" software for the Internet. Commercial products exist that search network traffic and block files containing substrings considered characteristic of material unsuitable for children. Design of such software is complicated by the fact that the manufacturer may want to keep the list of blocked substrings secret from competitors, political groups, and intelligent children.

Suppose Alice is the president of Cyphersitter, Inc. (Motto: "We'll protect you from knowing what our motto is.") She wants to sell Bob a program that will keep an eye on Junior's web-surfing activities and keep him from getting to certain material. But Junior is able to reverse-engineer the program. Encrypting the block list with any conventional encryption algorithm, no matter how strong, will not work. The software must decrypt the file to use it, and at that point Junior can interrupt the program with a debugger and read out the plaintext.

Considerable resources may be brought to bear on reverse-engineering a blocking program, and manufacturers of blocking software have proven to be incautious about security, to their great detriment. One major product concealed its block list by XORing every byte with the 8-bit constant 94_{16}; the manufacturer was publicly embarrassed when this fact was reported to the community by an 18-year-old youth rights activist, causing a high-profile legal dispute [7]. Clearly, such techniques are inadequate.

There are other applications for a blind substring search. Consider the predicament of an intelligence agency which has the opportunity to place a "sniffer" device on an enemy network. They want to search email messages for any mention of their agents' secret code names, but in the event that the sniffer device falls into enemy hands, it should not be possible to determine the code names by examining the device. Conventional tamper-proof hardware may be insufficient; it would be better to use a cryptographic technique.

Blind substring search could also find applications in computational biology. Nucleic acid molecules, as nature's mass storage medium, can be viewed as long bit strings describing the structure of proteins. Computational biology problems involve searching for substrings in gene sequence databases. Since those databases are compiled at tremendous effort and expense, intellectual property rights issues become significant. If Alice and Bob have each sequenced different genes, blind substring searching could allow them to decide whether their genes contain related substrings, without revealing their secret sequence databases to each other except where the databases are found to overlap.

Securely hashing each possible substring may be useful in some of these situations, but it is not a generally satisfactory approach because the number of possible substrings increases quadratically with the size of the input. It would be preferable to process the input once and then do some simple operation such as an ordinary substring search.

It should be noted that the protection provided by any blind substring search is strictly limited. If an attacker can find, by guessing, a string for which the test returns "match found," then the attacker can

test variations of the input and easily find the substring that caused the match. At best, we can make all possible guesses appear equally likely to match, so that an attacker cannot rule out categories of guesses, and must try them all to break the system.

Limited diffusion approach

One approach would be to process the input and the substrings through a "limited diffusion" function [6]. The limited diffusion function is like a single generation of a cellular automaton with a large "neighbourhood": output bits correspond to overlapping ranges of consecutive bits in the input, and each output bit is the same complicated function of the bits in its range.

Each output bit is affected only by the bits in its own neighbourhood. Conversely, each input bit only affects output bits within that range. If the input to a limited diffusion function with an **n**-bit neighbourhood contains a given substring **s** of length **m**, **m**≥**n**, then there is a substring **s'** of length **m**-**n**+1 that will always be found in the output. The blinded substring search can consist of applying the limited diffusion function to the input string and then testing the result for the presence of **s'**. It should be difficult to determine **s** from **s'**.

Security demands that the limited diffusion function behave in a "random," difficult to analyse fashion. Then since **s'** is shorter than **s** there will generally be other **m**-bit substrings as well as **s** that also produce **s'** in the output. On average, 2^{n-1} **m**-bit strings including **s** will produce **s'** in the output. If **n** is large, there is a risk that the input will contain one of those strings but not **s**, thereby triggering a false match. However, if **n** is small, and especially if the attacker can create and examine a list of all the 2^{n-1} possibilities, then the security of **s** will fail. Ideally, the neighbourhood size **n** should be adjustable to suit different applications.

If a meaningful input string **s** of length **m** contains **r** bits of redundant information, there is 2^{-r} probability that a randomly chosen string of length **m** will also appear meaningful. If **s** produces **s'** in the output of a limited diffusion function with an **n**-bit neighbourhood, the number of **m**-bit input strings that will produce **s'** in the output is approximately 2^{n-1}; to prevent unintended meaningful inputs from triggering matches, **n** should be somewhat smaller than **r**. This places an upper limit on the useful neighbourhood size, depending upon the entropy of the input and the lengths of the substrings to be tested. Compressing the input, a technique previously suggested to deter brute-force attacks [6], could prove to be a liability if it reduces the redundancy to the point where false matches become a problem.

A simple limited diffusion function would consist of a 9*1 S-box, i.e., a function $\mathbf{S}:\{0,1\}^9 \rightarrow \{0,1\}$, operating on all the 9-bit substrings in the input. The neighbourhood size can be increased in increments of 8 bits by applying the function repeatedly. Assuming we have **m** plaintext bits $\mathbf{b}_{0,j}$, $0 \le \mathbf{j} < \mathbf{m}$, define:

$$(1)\quad b_{i,j} = S(b_{i-1,j+8}, b_{i-1,j+7}, \ldots, b_{i-1,j}) \quad i \ge 1,\ 0 \le j < m - 8i$$

The output of the limited diffusion function is in the bits $\mathbf{b}_{r,j}$ where **r** is the number of rounds chosen for the desired security. The length of the output decreases by 8 bits for each round. This algorithm has no key, and inclusion of one would be pointless because it is assumed that the attacker can reverse-engineer the system. Nonetheless, the number 2^{8r}, representing the approximate number of input strings that produce a given output string, seems analogous to the size of the keyspace.

Note that for consistency with the usual bit-numbering convention of computer programming, we identify the most significant bits with the highest indices, even though they are usually written on the left. For the purposes of the algorithms in this paper, the first byte of a multi-byte string contains bits 0 to 7 with bit 0 corresponding to the integer value 1 and bit 7 corresponding to 128. The second byte contains bits 8 to 15, and so on, regardless of the religion of the underlying computer hardware.

S-box design

Much has been written about design criteria for S-boxes. As a first criterion, the S-box used in the limited diffusion function should be "balanced"; i.e., **S** should be such that $|\{\mathbf{x}|\mathbf{S}(\mathbf{x})=0\}| = |\{\mathbf{y}|\mathbf{S}(\mathbf{y})=1\}|$. In the limited diffusion function, this criterion is especially important, because the round output is determined entirely by the S-box output. If the S-box output were not equally likely to be a 1 or a 0, then the output stream would be biased, harming the entropy of the output. Nonlinearity is also desirable, because if any linear approximations of the S-box exist, they may be useful in differential and linear cryptanalysis. Although the nature of this application renders many standard cryptanalytic attacks irrelevant, linearity in the S-box could make it easier to reverse the round function.

A function $\mathbf{f}:\{0,1\}^n \rightarrow \{0,1\}$ is called "perfectly nonlinear" if $\mathbf{g}(\mathbf{x}) = \mathbf{f}(\mathbf{x}) \oplus \mathbf{f}(\mathbf{x} \oplus \mathbf{a})$ is a balanced function for all nonzero **a**, and perfectly nonlinear functions are equivalent to "bent functions" [4]. Several kinds of "propagation criteria" may be defined as subsets of perfect nonlinearity, for functions that are nonlinear with respect to some but not all values of **a** [2]. In simple terms, perfect nonlinearity says: if the attacker complements some combination of input bits, the output has an even chance of changing. Unfortunately, a perfectly nonlinear function cannot be balanced [2]; neither can its input size **n** be odd [5].

Other criteria for Boolean functions include the Strict Avalanche Criterion (SAC) and correlation immunity, both of which are explored in detail by Lloyd [3]. The SAC in its zeroth order form as defined by Webster and Tavares [8] requires that changing one input bit of the S-box should have an even chance of changing the output. Forré [1] describes Strict Avalanche Criteria of order $\mathbf{r}>0$, which demand that the SAC continue to hold for the "sub-functions" formed by holding some number **r** of the input bits constant. Correlation immunity of degree **r** requires that for each set of **r** input bits, the 2^r possible values all appear equally likely when the output of the function is known [2, 3].

Lloyd [3] describes a method for designing S-boxes that satisfy combinations of these criteria. An S-box function $\mathbf{S}:\{0,1\}^n \rightarrow \{0,1\}$ can be expressed uniquely in "algebraic normal form" as an exclusive-or sum of products:

$$(2)\quad S(x_{n-1},x_{n-2},\ldots,x_0) = \bigoplus_{U\subseteq Z_n} \left(a_U \prod_{i\in U} x_i \right)$$

The function is designed by choosing values for the coefficients $\mathbf{a}_U$. All $\mathbf{a}_U$ are set to 0 for $|\mathbf{U}| \geq 3$. Zero or more disjoint ordered pairs of input bits $(\mathbf{x},\mathbf{y})$ are chosen; within each pair, $a_{\{x,y\}}=0$, $\mathbf{a}_{\{x\}}=1$, $\mathbf{a}_{\{y\}}=0$. For any $\{\mathbf{a},\mathbf{b}\}$ other than those pairs, $\mathbf{a}_{\{a,b\}}=1$. Finally, $\mathbf{a}_{\{x\}}$ may be chosen arbitrarily for any unpaired input bits **x**, and $\mathbf{a}_{\varnothing}$ may also be chosen arbitrarily. An **n**-input function designed by this procedure, with two or more pairs, will satisfy the SAC of order **n**-3 and be balanced and correlation immune of at least degree 1 [3]. More than two pairs will lead to greater degrees of correlation immunity.

While designing the 9-input Boolean function for use in the limited diffusion algorithm, each 9-input function that could be designed by this procedure was assigned a name consisting of three groups of decimal digits separated by hyphens, as "012-678-23". The first group represents, in ascending order, the paired bits **x** for which $\mathbf{a}_{\{x\}}=1$. The second group represents the corresponding bits **y** for which $\mathbf{a}_{\{y\}}=0$. The last group of digits represents, as a decimal integer, the binary bit string that has 1 in those positions for which the corresponding bit **x** has $\mathbf{a}_{\{x\}}=1$, including both paired and unpaired bits.

In the example of "012-678-23", the pairs are (0,6), (1,7), and (2,8). In addition to the values forced by those pairings, $\mathbf{a}_{\{4\}}=1$, and $\mathbf{a}_{\{i\}}=0$ where not otherwise specified. This notation does not deal with $\mathbf{a}_{\varnothing}$ because its value has no effect on the relevant properties of the S-box. Note that this notation was chosen for convenience in computer programming, not for any particular merit in human communication. It is not a generally useful nomenclature of Boolean functions.

Since balance and first-order correlation immunity are desirable, only functions with two or more pairs were considered. A list of all two-, three-, and four-pair 9-input functions was generated, and the resulting functions tested for nonlinearity. It was found that for all the two- and three-pair functions, $\mathbf{g}(\mathbf{x}) = \mathbf{f}(\mathbf{x}) \oplus \mathbf{f}(\mathbf{x} \oplus \mathbf{a})$ was balanced for all but 8 values of $\mathbf{a}$ including zero, which will be called "unsafe vectors." XOR addition of an unsafe vector to the input of the function was found to always or never change the output. Each set of 8 unsafe vectors consisted of the linear combinations of 3 linearly independent values. The four-pair functions were found to have 32 unsafe vectors each, apparently representing the linear combinations of five values.

It was decided that 32 unsafe input changes were too many, so further searches were confined to the two- and three-pair functions. Further, the S-box should not be symmetric with respect to reversing the order of the input bits. In other words, $\mathbf{S}(\mathbf{x}_8,\mathbf{x}_7,...,\mathbf{x}_0)$ should be uncorrelated with $\mathbf{S}(\mathbf{x}_0,\mathbf{x}_1,...,\mathbf{x}_8)$. All functions failing this test were disregarded. A symmetric S-box, in addition to seeming "less random," has the property of preserving palindromes in the input, which might be undesirable. Functions which had any unsafe vectors containing a 1 in the highest or lowest bit position were eliminated for reasons to be explained below.

The "shifty" way in which the limited diffusion algorithm uses its S-box gives rise to another criterion. If an attacker can make a change to the input with predictable effects on the output, that change should at least not also have predictable effects on nearby instances of the S-box, whose inputs overlap. For example, since bit 4 of each box is the same as bit 3 of the box on one side and bit 5 of the box on the other side, the S-box should if possible not be weak in more than one of those bits.

To quantify this "shift immunity" desire a scoring method was introduced. For each ordered pair of the 8 unsafe vectors of an S-box, the number of shifts was counted to render the overlapping portions of the values identical. The sum of those counts became the score of the function. To describe this more formally, consider the bit vectors as binary integers and define:

$$\delta(a,b) = \textit{smallest } d \geq 0 \textit{ such that } \left\lfloor \frac{b}{2^d} \right\rfloor \equiv a \pmod{2^{9-d}}$$

$$\sigma(S) = \sum_{\substack{a,b \text{ unsafe} \\ \text{vectors for } S}} \delta(a,b) \tag{3}$$

Note that pairs are counted in both directions, both $\delta(\mathbf{x},\mathbf{y})$ and $\delta(\mathbf{y},\mathbf{x})$, because they may be significantly different, as $\delta(000000000_2,000000001_2)=1$ and $\delta(000000001_2,000000000_2)=9$. Checking a vector against itself makes no difference to the total score, because $\delta(\mathbf{x},\mathbf{x})=0$.

Table 1: 9*1 S-box names with maximal $\sigma(\mathbf{S})$, excluding reflections

123-465-14	123-465-143	123-465-270	123-465-399	125-463-167	125-463-294
125-463-38	125-463-423	136-452-203	136-452-330	136-452-459	136-452-74
156-432-227	156-432-354	156-432-483	156-432-98	234-651-157	234-651-28
234-651-284	234-651-413	245-613-181	245-613-308	245-613-437	245-613-52
346-512-217	346-512-344	346-512-473	346-512-88	456-132-112	456-132-241
456-132-368	456-132-497				

The greatest $\sigma(\mathbf{S})$ found among the eligible functions was 407, for 64 different function names, equivalent to 128 possible functions because of the neglected $\mathbf{a}_\varnothing$. Half those names are displayed in Table 1; the other 32 represent the same functions with the bits in the opposite order. The functions in Table 1 all have the same unsafe vectors: 000010010_2, 000101000_2, 001000100_2, and linear combinations of these.

One of these function names, and a value for $\mathbf{a}_{\varnothing}$, may be chosen arbitrarily. For the purposes of standardization, the function "456-132-241" with $\mathbf{a}_{\varnothing}$=0 is suggested. It is displayed in Table 2. Entries in the table are to be read left to right and top to bottom, and correspond to the input vectors 000000000_2, 000000001_2, ... , 111111111_2 in binary counting order. This function was chosen by visually inspecting the function tables and choosing one that "looked random"; with respect to the formal design criteria, the 128 functions with maximal $\sigma(\mathbf{S})$ should all be equally secure.

Table 2: The 9*1 S-box "456-132-241", $\mathbf{a}_{\varnothing}$=0

0	1	0	0	0	0	1	0	0	0	1	0	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	1	0	0	0	1
1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	1	1	0	0	0	1	1	1	0	0	0	0	1	1	0	0	0
1	1	0	1	1	0	1	1	0	1	0	0	1	1	0	1	1	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0
1	0	1	1	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	1	0	1	1	1	0	1	1	1	1	1	1	0
1	1	0	1	0	1	0	0	0	1	0	0	0	0	1	0	1	0	0	0	1	1	1	0	1	1	1	1	0	0	1	1
1	0	1	1	1	1	0	1	0	0	1	0	1	0	1	1	0	0	0	1	1	0	0	0	0	1	1	1	0	0	0	1
1	0	1	1	0	0	1	0	1	1	0	1	1	0	1	1	0	0	0	1	0	1	1	1	1	0	0	0	0	0	0	1
0	0	1	0	0	1	0	0	0	1	0	0	1	1	0	1	0	1	1	1	1	1	1	0	1	1	1	0	1	0	0	0
0	0	1	0	1	0	1	1	1	0	1	1	1	1	0	1	0	1	1	1	0	0	0	1	0	0	0	1	1	0	0	0
0	1	0	0	0	0	1	0	1	1	0	1	0	1	0	0	1	1	1	0	0	1	1	1	1	0	0	0	1	1	1	0
0	1	0	0	1	1	0	1	0	0	1	0	0	1	0	0	1	1	1	0	1	0	0	0	0	1	1	1	1	1	1	0
1	1	0	1	1	0	1	1	1	0	1	1	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1	1	1
0	1	0	0	0	0	1	0	0	0	1	0	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	1	0	0	0	1
1	1	0	1	0	1	0	0	1	0	1	1	1	1	0	1	1	0	0	0	1	1	1	0	0	0	0	1	1	0	0	0
1	1	0	1	1	0	1	1	0	1	0	0	1	1	0	1	1	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0
1	0	1	1	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	1	1	1	0	1	1	1	1	1	0

Here are some desirable properties exhibited by S-box function "456-132-241":

· Balance

· Strict Avalanche Criterion of order 6

· Correlation immunity of degree 2

· Asymmetry

· The four transformations formed by shifting the inputs one bit in either direction and replacing the deleted bit with an 0 or a 1 each have an even chance of changing the output. This is believed to be a consequence of other properties; it means that successive output bits are uncorrelated if successive input bits are uncorrelated.

· Strong nonlinearity

· No unsafe vectors with 1 in the highest or lowest bit position

· $\sigma(\mathbf{S})$ = 407, indicating that linear weaknesses in one output bit should not easily affect nearby output bits

End effects

When more than one round is used, more problems appear. Consider one bit $\mathbf{b}_{2,j}$ in the output of the second round of (1). This bit depends upon the nine bits $\mathbf{b}_{1,j+8}...\mathbf{b}_{1,j}$ in the first round output, and they in turn depend upon the 17 plaintext bits $\mathbf{b}_{0,j+16}...\mathbf{b}_{0,j}$. Plaintext bits near the centre of this range will each affect several bits in the first round and therefore have good avalanche characteristics, but plaintext bits near the end of the range will affect very few first-round bits.

A change to plaintext bit $\mathbf{b}_{0,j}$ may have no effect, or it may change bit $\mathbf{b}_{1,j}$ in the first round. Those are the only possibilities. The bit $\mathbf{b}_{1,j}$ may or may not have any effect on the second-round output $\mathbf{b}_{2,j}$. If

the S-box satisfies the Strict Avalanche Criterion, then each one-bit input change only has 1/2 probability of changing the output of the round, and so a change to $\mathbf{b}_{0,j}$ only has 1/4 probability of changing $\mathbf{b}_{2,j}$. After several rounds, the plaintext bits on the ends very quickly become useless.

To preserve the information from the edges of the neighbourhood, we can modify the definition of the limited diffusion function to incorporate the neglected plaintext bits. If we have **m** plaintext bits $\mathbf{b}_{0,m-1}$ $\mathbf{b}_{0,m-2}...\mathbf{b}_{0,0}$ and a 9*1 S-box **S**, we can define:

$$
\begin{aligned}
& b_{i,j} = S(b_{0,j+8}, b_{0,j+7}, \ldots, b_{0,j}) && 0 \le j < m-8 \\
& x_{i,j} = b_{0,j-1} \oplus b_{i-1,j} \oplus b_{0,j+8i+1} && i \ge 1,\ 1 \le j < m-8i-1 \\
& b_{i,j} = S\begin{pmatrix} (b_{i-1,j+8} \oplus b_{0,j} \oplus b_{0,j+1} \oplus \ldots \oplus b_{0,j+7}), \\ x_{i-1,j+7}, x_{i-1,j+6}, \ldots, x_{i-1,j+1}, \\ (b_{i-1,j} \oplus b_{0,j+8i-7} \oplus b_{0,j+8i-6} \oplus \ldots \oplus b_{0,j+8i}) \end{pmatrix} && i \ge 1,\ 0 \le j < m-8i
\end{aligned}
\qquad (4)
$$

The output of the limited diffusion function is $\mathbf{b}_{r,j}$ with **r** the number of rounds selected for security and performance. Now, changing any odd number of bits in either of the eight-bit ranges at the start or end of the plaintext neighbourhood $\mathbf{b}_{0,j+8i}...\mathbf{b}_{0,j}$ will always affect the high or low bit of the S-box input. One of the design criteria for the S-box guaranteed that any change including one of the end bits would have an even chance of changing the output, so toggling odd numbers of bits in an end range always has 1/2 probability of changing the output.

If an even number of bits are changed in an end range, the end bit of the S-box input will be unchanged and there is some slight risk that the remaining bit changes will not have an even chance of changing the output. The worst case occurs when an even number of bits are toggled, of which the most significant is bit $\mathbf{b}_{0,j+3}$. Assuming that the previous round input is random and evenly distributed, such a change has 1/16 probability of producing the change 000010010_2 in the input to the last round S-box, and that is one of the unsafe vectors.

In this case the output will always change and it has an even chance of changing in the other cases, so toggling an even number of input bits of which the most significant is bit $\mathbf{b}_{0,j+3}$ has 17/32 probability of changing the output. Similar but less probable linearities may be found when even numbers of bits are toggled in other positions. No use for this linearity is immediately apparent, but it could render the algorithm susceptible to some form of linear attack. A less linear method for preserving information from the ends of the input range could improve security.

Incorporating plaintext bits in this way helps foil round-by-round cryptanalysis. An algorithm similar to (1) has been attacked [6] by computing a list of the approximately 2^8 possible previous inputs to the last round and then examining each of those recursively. The time complexity of this attack is 2^{8r} for **r** rounds, but the space complexity is only $(2^8)\mathbf{r}$ because the list of 2^{8r} inputs need not be generated all at once.

Short substrings necessitate few rounds, rendering this attack threatening. Adding information from previous rounds makes the function look like one large round, rendering the attack less effective: the 2^8 possible input strings to the last round each correspond to many different output strings from the second-to-last round, depending on the plaintext bits, which also influence the inputs and outputs of other rounds. Note that for this reason, it is much better to run the algorithm once with $2\mathbf{r}$ rounds than twice with **r** rounds. It is entirely possible that there are exploitable weaknesses in the multi-round function (4), but finding and exploiting them will be more difficult than the simple recursive attack.

One other attack could stem from the fact that the algorithm works both ways. If **s** contains more than twice as much redundancy as that eliminated by the limited diffusion function, then **s'** will be long

enough to itself contain testable substrings. An attacker can guess possible substrings of **s**, test **s'** for their presence, and thereby obtain clues to the contents of **s**. This problem may be avoided by using more rounds of diffusion, or by splitting **s** into chunks and searching for those chunks occurring together. Both approaches discard additional information, to make the diffused substrings shorter and less likely to themselves contain testable substrings.

Algorithm description

To describe the limited diffusion function concisely in psuedocode, assume the existence of a SUBSTR($\mathbf{b}_i$,**x**,**y**) subroutine which returns the **y**-bit substring of the bit string $\mathbf{b}_i$, starting at bit **x**, and a PARITY(**x**) subroutine which returns the XOR sum of all the bits in **x**. Assume the S-box subroutine S(**a**) returns the value of entry **a** from the S-box in Table 2. Then the limited diffusion function can be calculated by this psuedocode:

Put **m**-bit string **s** into $\mathbf{b}_{0,0}...\mathbf{b}_{0,m-1}$
For **i** ← 1 to **r** (number of rounds)
    If **i** == 1
        For **j** ← 0 to **m**-9
            $\mathbf{b}_{i,j}$ ← S(SUBSTR($\mathbf{b}_0$,**j**,9))
    Else
        For **j** ← 0 to **m**-8**i**-1
            $\mathbf{b}_{i,j}$ ← S(SUBSTR($\mathbf{b}_0$,**j**,8)<<1 ⊕ SUBSTR($\mathbf{b}_{i-1}$,**j**,9) ⊕ SUBSTR($\mathbf{b}_0$,**j**+8**i**-7,8) ⊕
                PARITY(SUBSTR($\mathbf{b}_0$,**j**+8**i**–6,7)) ⊕ PARITY(SUBSTR($\mathbf{b}_0$,**j**+1,7))<<8)
    End if **i** == 1
End for **i**
Return $\mathbf{b}_{r,0}...\mathbf{b}_{r,m-8r-1}$ as (**m**-8**r**)-bit string **s'**

A simple blind substring search, suitable for the "Cyphersitter" and "intelligence agency" examples, would involve running this algorithm on each of the list of substrings to be matched. Then the system installed in the insecure location would run the algorithm on the input, and search the result for any of the encrypted substrings using a conventional substring matching algorithm.

In the computational biology example, where Alice and Bob wish to compare secret databases, they can each run the limited diffusion function on their own databases, share the results, and perform whatever comparisons they desire. The genetic code contains a great deal of redundant error-checking information, so of a list of many random bit strings, very few will be valid sequences from genes. Therefore, if Alice and Bob find a shared sequence of length **m**-8**r** in both their diffused databases, it indicates a probable shared sequence of length **m** in the original databases. They may then decide whether to reveal to each other those parts of the original databases (or perhaps only secure hashes of those parts) to confirm the match. Shared sequences of length ≤8**r** will go undetected.

Because of the redundancy of the genetic code, the simple algorithm for blind database comparison described above should not be used without further analysis. Depending on the number of rounds, it may be possible for Bob to create in advance a codebook of all probable substrings, and then use those along with the information Alice reveals and any matches he does find, to guess at Alice's original database. This problem might be solvable if Bob and Alice use a coin-toss protocol to agree on a highly redundant code neither of them can predict or control. Then they translate their databases into that code before processing them, and because of the increased redundancy, they can use more rounds and effectively prevent precalculation.

A full investigation of the problems involved in gene sequence comparison would be beyond the scope of this paper; for now, the limited diffusion algorithm is only suggested for simple "found or not found" substring searching, where it is not necessary to give an attacker part of the substring while keeping the rest secret.

Verification

Some test vectors are provided in Table 3, to verify correctness of implementations of this algorithm. Note that the numbers in the table are hexadecimal integers written in the usual fashion with the most significant, highest numbered bits first, and spaces inserted every four digits for readability. Two input vectors are shown, with the resulting output after one, two, four, and eight rounds. Table 4 may be of interest to attackers of the algorithm: it contains four similar vectors, each resulting from a different secret input. All the secret inputs are English sentences in plain ASCII code.

Table 3: Test vectors for the limited diffusion algorithm

Input	0123 4567 89AB CDEF F0E1 D2C3 B4A5 9687 7869 5A4B 3C2D 1E0F
Round 1	1091 2FAE BF67 C42C 4082 AB1F 3D53 F43A 3E79 80BD 07EC 9B
Round 2	AFD6 1966 B794 4187 37B8 BC1B B49D CFDC D3EF A4BE 5131
Round 4	6120 ADA4 4BA8 2E5A 2CF0 3D95 EA63 98B6 36F6 22AE
Round 8	7C25 9278 F9A8 6183 09CF 19DF EF9D E2BB
Input	6D68 7469 726F 676C 6120 6E6F 6973 7566 6669 6420 6465 7469 6D69 4C ("Limited diffusion algorithm")
Round 1	7F40 2246 5A35 9ABA 92D4 0E3E 4670 2FDD D455 E7DC D07E 2258 7E78
Round 2	3F18 F4DE E9FC B18D 1E70 DC60 A6BF 7AA8 FD75 E654 A750 E40B D5
Round 4	3684 DA9E 5E4F 89C8 BBDF 49B8 5EB7 D75F A30D 0068 C561 27
Round 8	C822 AB73 C5C1 42CE D0B1 4ACF CE86 E5CD 07CE B0

Table 4: Sample diffused vectors for attack

1 round	7220 4259 DC67 4067 D6E5 B7BC 29DC 65E7 BB
2 rounds	E9F8 258B B575 DD39 8D14 AEF6 43AB 8E19 E0ED BC7F 261D
4 rounds	007B 8590 4719 9C00 681E 7123 458E 161C 324F CF36 DDE5 B4F7 9321
8 rounds	7340 367E E198 C343 7B35 B5A9 5568 F60D 9A6C 1DF2 7491 C792 6F13 A78B A5C2 FE5F BEEA 5D0E 3D

As a "sanity check" to determine whether the multi-round application of the S-box created unforseen linearity holes, the 8-round version of the limited diffusion function was empirically tested by a method similar to that of Webster and Tavares [8]. For each of the 65 input bits, 10^5 random pairs of 65-bit inputs differing only in that bit position were generated. The number of pairs which produced differing outputs was counted, and the results are shown in Table 5. The mean number of output changes is 50001.48 and the standard deviation is 142.2. These data seem to indicate that the multi-round function has good avalanche behaviour, at least with respect to single-bit input changes. Despite the linearity noted earlier, it should provide good security in its intended use.

Table 5: Results of 10^5 tests on each input bit of the 8-round function

	0	1	2	3	4	5	6	7	8	9
0	49900	49990	50131	50105	49773	49991	49891	50221	49994	49902
10	50247	50068	49905	49931	49995	50018	50251	50181	49791	50018
20	49736	50108	50219	50002	49926	50101	49828	50023	49986	49863
30	49954	50007	50236	49923	50069	49884	49994	50136	49769	50140
40	50323	49663	50130	49856	50050	49699	49861	49898	49883	50088
50	49992	50098	50144	50089	49976	50168	50061	49905	50004	50085
60	50169	50001	49990	49906	49850					

Acknowledgements

The author wishes to thank the members of Peacefire and of Americans for a Society Free from Age Restrictions, for their support of young computer professionals and their pioneering work in reverse engineering of blocking software, which inspired this work.

References

1. Forré, R. (1989). The Strict Avalanche Criterion: spectral properties of Boolean functions and an extended definition. *Advances in Cryptology - CRYPTO '88 Proceedings*, 450-468. Berlin: Springer-Verlag. Edited by S. Goldwasser.

2. Hirose, S., & Ikeda, K. (1994). Nonlinearity criteria of Boolean functions. *KUIS Technical Report KUIS-94-0002* [Online]. Kyoto, Japan: Kyoto University. Access: ftp://ftp.kuis.kyoto-u.ac.jp/kuis-tec-rep/94-0002.ps.gz

3. Lloyd, S. (1992). Counting binary functions with certain cryptographic properties. *Journal of Cryptology, 5*, 107-132.

4. Meier, W., & Staffelbach, O. (1990). Nonlinearity criteria for cryptographic functions. *Proceedings of EUROCRYPT '89*, 549-562. Berlin: Springer-Verlag. Edited by J.-J. Quisquater and J. Vandewalle.

5. Rothaus, O. S. (1976). On “bent" functions. *Journal of Combinatorial Theory, Series A, 20*, 300-305.

6. Skala, M. (1997, August 29). Blind substring searching. *Usenet newsgroup sci.crypt* [Online]. Access: http://www.csc.uvic.ca/~mskala/blindsub.html

7. Tyre, J. S. (1997, May 5). Letter to Brian Milburn. *Computer underground Digest, 9* (36), File 2 [Online]. Access: http://sun.soci.niu.edu/~cudigest/CUDS9/cud936

8. Webster, A. F., & Tavares, S. E. (1986). On the design of S-boxes. *Advances in Cryptology - CRYPTO '85 Proceedings*, 521-534. Berlin: Springer-Verlag. Edited by H. C. Williams.