(57) Abrégé/Abstract:
In traditional black-box cryptography, the attacker knows the algorithm, but has no visibility into its execution. For white-box cryptography, we would have to hide the key when the attacker has total visibility into its software implementation and execution.
(57) Abrégé(suite)/Abstract(continued):
(Many believe this impossible: we think there is progress but it needs more work.) This would open up a host of intriguing applications and would be likely to simplify many complex protocols. We suggest methods to make finding an encryption key harder (so far as we have yet discovered) than expected. Our purpose is similar to that of Aucsmith and Graunke's split encryption/decryption[1], but by somewhat different methods. We focus on symmetric block ciphers implemented by substitution boxes and linear transformations, with DES[2, 4] providing an example. We address efficiency only superficially, leaving full coverage for more work and a later paper. We have applied for patents on the methods we describe.
WHITE-BOX CRYPTOGRAPHY?

ABSTRACT. In traditional black-box cryptography, the attacker knows the algorithm, but has no visibility into its execution.

For white-box cryptography, we would have to hide the key when the attacker has total visibility into its software implementation and execution. (Many believe this impossible: we think there is progress but it needs more work.) This would open up a host of intriguing applications and would be likely to simplify many complex protocols.

We suggest methods to make finding an encryption key harder (so far as we have yet discovered) than expected. Our purpose is similar to that of Aucsmith and Graunke's split encryption/decryption[1], but by somewhat different methods.

We focus on symmetric block ciphers implemented by substitution boxes and linear transformations, with DES[2, 4] providing an example. We address efficiency only superficially, leaving full coverage for more work and a later paper.

We have applied for patents on the methods we describe.
WHITE-BOX CRYPTOGRAPHY?

Abstract. In traditional black-box cryptography, the attacker knows the algorithm, but has no visibility into its execution.

For white-box cryptography, we would have to hide the key when the attacker has total visibility into its software implementation and execution. (Many believe this impossible: we think there is progress but it needs more work.) This would open up a host of intriguing applications and would be likely to simplify many complex protocols.

We suggest methods to make finding an encryption key harder (so far as we have yet discovered) than expected. Our purpose is similar to that of Aucsmith and Graunke's split encryption/decryption[1], but by somewhat different methods.

We focus on symmetric block ciphers implemented by substitution boxes and linear transformations, with DES[2, 4] providing an example. We address efficiency only superficially, leaving full coverage for more work and a later paper.

We have applied for patents on the methods we describe.

1. Introduction

In traditional black-box symmetric-key cryptography, a particularly severe threat model is the adaptive chosen plaintext attack. The attacker does not know the key. However, in addition to knowing the actual encryption algorithm (or at least a reference algorithm which is functionally equivalent to it), the attacker controls how many plaintexts are encrypted, and the plaintexts themselves. And, of course, the attacker has access to the resulting ciphertexts.

The actual encryption process is not observable to the attacker, but only its result. Hence in the black-box case, it makes no difference whether the encryption is performed using the reference algorithm, or some other, functionally equivalent algorithm.

If the above threat model is to be valid, the attacker must not be able to observe the process of encryption. How much weaker the protection of an initially unknown key can then become is well demonstrated by experience with smart cards, for which a grey-box (limited visibility) threat model applies: even limited visibility of the encryption process itself severely compromises the security provided by the key.

In this paper, we make steps towards addressing a far more severe threat model. In our white-box threat model, the attacker has all of the advantages present for an adaptive chosen plaintext attack (control of the number of plaintexts and their content, and access to the resulting ciphertexts), and in addition, has full access to the software which performs the encryption. (In this paper, we deal only with the case where the software contains the encryption key in some form; as opposed to the cases where the key is presented dynamically to the software.) The attacker can arbitrarily trace execution and examine all results of sub-computations. That is, the attacker deals with full observability, rather than the partial observability enjoyed by an attacker making some form of power analysis attack on a smart card. And, of course, the attacker can also perform arbitrary static analyses on
the encryption software itself. We also assume the attacker can alter results of
sub-computation (e.g., by using breakpoints) to perform perturbation analysis.

Many believe that it is simply not possible to effectively hide an encryption
key when the attacker has such complete visibility. We believe that progress has
been made in this area, and we can expect more in the future. How strong an
implementation can be made against white-box threats remains to be seen, but
we are confident that useful levels of security — levels at which such methods can
make a real contribution to the security armamentarium — can be reached.

Our goal is similar to that of Aucsmith and Graunke’s split encryption/decryption (part of the work in [1]), but our techniques are somewhat different.

We propose ways in which the problem of finding the encryption key, even under
this extremely severe threat model, can be made combinatorially difficult for the
attacker (at least using the attacks we have discovered to date). Such methods are
inherently more (and currently, much more) costly in both space and time than
methods for black-box cryptography.

(We propose methods which can be used to produce smaller implementations of
white-box cryptographic functions. Many matters remain to be worked out; some
are partly understood but beyond the scope of one paper.)

However, there are many circumstances in which a bulky, slow implementation
may be worthwhile: a smart card, or any other alternative embodied in a piece of
hardware, cannot be transmitted over an arbitrary (sufficiently capacious) bit pipe,
such as a satellite link, nor can it, in itself, protect encryption performed within
mobile code. And, of course, the cost of transporting bit-strings, however long, is
dropping much faster than the cost of transporting hardware.

For such white-box cryptography, changing the specific implementation of the
algorithm, far from being irrelevant to security as in the black-box case, becomes
the primary means for providing security.

We focus on symmetric block ciphers where encryption requires only substitution
boxes and linear transformations, using DES[2, 4] to provide a detailed example of
hiding a key in the software. We will largely ignore space and time requirements,
leaving efficiency issues to some later paper.

We have applied for patents on the methods we describe.

2. TERMINOLOGY AND NOTATION

The term bit denotes an element of the Galois field of the integers modulo 2.
A bit-vector is a vector over this field, and a bit-matrix is a matrix over this field.
Other terms starting with "bit-" are to be similarly understood. We let the "bit-
prefix be understood wherever appropriate.

We denote by $P^e$ an encoded function derived from the function $P$.

An expression such as $(e_1, e_2, e_3, \ldots, e_k)$ denotes a vector or finite sequence of $k$
elements (the $e_i$'s). Whether the elements are bits will be evident from context.

We denote by $\mathbf{1}$ the identity function on bit-vectors of length $k$. For a function
$P$ from bit-vectors to bit-vectors, $m^k P$ denotes exactly the same function, but in
addition indicates that its domain comprises bit-vectors of length $m$, and its range,
bit-vectors of length $n$. We denote by $n^m \mathbf{E}$ (a mnemonic for an entropy-tranference
function) any function from bit-vectors of length $m$ to bit-vectors of length $n$, such
that, if $m \leq n$, its output loses no bits of information relative to its input (it is an
injection), and if $m > n$, its output loses at most $n - m$ bits of information relative
to its input. Multiple occurrences of $\mathbf{E}_m$ within a given formula or equation denote the same function.

We denote the concatenation of bit-vectors $x$ and $y$ by $x \| y$. We denote the $i$th element of a bit-vector $v$ by $v_i$ and the sub-vector containing the $i$th through $j$th elements, inclusive, by $v_{i:j}$. $\mathbf{E}_k$ denotes exactly the same bit-vector as $v$, but in addition indicates that $v$ has $k$ elements. We denote by $\mathbf{E}_k$ (a mnemonic for an entropy-vector) any vector with $k$ elements. Multiple occurrences of $\mathbf{E}_k$ within a given formula or equation denote the same vector. For bit-vectors $\mathbf{x}$ and $\mathbf{y}$, we denote by $\mathbf{x} \oplus \mathbf{y}$ the bitwise exclusive-or (XOR) of $x$ and $y$.

For bit-vector functions $\mathbf{P}$ and $\mathbf{Q}$, we denote their composition by $\mathbf{P} \circ \mathbf{Q}$ (where $a = d$), and by $\mathbf{P} \| \mathbf{Q}$ (the concatenation of $\mathbf{P}$ and $\mathbf{Q}$), that function $\mathbf{H}$ such that $H(a \| b) = P(a) \| Q(b)$, so that $H(\mathbf{a} \| \mathbf{b}) = H(\mathbf{a}) \| H(\mathbf{b})$, and $\mathbf{H}(\mathbf{a} \| \mathbf{b}) = \mathbf{H}(\mathbf{a}) \| \mathbf{H}(\mathbf{b})$ (whether $a = d$ or not). Obviously, if $\mathbf{P}$ and $\mathbf{Q}$ have inverses, $H^{-1} = P^{-1} \| Q^{-1}$, and the set of all functions is associative for both vectors and functions.

For a bit-matrix $M$, $^n_m M$ denotes exactly the same bit-matrix, but in addition, indicates that $M$ has $m$ columns and $n$ rows (so that, if we interpret the application of $M$ to a bit-vector as function application, this is the same notation as that shown for functions above).

3. **Linear Transformations, De-Linearization, and Substitution Boxes**

A linear transformation (LT) is, of course, a vector-to-vector transformation function $P$ from vectors to vectors which can be defined by $^m_n P(m \mathbf{e}) = ^m_n M \mathbf{e} + ^m_n \mathbf{d}$ for all $m \mathbf{e}$, where $M$ is a constant matrix and $\mathbf{d}$ is a constant vector.

LTs are useful in mixing and reconfiguring information. In the black-box context, the LTs can be very simple — e.g., the Expansion and P-box permutations in DES (see Figure 2. In the white-box context, however, such simple LTs cannot be used. To use LTs there, we must (1) use more complex LTs, and (2) disguise the LTs as non-linear functions. We consider below how linear transformations — such eminently serviceable components for block-box cryptography — may be strengthened for use in white-box cryptography in disguised form.

3.1. **Notes on Linear Transformations.** The “permutations” of DES are, of course, all LTs, as are its bitwise XOR operations. Hence DES is performed by LTs and substitution boxes (SBs) alone.

For a given $m$ and $n$, there are $2^{mn+n}$ LTs, but we are primarily interested in those which discard minimal, or nearly minimal, input information (i.e., not much more than $m$-n bits). If $m = n$, then there are $2^n \prod_{i=0}^{n-1} (2^n - 2^i)$ bijective LTs, since there are $\prod_{i=0}^{n-1} (2^n - 2^i)$ nonsingular $n \times n$ matrices.[5] It is the latter which are of greater significance, since we will often use LTs to reconfigure information, and changing the displacement vector, $\mathbf{d}$, of an LT, affects only the sense of the output vector elements, and not how the LT redistributes input information to the elements of its output vector.

Plainly, there are $2!$ bijections of the form $\mathbf{P}^1$. Considering the above formulas, we see that the proportion which are linear shrinks rapidly as $n$ increases. For example: there are $2! = 40,320$ bijections of the form $\mathbf{P}^2$. Out of this number, $2^3 \prod_{i=0}^{3-1} (2^n - 2^i) = 1,344$ are linear: one in 30. There are $2^{5!} = 2.631 \times 10^{35}$
bijections of the form \( \frac{5}{8} P \), of which \( 25 \prod_{i=0}^{5-1} (2^m - 2^i) = 319,979,520 \) are linear: one in about \( 8.222 \times 10^{26} \).

Nevertheless, the number of linear bijections \( \frac{5}{8} L \) becomes very large for large \( n \): for example, there are about \( 6.442 \times 10^{42} \cong 2^{142.21} \) non-singular \( 12 \times 12 \) matrices — a large number, considering that there are only \( 2^{144} 12 \times 12 \) matrices in total.

We note that if \( A \) and \( B \) are LTs, then so is \( A||B \), and so is \( A \circ B \) (where defined).

LTs, per se, are of little use in white-box cryptography, because they are so easily decomposed by Gaussian elimination and related methods.

3.2. De-Linearized LTs and Encoded Functions. Let us consider how we can de-linearize LTs and encode functions in general:

3.2.1. Partial Evaluation. Let \( \frac{5}{8} P \) be a function used in a context where its input is \( \alpha x \| \beta c \) where \( m = a + b \) and \( c \) is a constant. We can hide \( c \) by replacing \( P \) in that context with \( Q \) defined by \( \frac{5}{8} Q(\alpha x) = P(\alpha x||c) \) for all \( \alpha e \).

3.2.2. Simple Encoding. Let \( \frac{5}{8} L \) be an LT. Plainly, we can choose non-linear bijections \( \frac{5}{8} F_L \) and \( \frac{5}{8} G_L \) such that the function \( L' \) defined by \( L' = G_L \circ L \circ F_L^{-1} \) is non-linear. We call \( F_L \) the input coding, and \( G_L \) the output coding, of \( L' \) with respect to \( L \): when the input to \( L' \) is \( x \), the corresponding input to \( L' \) is \( F_L(x) \), and when the output from \( L \) is \( y \), the corresponding output from \( L' \) is \( G_L(y) \). We call \( L' \) an encoded LT. (We can only do this where \( m \) and \( n \) are sufficiently large. E.g., finding such a non-linear bijection for \( m = n < 3 \) is impossible.)

We can perform the same kind of encoding for a non-linear function \( \frac{5}{8} X \): we can define \( X' \) by \( X' = G_X \circ X \circ F_X^{-1} \) where \( F_X \) and \( G_X \) are bijections chosen so that \( X' \) is non-linear. We then call \( F_X \) and \( G_X \) the input coding and output coding of \( X' \) with respect to \( X \). And we call \( X' \) an encoded function.

3.2.3. Encoded Function Compositions. Plainly, if we compose encoded functions in order to obtain an encoding of a corresponding unencoded composition, then the input coding of the left operand must match the output coding of the right operand. E.g., to have \( L' \circ X' \) be an encoding of \( L \circ X \), we require \( F_L = G_X \), because \( L' \circ X' \) is equivalent to \( G_L \circ L \circ F_L^{-1} \circ G_X \circ X \circ F_X^{-1} \).

Variations on this theme can be used to construct encoded networks from encoded functions.

3.2.4. I/O-Blocked Encoding. In 3.2.2 above, suppose that \( m \) and \( n \) are inconveniently large when we are encoding some linear or non-linear function \( P \). (This may easily happen when using the linear blocking method of 4.1).

In that case, we can define \( F_P \) and \( G_P \) as follows:

Suppose \( m = ja \) and \( n = kb \). Let \( \frac{5}{8} J \) and \( \frac{5}{8} K \) be two 'mixing' linear bijections (two bijective LTs each of which mixes the entropy of its input bits across all of its output bits as much as possible).

We choose non-linear coding bijections \( \frac{5}{8} F_1, \ldots, \frac{5}{8} F_J \) and \( \frac{5}{8} G_1, \ldots, \frac{5}{8} G_K \). We then define \( F_P = (F_1 \| \cdots \| F_J) \circ J \) and \( G_P = (G_1 \| \cdots \| G_K) \circ K \).

Then \( P' = G_P \circ P \circ F_P^{-1} \) as usual. This permits us to connect with a 'wide I/O' linear function in encoded form, since, prior to encoding, as a preliminary step, we only need to deal with \( J \) and \( K \) (i.e., we replace \( P \) with \( K \circ P \circ J^{-1} \)), which can be done using the smaller blocking factors of the \( F_i \)'s and \( G_i \)'s which we add during encoding.
That is, if the input to $P$ is provided by an LT $X$, and the output from $P$ is used by an LT $Y$, we would use $J \circ X$ and $Y \circ K^{-1}$ instead. Then the input and output coding of the parts can ignore $J$ and $K$ — they have already been handled — and deal only with the concatenated non-linear partial I/O encodings $F_1 \| \cdots \| F_j$ and $G_1 \| \cdots \| G_k$, which conform to smaller blocking factors.

As an example of the combinatorics for such encodings, consider the case where we must encode $\frac{1}{12} P$. If we choose $a = b = 4$, then $j = k = 3$, and the number of choices for each of $F_P$ and $G_P$ is about $6 \times 10^{42}$ (non-singular $12 \times 12$ matrices) $\times 9 \times 10^{39}$ (choices for sequences of three non-linear block coding functions): about $5 \times 10^{82}$.

This easily extends to non-uniform I/O blocked encoding.

3.2.5. **Encoded Function Concatenations.** For functions $P$ and $Q$, we could choose an encoding of $P\|Q$ such as $G_{P\|Q} \circ (P\|Q) \circ F_{P\|Q}^{-1}$. This mixes $P$'s input and output entropy with $Q$'s, making it harder for an attacker to separate and determine the components $P$ and $Q$.

3.2.6. **Identity By-Pass Encoding.** For a function $n_m P$, we may choose to carry $k$ extra bits of entropy in its input and output coding to make statistical ‘bucketing’ attacks harder. (Multiplying the number of alternatives by $s$ increases the required sample count required for a given statistical precision by $s^2$, because the standard error of a statistical distribution is the square root of its standard deviation.) We could then encode $P$ as $n_{m+k} P' = G_{P\|I} \circ (P\|I) \circ F_{P\|I}^{-1}$. ($\cdot I$ is the identity by-pass component of $P'$.)

For a fixed $P$, varying the input and output codings arbitrarily, this provides the same set of functions as if we were to replace $\cdot I$ above by some arbitrary bijection $\cdot B$.

3.2.7. **General By-Pass Encoding.** In general, to carry $a$ extra bits of entropy at the input, and $b$ extra bits of entropy at the output, of $n_m P$, we can encode $n_{m+a} P'$ as $G_{P\|E} \circ (P\|E) \circ F_{P\|E}^{-1}$. ($\cdot E$ is the general by-pass component of $P'$.)

In this case, varying the input and output codings arbitrarily, if $a > b$, we cannot arbitrarily substitute the general by-pass component without effect on the set of functions obtained. Its (vector-set valued) inverse mapping implies a particular partition of the possible input subvectors for that component. We cannot entirely hide the cardinalities in the subsets forming this partition.

However, even if some statistical attack finds these cardinalities, it may yet remain quite difficult to resolve the input and output codings of $P'$.

3.2.8. **Split-Path Encoding.** For a function $n_m P$, we may encode it by encoding $n_{m+k} Q$, where we define $Q_{(m,e)} = P_{(m,e)} \| R_{(m,e)}$ for all $m,e$, for some fixed function $R$. The effect is that, if $P$ is lossy, $Q$ may be less lossy or even injective (non-lossy).

In particular, we can use this method to achieve local security, as described in 3.3.

3.2.9. **Simultaneous By-Pass Encoding.** We may need to eat our cake and have it: we may need to perform some function on an input, and yet preserve the input in the output for later use.

This can be achieved if we can arrange that our function is encoded as a bijection. Suppose, for example, that we have a bijective encoded function $P' = G_X \| X \| F_X^{-1}$ where $X$ is derived from $P$ using split-path encoding (see 3.2.8).
We can view this same function as $Q' = G_Q || Q || F_Q^{-1}$ where we define $Q = F_Q = mI$ for an appropriate $m$, and $G_Q = P'$. That is, we use an identity input encoding for an identity and regard $P'$ as the output coding. Then applying its inverse, $P'^{-1}$, to the output, we retrieve the input.

3.2.10. Output Splitting. This technique is useful for disguising outputs where input information can be well hidden. This does not appear to be the case for DES: for implementations of DES, output splitting is not recommended since it cannot provide much security.

Where the technique is appropriate, to make statistical 'bucketing' attacks more difficult, we may encode a function $P$ as $k \geq 2$ functions, $P_1, P_2, \ldots, P_k$, where each encoding can mix in additional entropy as described in 3.2.5, 3.2.6, or 3.2.7 above, and where the output of all of the encoded $P_i$'s is needed to determine the original output of $P$.

For example, given a function $\bar{P}$, we can choose $k = 2$, define $\bar{P}_1$ to be a randomly selected fixed $\bar{P}_1$, and define $\bar{P}_2(m \oplus _e) = P_1(m \oplus _e) \oplus P_2(m \oplus _e)$ for all $m \oplus _e$.

At this point, we can compute the $P$ output from the XOR of the outputs of the two $P_i$'s. However, after we then independently encode the $P_i$'s, the output of $P_1$ and $P_2$ is not combinable via an LT into information about $P$'s output.

3.3. Substitution Boxes. We can represent any function $\bar{P}$ by a substitution box (SB): a zero-origin array containing $2^n$ entries, each $n$ bits wide. To find the value of the function, we interpret the input vector as a non-negative binary number. The result is the entry of the array indexed by that number.

Plainly, the exponential growth in the size of an SB with its input width means that SBs should be used directly only to represent functions taking rather short input vectors. For example, an SB representation of a $32 \times 32$ DES "P-box" permutation would occupy about 17 gigabytes of storage.

Suppose that we use an SB to represent $L'$ (or $X'$), where these symbols have the same meaning as in 3.2.2. If $m = n$, and $L$ (or $X$) is bijective, then the SB for such an encoded bijection is locally secure: irrespective of $L$ (or $X$), for any given $n \times n$ SB without repeated entries, we can always choose an input or output coding such that the SB of $L'$ (or $X'$) is that SB. Hence, unless we know something about both the input and the output codings, any information about $L$ (or $X$), other than the value of $n$ and that it is a bijection, cannot possibly be discovered from the SB itself (which only means, of course, that an attack must be non-local).

The lossy case is not locally secure. When a slightly lossy encoded function is represented as an SB, some information about the function beyond its input and output widths can be found by examining its SB. Completely understanding it, however, still requires a non-local attack (as we will see in the DES example).

The price of using such a protean representation is that SBs are often redundant, and may include considerable non-productive entropy. For this paper we simply tolerate this; for practical implementations, we need something less bulky. Of course, any substantial implementation shrinkage compared to an SB implies restrictions on input and output codings.

Plainly, for any function $\bar{P}$ and any bijection $\bar{B}$, the SB for $P \circ B$ has the same elements as the SB for $P$, but (unless $B = mI$) in a different order.
4. WIDE-INPUT ENCODED LTs: BUILDING ENCODED NETWORKS

Because of the previously noted immense storage needs of wide-input SBs, it is infeasible to represent a wide-input encoded LT by an SB. We can, however, construct networks of SBs which implement a wide-input encoded LT.

4.1. A Blocking Method. The following construction is protean: it can handle LTs in considerable generality, including compositions of LTs, and for a wide variety of LTs of the form \( n \sigma \mathcal{L} \) encoded as \( \sigma n \mathcal{L} \), the form of the network can remain invariant except for variations in the bit patterns within its SBs.

For an LT, \( L \), we simply partition the matrix and vectors used in the LT into blocks, giving us well-known formulas using the blocks from the partition which subdivide the computation of \( L \). We can then encode the functions defined by the blocks, and combine the result into a network, using the methods in 3.2 above, so that the resulting network is an encoding of \( L \).

Consider an LT, \( L \), defined by \( n \mathcal{L} (m \mathbf{e}) = n m \mathcal{L} (m \mathbf{e}) + n d \) for all \( m \mathbf{e} \):

We choose partition counts \( m_{\#} \) and \( n_{\#} \) and sequences \( \langle m_1, \ldots, m_{m_{\#}} \rangle \) and \( \langle n_1, \ldots, n_{n_{\#}} \rangle \), such that \( \sum_i m_i = m \) and \( \sum_j n_i = n \). That is, the former sequence (the \( m \)-partition) is an additive partition of \( m \), and the latter sequence (the \( n \)-partition) is an additive partition of \( n \).

The \( m \)-partition partitions the inputs and the columns of \( M \); the \( n \)-partition partitions \( d \) and the outputs. Hence the \( i \), \( j \)th block in partitioned \( M \) contains \( m_i \) columns and \( n_j \) rows, the \( i \)th partition of the input contains \( m_i \) elements, and the \( j \)th partition of \( d \) or the output contains \( n_j \) elements.

At this point, it is straightforward to encode the components (of the network forming \( L \)) to obtain an encoded network, by the methods of 3.2, and then representing it as a network of SBs (see 3.3.) In such a network, none of the subcomputations is linear: each is encoded and represented as a non-linear SB.

A naive version of this consists of a forest of \( m_{\#} \) trees of binary ‘vector add’ SBs, with \( m_{\#}(m_{\#} - 1) \) ‘vector add’ nodes per tree. At the leaves are \( m_{\#} \) unary ‘constant vector multiply’ nodes, and at the root is either a binary ‘vector add’ node (if there is no displacement) or a unary ‘constant vector add’ node (if there is a displacement).

However, we can eliminate the unary ‘constant vector add’ and ‘constant vector multiply’ nodes entirely. We simply compose them into their adjacent binary ‘vector add’ nodes, thereby saving some space by eliminating their SBs.

A potential weakness of this entire approach is that the blocking of \( L \) may produce blocks, such as zero blocks, which convert to SBs whose output contains none, or little, of their input information. This narrows the search space for an attacker seeking to determine the underlying LT from the content and behavior of the network. However, so far as we have yet determined, such blocked implementations remain combinatorially quite difficult to crack, especially if we apply the proposals below:

To address the above potential weakness: (1) When mixing entropy by the methods of 3.2, do so opportunistically, in a manner calculated to avoid such blocks. (2) Instead of encoding \( n \mathcal{L} \), find linear \( m \mathcal{L}_1 \) and \( n \mathcal{L}_2 \), such that \( L_0 \) is a ‘mixing’ bijection (its input information is spread as much as possible across all output bits), and \( L_1 = L \circ L_2^{-1} \). Encode the two functions separately into networks of SBs, and
connect the outputs of the \( L_2 \) representation to the inputs of the \( L_1 \) representation, thus creating a representation of \( L_1 \circ L_2 = L' \).

While the above methods help, it is not easy, in general, to eliminate \( m \times n \) blocks which lose more bits of input information than the minimum indicated by \( m \) and \( n \). For example, if we partition a matrix \( \ell \times M \) into \( k \times k \) blocks, we cannot guarantee that all of the \( k \times k \) blocks are non-singular, even if the rank of \( M \) is greater than \( k \). Hence if \( M \) is non-singular, a partition of \( M \) into square blocks may contain some singular (lossy) blocks.

Therefore, some information about an encoded LT may leak in its representation as a blocked and de-linearized network of SBs when this blocking method is used.

5. Example: An Embedded, Hidden Key Implementation of DES

We now discuss a white-box implementation of DES. The implementation presented here has weaknesses, both in security and efficiency: we discuss how to address these in 5.4 and 7.

DES is performed in 16 rounds, each employing the same eight DES SBs (DSBs), \( S_1, \ldots, S_8 \), and the same LTs, sandwiched between initial and final LTs (the initial and final permutations). Each DSB is an instance of \( \mathbb{F} \). Two rounds of standard DES are shown in Figure 2.

5.1. Replacing the DES SBs. In Figure 2, we see an unrolling of two typical DES rounds. The round structure implements a Feistel network with a by-pass left-side data-path (\( L_{r-1}, L_r, L_{r+1} \) in the figure) and an active right-side data-path (everything else in the figure). \( K_r \) is the 'predicable information': the round subkey of round \( r \).

Here, we describe how we replace the DSBs with new SBs so that (1) the key is eliminated by partial evaluation (it is encoded into the new SBs; see 3.2.1), and (2) sufficient by-pass capacity is added per new SB so that all of the remaining connectivity within a round can be carried via the new SBs.

In each round, a DSB’s input is the XOR of ‘unpredictable’ information, not determined by the algorithm plus the key, and ‘predictable’ information, determined by the algorithm and the key. This predictable information can be determined in advance, without any knowledge of the unpredictable information.

Hence we can dispense with the ‘predictable’ information entirely by modifying the DSBs into new SBs. The reason is that the XOR of the ‘unpredictable’ information (the argument) with ‘predictable’ information (a constant) is a bijection (see the last paragraph in 3.3).

Let us therefore produce new SBs identified as \( K \circ S_i \), where \( K \) is the encryption key, \( r \) is the round number, and \( i \) is the corresponding DSB number, such that, for any given input, \( K \circ S_i \) yields the same result as \( S_i \) would produce in round \( r \) if the DES key were \( K \), but the XORs of the inputs of the original DSBs have been eliminated (see 3.2.1). Each of the \( 16 \times 8 = 128 \) \( K \circ S_i \)'s is still in \( \mathbb{F} \) form.

At this point, the overt key \( K \) has disappeared from the algorithm: it is represented in the contents of the \( K \circ S_i \)'s. In Figure 2, this corresponds to removing the XORs ("⊕") with the inputs to \( S_1, \ldots, S_8 \).

Next, we convert the SBs into \( \mathbb{F} \) form using split-path encoding (see 3.2.8), as shown in Figure 3 by the 8-bit data-paths entering and leaving each \( K \circ T_i \). We define \( K \circ T_i(s_e) = R(s_e) \| K \circ S_i(s_{e_{r-3}, e_r}) \) for all \( s_e \), for all \( K \), for \( r = 1, \ldots, 16 \), for \( i = 1, \ldots, 8 \), where \( R(s_e) = e_{e_1, e_8} \| (s_{e_2}, s_{e_3}, s_{e_8}) \) for all \( s_e \). This arrangement ensures
that $T_i$ is a bijection: the two outer input bits (1 and 6) of a DSB are used as high-order bits in an index to select a sub-table of the DSB containing a permutation of \( \{0, \ldots, 15\} \) (see [2]). Therefore, this arrangement cannot lose information.

This means that, for each $T_i$, the first 2 input bits are the first 2 output bits. The last 6 input bits go into the corresponding $S_i$ and yield the last 4 output bits. Bits 3 and 8 of the input become bits 3 and 4 of the output. (Again, we do not add input/output coding to these functions yet.)

At this point, if we employ simultaneous by-pass (see 3.2.9), the eight $T_i$'s in round $r$ have sufficient capacity to carry all of the information from a previous round into the next round, and each $T_i$ is locally secure (see 3.3). The use of simultaneous by-pass is shown in Figure 3 by the dual 8-bit data-paths exiting from each $T_i$.

Unfortunately, this isn’t quite enough, because the information entering the DSBs is redundant: the 32 bits of the right-side data-path of DES’s Feistel network are presented as 48 bits to the DSB inputs (by the Expansion transform; see Figure 2). Hence all of the right-side 32 bits are present and accounted for, but since $8 - 6 = 2$, the eight boxes have only 16 available bits left to carry the 32 bits of information of DES’s left-side data-path. We are short by 16 bits.

We therefore add two extra SBs, designated (pre-encoding) as $T_9$ and $T_{10}$. These are simply instances of $S_1$ prior to de-linearization and encoding, and provide the needed final 16 bits of data-path capacity.

5.2. Connecting and Encoding the New SBs to Implement DES. The over-all data-flow structure of our DES implementation immediately prior to de-linearization of LTs and encoding of SBs (see 3.2, 3.3), is shown in Figure 3. It would look just the same after de-linearization and encoding, except that each $M_i$ would be replaced by a corresponding $M_i'$ and each $T_i$ would be replaced by a corresponding $T_i'$. Except for the addition of these """ characters, it would be identical.

5.2.1. Data-Flow and Algorithm. Before de-linearization and encoding, each $M_i$ is representable as a matrix, with forms $s_6 M_1$, $s_9 M_2$, and $s_8 M_3$, respectively; we discuss them in 5.2.2.

In Figure 3, italic numbers such as 8 and 64 denote the length of the vectors traversing the data path to their left. Arrows represent data-paths and indicate their direction of data-flow.

The appearance of rows of $T_i$'s in order by $i$ in Figure 3 does not indicate any ordering of their appearance in the implementation: the intervening $M_i$ transformations can handle any such re-ordering. Let us suppose that there is a re-ordering vector $10^z$, where $z$ is a permutation of \( \{1, \ldots, 10\} \). We define $T_{10} = T_{z_1} \cdots || T_{z_{10}}$ for $r = 1, \ldots, 16$. The $T_i$'s are defined the same way, but with """ characters added to the $T_i$'s. Note that each $T_i$ or $T_i'$ is an instance of $s_6 E$.

Without the """ characters (i.e., prior to de-linearization and function encoding; see 3.2) our DES algorithm is shown in Figure 1. (We use the terms intext and outtext in the figure, rather than plaintext and ciphertext, because (1) the algorithm can be used for both encryption and decryption, and (2) in some of the scenarios discussed in section 6, neither the initial input nor the final output is a simple plaintext or ciphertext.)

The algorithm is the same, except for addition of the """ characters, after de-linearization and function encoding.
\[ v := M_1(\text{intext}) \]
\[ \text{for } r := 1, \ldots, 16 \text{ loop} \]
\[ v := M_2(\kappa^r T(v)) \]
\[ \text{end loop} \]
\[ \text{outtext} := M_3(v) \]

**Figure 1.** Modified DES Algorithm

5.2.2. *The Transfer Functions.* In constructing \( M_1 \), \( M_2 \), and \( M_3 \), we must deal with the sparseness of the matrices for the LTs used in standard DES. The bit-reorganizations, such as the **Expansion** and **P-box** transforms appearing in Figure 2, are all 0-bits except for a single 1-bit in each row and column. The XOR operations ("\( \oplus \)" in Figure 2) are similarly sparse.

Therefore, we use the second method proposed for handling sparseness in 4.1: doubling the implementations into two block implementations, with the initial portion of each pair being a 'mixing' bijection. We will regard this as part of the encoding process, and discuss the nature of the \( M_i \)'s prior to this 'anti-sparseness' treatment.

The following constructions all involve only various combinations, compositions, simple reorganizations, and concatenations of LTs, and are therefore straightforward:

\( M_1 \) combines the following: (1) the initial permutation of DES, (2) the **Expansion** (see Figure 2, modified to deliver its output bits to the last six inputs of each \( \kappa^r T_i \), combined with (3) the delivery of the left-side data-path bits to be passed through the 32 bit-by-pass provided by inputs 1 and 2 of \( \kappa^r T_1 \) through \( \kappa^r T_8 \) and the two 'dummies', \( \kappa^r T_9 \) and \( \kappa^r T_{10} \) in randomly chosen order.

\( M_2 \) combines the following: (1) the **P-box** transform (see Figure 2), (2) the XOR of the left-side data with the **P-box** output, (3) extraction of the original input of the right-side data-path using the method of 3.2.9, (4) the **Expansion**, as in \( M_1 \), (5) the left-side by-pass, as in \( M_1 \).

\( M_3 \) combines the following: (1) ignoring the inputs provided for simultaneous by-pass, (2) the left-side by-pass, as in \( M_1 \), (3) inversion of the **Expansion** by ignoring redundant bits, (4) swapping the left-side and right-side data (DES effectively swaps the left and right halves after the last round), and (5) the final permutation.

5.2.3. **Blocking Factors.** We recommend using \( 4 \times 4 \) blocking for the \( M_i \)'s. As a result of the optimization noted in 4.1, this means that the entire implementation consists entirely of networked \( 8 \times 4 \) ('vector add') and \( 8 \times 8 \) \( (\kappa^r T_i) \) SBs.

5.2.4. **Encodings.** Aside from \( M_1 \)'s input coding and \( M_3 \)'s output coding, both of which are simply \( 64 I \) (appropriately blocked), all SBs are input- and output-coded using the method of 3.2.4 in order to match the 4-bit blocking factor required for each input by the binary 'vector add' SBs.

5.3. **Complexity of Attacks on the Naïve Variant.** As we will see, in its naïve form, where \textit{intext} is \textit{plaintext} (for encryption) or \textit{ciphertext} (for decryption), or \textit{outtext} is \textit{ciphertext} (for encryption) or \textit{plaintext} (for decryption), the above implementation can be cracked quite easily using a statistical bucketing attack.

(The moral of the story is: don't use the naïve form. Use an encoded \textit{intext} and an encoded \textit{outtext}.)

10
A good way to measure the computational complexity of an attack is as the exponent of a power of two. For example, a totally naive attack on the $2^{56}$-element DES key space averages about $2^{55}$ steps: we have to try about half of the possible keys.

Attacking the input or output encodings in the above encoded DES implementation would not be an efficient approach. For a particular $4 \times 4$ output coding, there are about $2.092 \times 10^{13} \approx 2^{44.29}$ of them. For an $8 \times 8$ output coding using the method of 3.2.4 (8 $\times$ 8 non-singular matrix composed with the concatenation of two non-linear $4 \times 4$ encodings), there are about $2.341 \times 10^{45} \approx 2^{150.71}$ of them — a brute-force attack on the $2^{56}$-element DES key-space would be easier. Of course, complete knowledge is not necessary: an attacker might guess at part of the encoding, which greatly reduces the computational complexity. However, mounting an attack on such a basis would be a complex undertaking, and there is a much easier method which is quite effective.

By far the best place to attack our implementation in naive form is at points where information from the first and last rounds is available. In the first round (round 1), the initial input is known (the $M_1$ input is not coded), and in the last round (round 16), the final output is known (the $M_3$ output is not coded).

Plainly, attacks should be focussed on round 1 and round 16 (the first and final rounds). An attacker can isolate the middle rounds and consider them in isolation, but must then deal with unknown inputs and outputs. Cracking either round 1 or round 16 would provide 48 key bits; the remaining 8 bits of the 56-bit DES key can then be found by brute-force search on the 256 remaining possibilities using an ordinary DES implementation.

The attacker cannot possibly extract information from the $I^r_K T_i'$'s themselves: they are locally secure (see 3.3).

Therefore, for an attack on round 1, the attacker should focus on the $I^r_K T_i'$ inputs to round 2, and for an attack on the last round, on the $I^r_K T_i'$ outputs from round 15. This permits the attacker to deal with the input or output information after it has been broken up from (round 1), or before it has been merged into (round 16), the 8-bit bundles into and output from the $I^r_K T_i'$'s.

A statistical 'bucketing' attack is the best approach. We choose a particular DSB in standard DES, and focus on one of its round 1 output bits which is not shared by two SBs in round 2. (Each DSB has two such outputs.) We then consider the encoded round 2 inputs (1 per $I^r_K T_i$) which contain the bit. (It is only otherwise affected by the left-side data, which we hold constant, thereby freezing the outputs of the $I^r_K T_9$ and $I^r_K T_{10}$ boxes, which we then ignore.)

We know exactly which inputs each DSB takes from the sbox, so we can hold those constant while varying the other inputs relevant to the round 1 right-side data-path. This allows us to build a sample space.

Each DSB is affected by only 6 bits of the round sub-key. We hold each 6-bit data input and each 6-bit key input constant while varying the other right-side bits, and call the results we find in each of the 8 round 2 input groups (after elimination of the two irrelevant groups) the 0-sets (if the standard DES would make the bit of interest a 0-bit with that key and input to that DSB) or the 1-sets (if the standard DES would make the bit of interest a 1-bit with that key and input to that DSB). If we get a collision (an element shows up in the 0-set after it previously appeared in the 1-set) of the eight round 2 input groups, our key guess is wrong: we must try another. If we get no collisions in one of the groups, we verify with the other
unshared bit for that DSB. Once our key guess is right, we have six bits of the key. We repeat this eight times to get 48 bits of the key, and then brute-force the remaining eight bits in an average of 128 tries.

This gives a cracking complexity of 64 (6-bit inputs) \( \times \) an average of 32 (keys tried before one matches) \( \times \), say, 128 (tries before a collision is seen) \( \times \) two (unshared output bits for verification) \( \times \) eight (round 1 non-dummy \( \frac{1}{K}T_i \)'s) \( \times \) eight (input groups for the round 2 non-dummy \( \frac{2}{K}T_i \)'s) \( \sim \) \( 2^{25} \). This is better than nothing, but certainly not an acceptable cryptographic level of security.

5.4. Complexity of Attacks on the Recommended Variant. The recommended variant of the implementation uses an intext and outtext encoded as a whole using the method of 3.2.4 (or better yet, similarly encoded but in much larger blocks, with chaining). This completely foils statistical ‘bucketing’ attacks depending on the control of unencoded bits.

(This might not seem useful, but au contraire: see section 6.)

We don’t know how complex an attack on this variant is, since we have not yet found an effective way to attack it. The difficulty of cracking the individual encodings suggests that it will have a high complexity. The weakest point would seem to be the block-encoded wide-input LTs. However, it is not merely a matter of finding weak \( 4 \times 4 \) blocks (ones where an output’s entropy is reduced to three bits, say, where there are only 38,976 possible non-linear encodings). The first problem is: the output will often depend on multiple such blocks, which will then require some power of 38,976 tries. Of course, as previously noted, we may guess part of such encodings. However, we must still deal with the second, and much more difficult, problem, which is: once the attacker has a guess at a set of encodings, partial or otherwise, for certain SBs, how can it be verified? Unless there is some way to verify a guess, such an attack cannot be effective.

Whether the recommended variant is as strong as it appears at the present state of our progress remains to be seen. Only time can tell for sure.

6. HOW USEFUL IS IT?

It may not be obvious that the recommended variant (see 5.4) of our white-box DES implementation, or white-box implementations of other ciphers using the recommended variant, can be useful. We consider here how such implementations can be made to do useful work.

6.1. Handling Plain Input or Output. Although we noted that the recommended variant employed an encoded intext and outtext, we can, in fact, employ an unencoded intext or outtext with perfect safety. It just depends on how we interpret the meaning of ‘encoded’.

Suppose, for example, we take an intext which is totally ‘in the clear’ — plain English, say. If we submit it to a recommended-variant white-box encryption engine, it assumes that the plain English is an encoding of something else, and (by the time the information reaches the portion of the encryption engine corresponding to the coded implementation of the underlying cipher) the intext has become sufficiently encoded that entropy from every one of the input bits is mixed with all of the input bits. As a result, any kind of statistical ‘bucketing’ attack is foiled.

The price we pay (other than the current slowness and size of white-box implementations) is that we are no longer using a standard encryption algorithm.
However, it seems most likely to be much stronger than the original algorithm under black-box attack, and is certain to be much stronger (of course) under white-box attack.

6.2. White-Box Transaction Processing. We note that it does not take a great deal of extensions to the SB implementation to add some decision capability and flow-control and, indeed, to produce a Turing-complete set of operations, which nevertheless can be implemented in an input-output encoded fashion similar to our SBs. (The building blocks might be SBs, and networks of SBs, with added interpretive behavioral capabilities.) We can use this to perform file-updates and the like, in such a fashion that we decrypt encoded data to encoded, but usable, form, modify the information, encrypt it to encrypted and encoded form, and store it again.

So long as only small amounts of information enter or leave a transaction in plain form at unsecured sites, and almost all of the information transferred is encoded and encrypted for storage and at least encoded for update at unsecured sites, we can then perform file updates and transaction processing using white-box cryptography in such a fashion that nothing substantial ever leaves the encoded world, although some data at times is not in the encrypted world, except at secured sites. (Any substantial decrypting and decoding of information can then be reserved for sites which are very well controlled.)

This provides a means whereby software can be protected against insider attacks, particularly at sites which cannot otherwise be well protected by their owners. For example, if we fear military or industrial espionage, transaction processing as sketched above might be a significant help in managing that risk.

6.3. White-Box 'Whitening'. It is sometimes recommended to use 'pre- and post whitening' in encryption or decryption, as in DES-X[2]. We note that the recommended variant computes some cipher, based on the cipher from which it was derived, but the variant is an exceedingly inobvious one. In effect, it can serve as a form of 'super pre- and post-whitening'.

In effect, it allows us to derive innumerable new ciphers from a base cipher. All attacks on cryptography depend on some notion of the search space of functions which the cipher might compute. The white-box approach increases the search space, probably by a significant amount.

6.4. White-Box Asymmetry and Water-Mark. The effect of using the recommended variant is to convert a symmetric cipher into a one-way engine: possession of the means to encrypt in no way implies the capability to decrypt, and vice versa.

This means that we can give out very specific communication capabilities to control communication patterns by giving out specific encryption and decryption engines to particular parties. By using double encryption or decryption based on a pass phrase, we can arrange that changing the communication patterns requires both a communicated pass phrase and a communicated encryption or decryption engine. And of course, every such engine is effectively water-marked by the function it computes.

It is also possible to identify a piece of information by the fact that a particular decryption engine decrypts it to a known form. There are many variations on this theme.
7. Conclusions and Future Work

We have presented the concept of white-box cryptography, and methods by which DES-like cryptographic computations may well be able to withstand white-box attacks. We have argued that white-box cryptography has the potential to contribute significant new capabilities to the security armamentarium.

That said, it is certainly the case that many improvements, as well as extensions to other kinds of cryptographic computations, remain to be found! (We have begun work on RSA-like computations, for example.)

We must seek ways to provide metrics for the security of white-box implementations. (Hence, for example, the properties of encoded linear and non-linear computations in networks of various topologies must be much better understood.)

For DES-like ciphers, the blocking technique in 4.1, while general, is of $O(m\#n\#)$ complexity for a fixed bound on SB size. We must explore the potential of other network topologies, such as shuffle-exchange topologies (especially irregular ones, for security). Where $m = n$, for example, these have the potential to reduce the spatial complexity for representing LTs to $O(n\#\log n\#)$ for a fixed bound on SB size.

We must seek ways other than SBs of representing non-linear functions. Much of the entropy in SBs is not productive of security, despite its spatial cost. We could probably reduce our choice of non-linear functions greatly without compromising security.

We must seek to implement practical ‘general’ capabilities as suggested in 6.2. Aside from the use already mentioned, this will help us to prevent attackers from using the white-box encryption or decryption engine itself as a particularly large one-way key.

And we must certainly seek in every way to integrate white-box cryptographic capabilities with general methods of tamper-proofing software; i.e., to add white-box protection to software other than cryptographic computation.

References

While particular embodiments of the present invention have been shown and described, it is clear that changes and modifications may be made to such embodiments without departing from the true scope and spirit of the invention.

It is understood that as de-compiling and debugging tools become more and more powerful, the degree to which the techniques of the invention must be applied to ensure tamper protection, will also rise. As well, the concern for system resources may also be reduced over time as the cost and speed of computer execution and memory storage capacity continue to improve.

These improvements will also increase the attacker's ability to overcome the simpler tamper-resistance techniques included in the scope of the claims. It is understood, therefore, that the utility of some of the simpler encoding techniques that fall within the scope of the claims, may correspondingly decrease over time. That is, just as in the world of cryptography, increasing key-lengths become necessary over time in order to provide a given level of protection, so in the world of the instant invention, increasing complexity of encoding will become necessary to achieve a given level of protection.

The method steps of the invention may be embodiment in sets of executable machine code stored in a variety of formats such as object code or source code. Such code is described generically herein as programming code, or a computer program for simplification. Clearly, the executable machine code may be integrated with the code of other programs, implemented as subroutines, by external program calls or by other techniques as known in the art.

The embodiments of the invention may be executed by a computer processor or similar device programmed in the manner of method steps, or may be executed by an electronic system which is provided with means for executing these steps. Similarly, an electronic memory medium may be programmed to execute such method steps. Suitable memory media would include serial access formats such as magnetic tape, or random access formats such as floppy disks, hard drives, computer diskettes, CD-Roms, bubble memory, EEPROM, Random Access Memory (RAM), Read Only Memory (ROM) or similar computer software storage media known in the art. Furthermore, electronic signals representing these method steps may also be transmitted via a communication network.

It will be obvious to one skilled in these arts that there are many practical embodiments of the DES implementation produced by the instant invention, whether in
normal executable machine code, code for a virtual machine, or code for a special
purpose interpreter. It would also be possible to directly embed the invention in a net-list
for the production of a pure hardware implementation, that is, an ASIC.

It would also be clear to one skilled in the art that this invention need not be
limited to the existing scope of computers and computer systems. Credit, debit, bank
and smart cards could be encoded to apply the invention to their respective applications.
An electronic commerce system in a manner of the invention could for example, be
applied to parking meters, vending machines, pay telephones, inventory control or rental
cars and using magnetic strips or electronic circuits to store the software and
passwords. Again, such implementations would be clear to one skilled in the art, and do
not take away from the invention.
WHAT IS CLAIMED IS:

1. A method of obscuring computer software comprising the step of: de-linearizing functions in said computer software.

2. A method of obscuring computer software comprising the step of: partially evaluating functions in said computer software.

3. A method of obscuring computer software comprising the step of: substituting functions in said computer software with suitable non-linear bijections.

4. A method of obscuring computer software comprising the step of: performing complementary encoding of operands of functions in said computer software.

5. A method of obscuring computer software comprising the step of: substituting functions in said computer software with suitable non-linear input/output (I/O) blocked bijections of said functions.

6. A method of obscuring computer software comprising the step of: concatenating functions in said computer software with one another.

7. A method of obscuring computer software comprising the step of: concatenating functions in said computer software with one another, and with other functions.


10. A method of obscuring computer software comprising the step of: split-path encoding of functions in said computer software.
11. A method of obscuring computer software comprising the step of: encoding functions in said computer software as bijections.


15. A method of obscuring computer software comprising the step of: replacing functions in said computer software with networks of substitution boxes.

16. A system for executing the method of any one of claims 1 through 15.

17. An apparatus for executing the method of any one of claims 1 through 15.

18. A computer readable memory medium for storing software code executable to perform the method steps of any one of claims 1 through 15.

19. A carrier signal incorporating software code executable to perform the method steps of any one of claims 1 through 15.
Fig. 2 Two rounds of DES

Fig. 3 Modified DES
before De-Linearization and Encoding
Two rounds of DES