A White-Box DES Implementation for DRM Applications*

S. Chow[1], P. Eisen[1], H. Johnson[1], P.C. van Oorschot[2]

[1] Cloakware Corporation, Ottawa, Canada
[2] School of Computer Science, Carleton University, Ottawa, Canada

Abstract. For digital rights management (DRM) software implementations incorporating cryptography, *white-box* cryptography (cryptographic implementation designed to withstand the *white-box attack context*) is more appropriate than traditional *black-box* cryptography. In the white-box context, the attacker has total visibility into software implementation and execution. Our objective is to prevent extraction of secret keys from the program. We present methods to make such key extraction difficult, with focus on symmetric block ciphers implemented by substitution boxes and linear transformations. A DES implementation (useful also for triple-DES) is presented as a concrete example.

1 Introduction

In typical software digital rights management (DRM) implementations, cryptographic algorithms are part of the security solution. However, the traditional cryptographic model — employing a strong known algorithm, and relying on the secrecy of the cryptographic key — is inappropriate surprisingly often, since the platforms on which many DRM applications execute are subject to the control of a potentially hostile end-user. This is the challenge we seek to address.

A traditional threat model used in *black-box* symmetric-key cryptography is the adaptive chosen plaintext attack model. It assumes the attacker does not know the encryption key, but knows the algorithm, controls the plaintexts encrypted (their number and content), and has access to the resulting ciphertexts. However, the dynamic encryption operation is hidden — the attacker has no visibility into its execution.

We make steps towards providing software cryptographic solutions suitable in the more realistic (for DRM applications) *white-box* attack context: the attacker is assumed to have all the advantages of an adaptive chosen-text attack, plus full access to the encrypting software and control of the execution environment. This includes arbitrary trace execution, examining sub-results and keys in memory, performing arbitrary static analyses on the software, and altering results of sub-computation (e.g. via breakpoints) for perturbation analysis.

* Version dated 13 Jan. 2003 for: Proceedings of 2nd ACM workshop on Digital Rights Management — ACM CCS-9 Workshop DRM 2002 (Springer-Verlag LNCS to appear).

Our main goal is to make key extraction difficult. While an attacker controlling the execution environment can clearly make use of the software itself (e.g. for decryption) without explicitly extracting the key, forcing an attacker to use the installed instance at hand is often of value to DRM systems providers. How strong an implementation can be made against white-box threats is unknown. We presently have no security proofs for our methods. Nonetheless, regardless of the security of our particular proposal, we believe the general approach offers *useful* levels of security in the form of additional protection suitable in the commercial world, forcing an attacker to expend additional effort (compared to conventional black-box implementations). Our goal is similar to Aucsmith and Graunke's split encryption/decryption [1]; the solutions differ.

White-box solutions are inherently (and currently, quite significantly) bulkier and slower than black-box cryptography. These drawbacks are offset by advantages justifying white-box solutions in certain applications. Software-only white-box key-hiding components may be cost-effectively installed and updated periodically (cf. Jakobsson and Reiter [8]), whereas smart cards and hardware alternatives can't be transmitted electronically. Hardware solutions also cannot protect encryption within mobile code. While white-box implementations are clearly not appropriate for all cryptographic applications (see [4]), over time, we expect increases in processing power, memory capacity and transmission bandwidth, along with decreasing costs, to ameliorate the efficiency concerns.

In black-box cryptography, differences in implementation details among functionally equivalent instances are generally irrelevant with respect to security implications. In contrast, for white-box cryptography, changing implementation details becomes a *primary* means for providing security. (This is also true, to a lesser extent, for cryptographic solutions implemented on smart cards and environments subject to so-called side-channel attacks.)

In this paper, we focus on general techniques that are useful in producing white-box implementations of Feistel ciphers. We use DES (e.g. see [11]) to provide a detailed example of hiding a key in the software. DES-like ciphers are challenging in the white-box context since each round leaves half the bits unchanged and the expansions, permutations and substitution boxes are very simple (and known). We propose techniques to handle these problems.

We largely ignore space and time requirements in the present paper, noting only that white-box implementations have been successfully used in commercial practice. In the present paper we restrict attention to the embedded (fixed) key case; dynamic-key white-box cryptography is the subject of ongoing research. The motivation for using DES is twofold: (1) DES needs only linear transformations and substitution boxes, simplifying our discussion; and (2) our technique readily extends to triple-DES which remains popular. We outline a white-box implementation for AES [5] elsewhere — see Chow et al. [4], to which we also refer for further discussion of the goals of white-box cryptography, related literature, and why theoretical results such as that of Barak et al. [2] are not roadblocks to practical solutions.

Following terminology and notation in §2, §3 outlines basic white-box construction techniques. §4 presents a blocking method for building encoded networks. §5 provides an example white-box DES implementation, with a recommended variant discussed in §5.3. Concluding remarks are found in §6.

2 Terminology and Notation

We follow the terminology of Chow et al. [4]. A major concept used is the encoding of a transformation. In our work, examples of transformations include a *substitution-box* (S-box or lookup table) as well as the overall DES function. Input/output encodings are used to protect these transformations as follows.

Definition 1 (encoding) *Let X be a transformation from m to n bits. Choose an m-bit bijection F and an n-bit bijection G. Call $X' = G \circ X \circ F^{-1}$ an* encoded *version of X. F is an* input encoding *and G is an* output encoding.

$\langle v_1, v_2, v_3, \ldots, v_k \rangle$ is a k-vector with elements v_i; context indicates whether elements are bits. v_i is the ith element; $v_{i\cdots j}$ is the sub-vector containing elements i through j. ${}_kv$ denotes explicitly that v has k elements. ${}_k\mathbf{e}$ is any vector with k elements (mnemonically: an *entropy*-vector); ${}_k\mathbf{e}_i$ is its ith element, and ${}_k\mathbf{e}_{i\cdots j}$ is the subvector from its ith to its jth element. $x\|y$ is the *vector concatenation* of vectors x, y. $x \oplus y$ denotes their bitwise *xor*.

Transformations may have wide inputs and/or outputs (in the DES construction, some are 96 bits input and output). To avoid huge tables, we construct encodings as the concatenation of smaller bijections. Consider bijections F_i of size n_i, where $n_1 + n_2 + \ldots + n_k = n$. Having used $\|$ for vector concatenation, we analogously use $\|$ for *function concatenation* as follows.

Definition 2 (concatenated encoding) *The* function concatenation *$F_1\|F_2\| \ldots \|F_k$ is the bijection F such that, for any n-bit vector $b = (b_1, b_2, \ldots, b_n)$, $F(b) = F_1(b_1, \ldots, b_{n_1})\|F_2(b_{n_1+1}, \ldots, b_{n_1+n_2})\| \ldots \|F_k(b_{n_1+\ldots+n_{k-1}+1}, \ldots, b_n)$. For such a bijection F, plainly $F^{-1} = F_1^{-1}\|F_2^{-1}\| \ldots \|F_k^{-1}$. Such an encoding F is called a* concatenated encoding.

Generally, output of a transformation will become the input to another subsequent transformation, which means the output encoding of the first must match the input encoding of the second as follows.

Definition 3 (networked encoding) *A* networked encoding *for computing $Y \circ X$ (i.e. transformation X followed by transformation Y) is an encoding of the form: $Y' \circ X' = (H \circ Y \circ G^{-1}) \circ (G \circ X \circ F^{-1}) = H \circ (Y \circ X) \circ F^{-1}$.*

P' denotes an encoded implementation derived from function P. To emphasize that P maps m-vectors to n-vectors, we write ${}^n_mP$. For a matrix M, ${}^n_mM$ indicates that M has m columns and n rows. (These notations naturally correspond, taking application of M to a vector as function application.)

${}^n_m\mathbf{E}$ (mnemonic: *entropy*-transfer function) is any function from m-vectors to n-vectors which loses no bits of information for $m \leq n$ and at most $m - n$ bits for $m > n$. A function ${}^n_n f$ which is *not* an instance of ${}^n_n\mathbf{E}$ is *lossy*.

An *affine transformation* (AT) is a vector-to-vector function V defined for all ${}_m\mathbf{e}$ by ${}^n_m V({}_m\mathbf{e}) = {}^n_m M_m \mathbf{e} + {}_n d$ (concisely: $V(\mathbf{e}) = M\mathbf{e} + d$). M is a constant matrix, and d a constant *displacement vector*, over GF(2). If A and B are ATs, then so are $A \| B$ and $A \circ B$ where defined.

3 Producing Encoded Implementations

DES consists of permutations, S-box lookups and *xor* operations, as is well known (e.g. [11]). Our approach is to apply encodings to each of these steps. For S-box lookups and *xor* operations, encoding each operation (along with its input and output) seems to increase security adequately within our context. For the various permutations (bit re-orderings), the problem is more difficult.

As these permutations are, by nature, very simple, it is difficult to hide the information being manipulated. To access more tools, we find it convenient to change the domain from bit re-orderings to linear algebra. We first express each of the DES permutations and bitwise *xor* operations as ATs. While the resulting ATs are still very simple and fail to hide information well, the idea is that subsequent use of non-linear encoding (see §4) significantly changes the situation.

3.1 Techniques for Tabularizing Functions

We produce implementations of conventional ciphers as networks of substitution boxes (lookup tables). Since ATs are easy to compose or decompose, we obfuscate even subnetworks representing affine subcomputations by using non-affine substitution boxes. In this section we describe several building-blocks useful for such implementations. We will use all of these except **Combined Function Encoding** in our DES example.

Partial Evaluation If part of the input to P is known at implementation creation time, we can simply input the known values to P' and pre-evaluate all constant expressions. For example, in the fixed-key case where the key is known in advance, pre-evaluate all operations involving the key. For DES this essentially means replacing the standard S-boxes with round-subkey-specific S-boxes.

Mixing Bijections We diffuse information over multiple bits as follows.

Definition 4 (mixing bijection) *A* mixing bijection ${}^n_n V$ *is a randomly chosen* $n \times n$ *bijective* AT.

In DES, for example, the permutations, represented as ATs, have very sparse matrices (i.e., contain mostly zero entries): one or two 1-bits per row or column. To diffuse information over more bits, rewrite such a permutation P as $J \circ K$

where K is a mixing bijection and $J = PK^{-1}$, replacing a sparse matrix by two non-sparse ones with high probability. This is advantageous in subsequent de-linearizing encoding steps (see §4).

I/O-Blocked Encoding For large m, encoding an arbitrary function ${}^{n}_{m}P$ as a substitution box for $P' = G \circ P \circ F^{-1}$ takes too much space (box size varies exponentially with m). For large n, the same problem arises for P''s successors. We must therefore divide P's input into a-bit blocks ($m = ja$), and its output into b-bit blocks ($n = kb$). Let ${}^{m}_{m}J$ and ${}^{n}_{n}K$ be mixing bijections. Randomly choose encoding bijections for each input and output block: ${}^{a}_{a}F_1, \ldots, {}^{a}_{a}F_j$ and ${}^{b}_{b}G_1, \ldots, {}^{b}_{b}G_k$. Define $F_P = (F_1 \| \cdots \| F_j) \circ J$ and $G_P = (G_1 \| \cdots \| G_k) \circ K$, and then $P' = G_P \circ P \circ F_P^{-1}$ as usual. (See §4 for methods used to represent wide-input ATs such as J, K above by networks of substitution boxes.)

This permits us to use networked encoding (def. 3) with a 'wide I/O' linear function in encoded form, because as a preliminary step before encoding, we need only deal with J and K (i.e., we replace P by $K \circ P \circ J^{-1}$), using the smaller blocking factors of the F_i and G_i. That is, if the input to P is provided by an AT X, and the output from P is used by an AT Y, we 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 easily handled by substitution boxes. This easily extends to non-uniform I/O blocked encoding (where blocks vary in size).

Combined Function Encoding For functions P and Q that happen to be evaluated together, we could choose an encoding of $P \| Q$ such as $G \circ (P \| Q) \circ F^{-1}$. Essentially, we combine P and Q into a single function, then encode the combined input and output. The encoding mixes P's input and output entropy with Q's, ideally making it harder for an attacker to determine the components P and Q. Note that this differs from concatenated encoding (def. 2) in how the encoding is applied. Here, the encoding applies to all components as a single unit.

By-Pass Encoding Generally, an encoded transform implementation should have a wider input and/or output than the function it implements, to make transform identification difficult. For example, for ${}^{n}_{m}P$ to have a extra bits at input and b extra bits at output, $a \geq b$, encode ${}^{n+b}_{m+a}P'$ as $G \circ (P \| {}^{b}_{a}\mathbf{E}) \circ F^{-1}$. ${}^{b}_{a}\mathbf{E}$ is the *by-pass* component of P'.

Split-Path Encoding To encode a function ${}^{n}_{m}P$, use a concatenation of two separate encodings: for a fixed function R and all ${}_{m}\mathbf{e}$, define ${}^{n+k}_{m}Q({}_{m}\mathbf{e}) = P({}_{m}\mathbf{e}) \| {}^{k}_{m}R({}_{m}\mathbf{e})$. The effect is that, if P is lossy, Q may lose less (or no) information. We sometimes use this technique to achieve *local security* (see §3.2.)

3.2 Substitution Boxes and Local Security

We can represent a function ${}^n_m P$ by a substitution box (S-box) or lookup table: an array of 2^m n-bit entries. To compute $P(x)$, find the array entry indexed by the binary magnitude x. The exponential growth in S-box size with its input width limits S-boxes to the representation of narrow input functions.

When the underlying P is bijective, the encoded S-box for P' is *locally secure*: it is not possible to extract useful information by examining the encoded S-box alone, since given an S-box for P', every possible bijective P is a candidate. (This is similar to a Vernam cipher $c = m \oplus k$, where given ciphertext c, every plaintext is a candidate m because for each, some key k exists whose *xor* with m yields c.) This means only that successful attacks must be non-local.

The lossy case is not locally secure. When a lossy encoded function f is represented as an S-box, its inverse relation f^{-1} relates each output element to a *set* of bit-vectors, thus locally partitioning f's domain. Leaking this partition can provide enough information to allow subsequent non-local attacks such as the one in the **Statistical Bucketing Attack** subsection of §5.4.

4 Wide-Input Encoded ATs: Building Encoded Networks

Constructing an S-box with wide-input, say 96 bits (or even 32), consumes immense amounts of storage. Thus in practice, a wide-input encoded AT cannot be represented by a single S-box. *Networks* of S-boxes, however, can be constructed to do so. The following construction handles ATs in considerable generality, including compositions of ATs, and for a wide variety of ATs of the form ${}^n_m A$ encoded as ${}^n_m A'$. A network's form can remain invariant aside from variations in the bit patterns within its S-boxes.

For an AT A, we partition the matrix and vectors into blocks, yielding well-known formulas using the blocks from the partition which subdivide the computation of A. We can then use (smaller) S-boxes to encode the functions defined by the blocks, and combine the result into a network using techniques from §3.1, so that the resulting network is an encoding of A.

Consider an AT A, defined by ${}^n_m A({}_m\mathbf{e}) = {}^n_m M\, {}_m\mathbf{e} + {}_n d$ for all ${}_m\mathbf{e}$. 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 $\sum_1^{m_\#} m_i = m$ and $\sum_1^{n_\#} n_i = n$. The m-partition partitions the inputs (and columns of M); the n-partition partitions d and the outputs. Block (i, j) in partitioned M contains m_i columns and n_j rows; partition i of the input contains m_i elements; and partition j of d or the output contains n_j elements.

At this point, it is straightforward to encode the components (of the network forming A) to obtain an encoded network, by the methods of §3.1, and then represent it as a network of S-boxes (see §3.2.) In such a network, *no* subcomputations are linear; each is encoded and represented as a non-linear S-box.

A naive version of this network of S-boxes is a forest of $n_\#$ trees of binary 'vector add' S-boxes ($m_\#(m_\# - 1)$ 'vector add' nodes per tree). At the leaves are $m_\#$ unary 'constant vector multiply' nodes. At the root is a binary 'vector add' node (for no displacement), or a unary 'constant vector add' node. These constant unary nodes can be optimized away by composing them into their adjacent binary 'vector add' nodes, saving the space for their S-boxes.

A potential weakness of this entire approach is that the blocking of A may produce blocks (e.g. zero blocks) which convert to S-boxes whose output contains none, or little, of their input information. This narrows the search space for an attacker seeking to determine the underlying AT from the content and behavior of the network. However, such blocked implementations appear to remain combinatorially difficult to crack, especially if the following proposal is used.

ADDRESSING THE POTENTIAL WEAKNESS. Encode ${}^{n}_{m}A$ *via* ${}^{n}_{m}A_1$ and ${}^{m}_{m}A_2$, with mixing bijection (see def. 4) A_2 and $A_1 = A \circ A_2^{-1}$. Encode A_1, A_2 separately into S-box networks using this matrix and vector blocking method, connecting outputs of A_2''s representation to inputs of A_1''s, thus representing $A' = A_1' \circ A_2'$.

While this helps, in general it is not easy 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 non-singular matrix ${}^{kn}_{kn}M$ into $k \times k$ blocks, some $k \times k$ blocks may be singular. Therefore, *some* information about an encoded AT may leak in its representation as a blocked and de-linearized network of S-boxes when this blocking method is used.

5 A White-Box DES Implementation Example

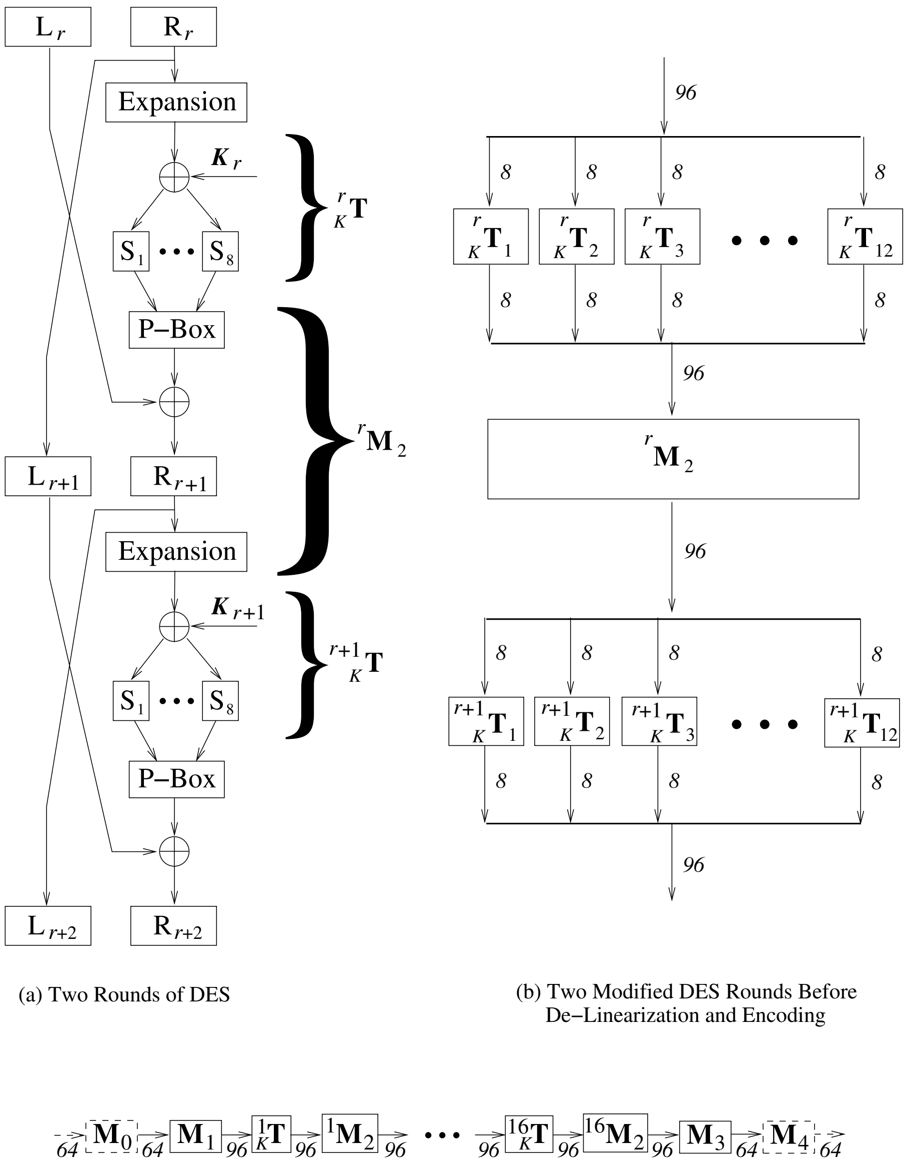
We now construct an embedded, fixed-key DES implementation. We begin with a simple construction having weaknesses, in both security and efficiency. These are addressed in §5.3.

DES is performed in 16 rounds, each employing the same 8 DES S-boxes (DSBs), $\mathbf{S}_1, \ldots \mathbf{S}_8$, and the same ATs, sandwiched between initial and final ATs (the initial and final permutations). Each DSB is an instance of ${}^{4}_{6}\mathbf{E}$ (see e.g. [11]). Fig. 1(a) shows an unrolling of 2 DES rounds. The round structure implements a Feistel network with a by-pass left-side data-path ($\mathbf{L}_r$, $\mathbf{L}_{r+1}$, $\mathbf{L}_{r+2}$) and active right-side data-path (everything else in the figure). $\mathbf{K}_r$ is the round-r subkey.

5.1 Replacing the DES SBs

Fig. 1(b) shows the modified implementation of the two rounds. Each round is represented by 12 '**T**-boxes' (see **Preparing**... below). (Each such group of 12 is denoted by an ${}^{r}_{K}\mathbf{T}$ in Fig. 1(c).) Between rounds, the left and right sides are combined into one 96-bit representation. Each round's ${}^{r}\mathbf{M}_2$ transform subsumes the P-Box, round-key *xor*, side flip and **Expansion** after the round-r S-box step (for details, see **The Transfer Functions** in §5.2).

As shown in Fig. 1(c), a transform $\mathbf{M}_1$ is needed for an initial input expansion from 64 to 96 bits. Likewise a transform $\mathbf{M}_3$ is needed to reduce the final output size. ($\mathbf{M}_0$ and $\mathbf{M}_4$ are discussed in §5.3: **Recommended Variant**.)



(a) Two Rounds of DES

(b) Two Modified DES Rounds Before De−Linearization and Encoding

(c) Modified DES Before De−Linearization and Encoding

Fig. 1. Original and Modified DES

Eliminating the Overt Key by Partial Evaluation In each round, a DSB's input is the *xor* of 'unpredictable' information (i.e. data), and 'predictable' information (from the algorithm and the key). We can merge the 'predictable' information and the DSBs into new S-boxes dependent on the key and round. The new S-boxes are identified as ${}^{r}_{K}\mathbf{S}_i$. Here K is the encryption key, r is the round number, and i is the corresponding DSB number, such that, for any given input, ${}^{r}_{K}\mathbf{S}_i$ yields the same result as $\mathbf{S}_i$ would produce in round r if the DES key were K, but the *xor*s of the inputs of the original DSBs have been eliminated (see **Partial Evaluation** in §3.1). Each of the $16 \times 8 = 128$ ${}^{r}_{K}\mathbf{S}_i$'s is still in ${}^{4}_{6}\mathbf{E}$ form (6 input bits, 4 output bits).

At this point, the overt key K has disappeared from the algorithm: it is represented in the contents of the ${}^{r}_{K}\mathbf{S}_i$'s. This permits us to remove the *xor*s ("$\oplus$") with the inputs to $\mathbf{S}_1, \ldots, \mathbf{S}_8$ shown in Fig. 1(a).

Preparing the Modified DSBs for Local Security In *grey-box* (smart card) implementations of DES, the DSBs are now known to be effective sites for statistical attacks. To make such attacks more difficult in a white-box implementation, we prefer to employ S-boxes which are locally secure (see §3.2). This implies replacing lossy S-boxes with something bijective. We convert the lossy ${}^{r}_{K}\mathbf{S}_i$'s into ${}^{8}_{8}\mathbf{E}$ form using split-path encoding (see §3.1) as follows. Define

$${}^{r}_{K}\mathbf{T}_i({}_8\mathbf{e}) \;=\; {}^{r}_{K}\mathbf{S}_i({}_8\mathbf{e}_{1..6}) \| \; R({}_8\mathbf{e})$$

for all ${}_8\mathbf{e}$, fixed key K, rounds $r = 1, \ldots, 16$, and S-box number $i = 1, \ldots, 8$. Here we also define $R({}_8\mathbf{e}) = \langle\, {}_8\mathbf{e}_1,\, {}_8\mathbf{e}_6,\, {}_8\mathbf{e}_7,\, {}_8\mathbf{e}_8 \,\rangle$ for all ${}_8\mathbf{e}$.

The first six bits of the input of a ${}^{r}_{K}\mathbf{T}_i$ will be the 6-bit input to DSB i in round r. We then add two extra input bits. The left 4-bit half of the output of a ${}^{r}_{K}\mathbf{T}_i$ is the output of DSB i in round r, and the right 4-bit half contains the first and last input bits of DSB i in round r followed by the two extra input bits. That is, the right half of the output contains copies of four of the input bits.

Each ${}^{r}_{K}\mathbf{T}_i$ is a bijection, as the function $F_{a,b,c,d}$ defined for any constant bits a, b, c, d by $F_{a,b,c,d}({}_4\mathbf{e}) = {}^{r}_{K}\mathbf{T}_i(\langle a \rangle \| {}_4\mathbf{e} \| \langle b, c, d \rangle)$ is a bijection. (Every row of every DSB contains a permutation of $\langle 0, \ldots, 15 \rangle$, with the row selected by the bits corresponding to a, b above. The *xor* with the relevant bits of key K effectively re-orders this permutation into a new one. The output of $F_{a,b,c,d}$ is therefore a bijection mapping the ${}_4\mathbf{e}$ according to a 1-to-1 mapping of the input space determined by a permutation. Since ${}^{r}_{K}\mathbf{T}_i$ simply copies the bits corresponding to a, b, c, d to the output, ${}^{r}_{K}\mathbf{T}_i$ preserves *all* of its input entropy, i.e. is a bijection.)

Providing 64 bits of By-Pass Capacity In our construction, we wish to hide the difference between the left and right Feistel data-path sides, so each ${}^{r}\mathbf{M}_2$ expects more than just 32 bits of S-box outputs. Both the left and (unchanged) right sides are needed. We refer to this as needing 64 bits of by-pass.

As converted above, each ${}^{r}_{K}\mathbf{T}_i$ carries 8 bits to the next ${}^{r}\mathbf{M}_2$: 4 bits of S-box output, 2 bits from the right side and 2 bits that can be chosen to be from the left. This means 8 T-boxes will carry only 16 bits from the left and 16 from the right. Thus the by-pass capacity of the ${}^r_K\mathbf{T}_i$'s is deficient by 32 bits.

Therefore we add four more S-boxes per round, designated ${}^r_K\mathbf{T}_9, \ldots, {}^r_K\mathbf{T}_{12}$. Each is a bijective AT of 8 bits to 8 bits. These extra S-boxes are AT's to make it easier to access the bypassed bits for subsequent processing. (Subsequent steps will de-linearize *every* S-box, so use of ATs for these by-pass paths need not compromise security.) These extra S-boxes provide the remaining 32 bits, 16 bits each of right-side and left-side by-pass capacity.

5.2 Connecting and Encoding the New SBs to Implement DES

Data-flow for our DES implementation just before AT de-linearization and S-box encoding (§3.1, §3.2) is shown in Figs. 1(b,c). After de-linearization and encoding, $\mathbf{M}_0$ and $\mathbf{M}_4$ are composed with their diagrammatically adjacent transforms and all $\mathbf{M}$'s and $\mathbf{T}$'s are replaced with corresponding $\mathbf{M}'$'s and $\mathbf{T}'$'s. Except for this composition and addition of "$'$" characters (indicating de-linearized, encoded functionality, including, where required, the 'anti-sparseness' treatment in **The Transfer Functions** below), the figures are unchanged.

Data-Flow and Algorithm Before de-linearization and encoding, each $\mathbf{M}_i$ or ${}^r\mathbf{M}_i$ is representable as a matrix, with forms ${}^{96}_{64}\mathbf{M}_1$, ${}^{64}_{96}\mathbf{M}_3$, and, for each round's ${}^r\mathbf{M}_2$, ${}^{96}_{96}\mathbf{M}_2$. (See §5.1, and for more details **The Transfer Functions** below.)

In Figs. 1(b,c), arrows represent data-paths and indicate their direction of data-flow. The italic numbers *8*, *64*, and *96* denote the length of the vectors traversing the data path arrow next to them. The appearance of rows of ${}^r_K\mathbf{T}_i$'s in order by i in Fig. 1(b) does not indicate any ordering of their appearance in the implementation. The intervening ${}^r\mathbf{M}_2$ transformations can handle any such re-ordering.

The Transfer Functions In constructing $\mathbf{M}_1$, ${}^r\mathbf{M}_2$'s, and $\mathbf{M}_3$, we must deal with the sparseness of the matrices for the ATs used in standard DES. The bit-reorganizations, such as the **Expansion** and **P-box** transforms in Fig. 1(a), are all 0-bits except for one or two 1-bits in each row and column. The *xor* operations ("$\oplus$" in Fig. 1(a)) are similarly sparse. Therefore, we use the method proposed for handling sparseness in §4's ADDRESSING THE POTENTIAL WEAKNESS: doubling the implementations into two blocked 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 $\mathbf{M}_i$'s prior to this 'anti-sparseness' treatment.

The following constructions are straightforward, all involving only various combinations, compositions, simple reorganizations, and concatenations of ATs.

$\mathbf{M}_1$ combines the following: (1) the initial permutation of DES; (2) the **Expansion** (see Fig. 1(a)), modified to deliver its output bits to the first six inputs of each ${}^1_K\mathbf{T}_i$; combined with (3) the delivery of the 32 left-side data-path bits to be passed through the by-pass provided by inputs 7 and 8 of ${}^1_K\mathbf{T}_1, \ldots, {}^1_K\mathbf{T}_8$ and

16 bits of by-pass provided at randomly chosen positions in the four 'dummies', ${}^{1}_{K}\mathbf{T}_9, \ldots, {}^{1}_{K}\mathbf{T}_{12}$, all in randomly chosen order.

${}^{r}\mathbf{M}_2$ for each round r combines the following: (1) the **P-box** transform (see Fig. 1(a)); (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; (4) the round's **Expansion** (which was provided by $\mathbf{M}_1$ for the first round); and (5) the left-side by-pass (provided by $\mathbf{M}_1$ for the first round).

$\mathbf{M}_3$ combines the following: (1) ignoring the inputs provided for simultaneous by-pass; (2) the left-side by-pass (provided by $\mathbf{M}_1$ and $\mathbf{M}_2$ for the previous rounds); (3) inversion of the **Expansion**, ignoring half of each redundant bit pair; (4) swapping the left- and right-side data (DES effectively swaps the left and right halves after the last round); and (5) the final permutation.

Blocking and Encoding Details We recommend 4×4 blocking for the $\mathbf{M}_i$'s. As a result of the optimization noted in §4, this means the implementation consists entirely of networked 8×4 ('vector add') and 8×8 (${}^{r}_{K}\mathbf{T}'_i$) S-boxes.

Aside from $\mathbf{M}_1$'s input coding and $\mathbf{M}_3$'s output coding, both of which are simply 64×64 identities (appropriately blocked), all S-boxes are input- and output-coded using the method of §3.1 in order to match the 4-bit blocking factor required for each input by the binary 'vector add' S-boxes.

5.3 Recommended Variant

The above section completes a *naked* variant of white-box DES. The *recommended* variant applies input and output encodings to the whole DES operation. Referring to Fig. 1(c), we modify our scheme so that $\mathbf{M}_1$ is replaced by $\mathbf{M}_1 \circ \mathbf{M}_0$ and $\mathbf{M}_3$ is replaced by $\mathbf{M}_4 \circ \mathbf{M}_3$, where the $\mathbf{M}_0$ and $\mathbf{M}_4$ ATs are ${}^{64}_{64}\mathbf{E}$ mixing bijections. As part of our encoding, we combine $\mathbf{M}_1 \circ \mathbf{M}_0$ and $\mathbf{M}_4 \circ \mathbf{M}_3$ into single ATs. When encoded in 4-bit blocks, they become non-linear.

One issue that arises is whether this recommended variant of DES (or other ciphers) is still an implementation of the standard algorithm. Although it employs an encoded input and output, we can pre- and post-process the input to this computation by the inverses of the pre- and post-encodings, to effectively cancel both. One might refer to this as operating on *de*-encoded *intext* and *outtext*. The de-encoding process can be done in any one or a combination of several places, for example: the software immediately surrounding the cryptographic computation; more distant surrounding software; or ideally, software executing on a separate node (with obvious coordination required). The pre- and post-encoding itself can be folded into the component operations of the standard algorithm, e.g., DES, as explained under I/O-Blocked Encoding per §3.1. Taking into account the de-encodings, the overall result is again equivalent to the standard algorithm.

The overall result is a data transformation which embeds DES. By embedding the standard algorithm within a larger computation we retain the (black-box) strength of the original algorithm within this embedded portion (which does implement the standard algorithm). Furthermore the encompassing computation provides greater resistance to white-box attacks. By using pre- and post-encodings that are bijections, we have in effect composed 3 bijections.

White-Box 'Whitening'. It is sometimes recommended to use 'pre- and post whitening' in encryption or decryption, as in Rivest's DESX [9]. We note that the recommended variant computes *some* cipher, based on the cipher from which it was derived, but the variant is far from obvious. In effect, it serves as an aggressive form of pre- and post-whitening, and allows us to derive innumerable new ciphers from a base cipher. Essentially all cryptographic attacks depend on some notion of the search space of functions which the cipher might compute. The white-box approach increases the search space.

White-Box Asymmetry and Watermarking. The recommended variant has additional advantages. 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. Every such engine is also effectively watermarked (fingerprinted) by the function it computes, and it is possible to identify a piece of information by the fact that a particular decryption engine decrypts it to a known form.

5.4 Attacks on Naked Variant DES Implementation

The attacker cannot extract information from the ${}^{r}_{K}\mathbf{T}'_i$'s themselves: they are *locally secure* (see §3.2). Consequently all attacks must be global in the sense of having to look at multiple S-boxes and somehow correlate the information. We know of no efficient attacks on the recommended variant.

By far the best place to attack the naked variant of our implementation seems to be at points where information from the first and last rounds is available. In round 1, the initial input is known (the $\mathbf{M}_1$ input is not coded), and in round 16, the final output is known (the $\mathbf{M}_3$ output is not coded). Both known attacks (see below) on the naked variant exploit this weak point.

The Jacob Attack on the Naked Variant The attack of Jacob et al. [7] is a clever DFA-like [3] attack, inducing a controlled fault by taking advantage of the unchanged data in the Feistel structure, thus bypassing much of the protection afforded by the encodings. However it requires that the input (or output) be naked (i.e., unencoded), and simultaneous access to a key-matched pair of encrypt and decrypt programs, a situation unlikely with an actual DRM application using white-box DES. It is not obvious how to relax either of these requirements. It is also not clear how this attack can apply to ciphers that are not Feistel-like.

Statistical Bucketing Attack on Naked Variant This attack is somewhat similar to the DPA attacks [10]. In the DPA attacks, keys are guessed and differences in power profiles are used to confirm or deny the guesses. Our statistical bucketing attack also involves guessing keys, but guesses are confirmed or denied by checking if buckets are disjoint.

Attacks should be focussed on the first and final rounds. Cracking either round 1 or round 16 provides 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 a reference DES implementation. For ease of explanation, we discuss only attacking round 1 of the encryption case.

Consider S-box ${}^1\mathbf{S}_i$ in round 1 of standard DES. Its 6 bits of input come directly from the input plaintext, and it is affected by 6 bits of round 1 sub-key. Its output bits go to different DSBs in round 2 (with an intervening *xor* operation). We focus on one of these output bits, which we denote b. ${}^2\mathbf{S}_j$ will refer to (one of) the round 2 DSBs affected by b. That is, we pick ${}^1\mathbf{S}_i$ in round 1 which produces bit b, which is then consumed by ${}^2\mathbf{S}_j$ in round 2. Potentially, bit b can go to two different S-boxes in round 2 (either one will suffice).

Make a guess on the 6 bits of sub-key affecting ${}^1\mathbf{S}_i$, run through the 64 inputs to it, and construct 64 corresponding plaintexts. The plaintexts must feed the correct bits into ${}^1\mathbf{S}_i$ as well as the *xor* operation involving b. For convenience, fix the left side to all zeros. This effectively nullifies the *xor* operations. The other 26 bits in the plaintexts should be chosen randomly for each plaintext. Using any reference implementation of DES, divide these 64 plaintexts into two buckets, I_0, I_1, which have the property that if the key guess is correct, bit b will have a value of 0 for the encryption of each plaintext in the I_0 set; similarly, for each plaintext in the I_1 set, if the guess is correct, b will have a value of 1.

Next take these two buckets of plaintexts and run them through the encoded implementation. Since the implementation is naked, one can easily track the data-flow to discover which ${}^2\mathbf{T}_{z_j}$ encodes ${}^2\mathbf{S}_j$. Examine the input to ${}^2\mathbf{T}_{z_j}$ to confirm or deny the guess. The encryption of the texts in I_0 (resp. I_1) will lead to a set of inputs I'_0 (resp. I'_1) to ${}^2\mathbf{T}_{z_j}$. The important point is that if the key guess is correct, I'_0 and I'_1 must necessarily be disjoint sets. Any overlap indicates that the guess is wrong. If no overlap occurs, the key guess may or may not be correct: this may happen simply by chance. (The likelihood of this happening is minimized when the aforementioned 26 bits of right hand side plaintext are chosen randomly.) To ensure the effectiveness of this technique, we would like the probability that no *collision* (an element occurring in both I'_0 and I'_1) occurs in the event of an incorrect key guess to be at most 2^{-6}. Experimentally, this occurs when $|I_0| = |I_1| \approx 27$ — 54 chosen plaintexts in all — so the 64 plaintexts mentioned above are normally adequate.

The above description works on one S-box at a time. We can work on the 8 S-boxes of a round in parallel, as follows. Due to the structure of the permutations of DES, output bits $\{3, 7, 11, 15, 18, 24, 28, 30\}$ have the property that each bit comes from a unique S-box and goes to a unique S-box in the following round. By tracking these bits, we can search for the sub-key affecting each round 1 DSB in parallel (this requires a clever choice of elements for I_0 and I_1, because of the overlap in the inputs to the round 1 DSBs). Experimentation shows that fewer than 2^7 plaintexts are necessary in total to identify a very small set of candidates for the 48-bit round 1 subkey. The remaining 8 bits of key can subsequently be determined by exhaustive search.

This gives a cracking complexity of 128 (chosen plaintexts) $\times$ 64 (number of 6 bit sub-keys) + 256 (remaining 8 bits of key) $\approx 2^{13}$ encryptions. This attack on the naked variant has been implemented, and it successfully finds the key in under 10 seconds.

5.5 Comments on Security of the Recommended Variant

While we are aware of no effective attack on the recommended variant, we also have no security proofs. The assumed difficulty of cracking the individual encodings leads us to believe the attack complexity will be high. The weakest point appears to be the block-encoded wide-input ATs. We note it is not merely a matter of finding weak 4×4 blocks (ones where an output's entropy is reduced to 3 bits, say, where there are only 38,976 possible non-linear encodings). The first problem is that the output will often depend on multiple such blocks, which will then require some power of 38,976 tries. Of course, as previously noted, *part* of such encodings may be guessed. However, the second, apparently much more difficult problem, is that once the attacker has a guess at a set of encodings, partial or otherwise, for certain S-boxes, how can it be verified? Unless there is some way to verify a guess, it appears such an attack cannot be effective.

Whether the recommended variant herein is reasonably strong or not remains to be seen. However, even should the answer be negative for this particular variant, we believe the general approach remains promising, due to the many variations possible using the multiplicity of approaches discussed.

5.6 Supplementary Notes on Cardinality of Transformations

For a given m and n, there are 2^{mn+n} m-input, n-output ATs, but we are primarily interested in those which discard minimal, or nearly minimal, input information — not much more than $m - n$ bits (cf. *lossy* in §2 and *locally secure* in §3.2). If $m = n$, then there are $2^n \prod_{i=0}^{n-1}(2^n - 2^i)$ bijective ATs, since there are $\prod_{i=0}^{n-1}(2^n - 2^i)$ nonsingular $n \times n$ matrices [6]. It is the latter figure which is of greater significance, since we will often use ATs to reconfigure information, and changing the displacement vector, d, of an AT, can at most invert selected output vector bits: it can't affect the AT's redistribution of input information to the elements of its output vector.

We note that while the number of bijective ATs is a tiny fraction of all bijections of the form ${}^n_nP$ (there being $2^n!$ of them), the absolute number of bijective ATs nonetheless is very large for large n. This ensures a large selection space of bijective ATs which we use, e.g. for pre- and post-encodings.

6 Concluding Remarks

For DES-like algorithms, we have presented building blocks for constructing implementations which increase resistance to white-box attacks, and as an example proposed a white-box DES implementation. The greatest drawbacks to our approach are size and speed, and as is common in new cryptographic proposals, the lack of both security metrics and proofs. Our techniques (though not using DES itself) are in use in commercial products, and we expect to see increased use of white-box cryptography in DRM applications as their deployment in hostile environments (including the threat of end-users) drives the requirement for stronger protection mechanisms within cryptographic implementations. While the current paper addresses fixed-key symmetric algorithms, ongoing research includes extensions of white-box ideas to the dynamic-key case, and to public-key algorithms such as RSA.

References

1. D. Aucsmith, G. Graunke, *Tamper-Resistant Methods and Apparatus*, U.S. Patent No. 5,892,899, 1999.
2. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang, *On the (Im)possibility of Obfuscating Programs*, pp. 1–18 in: Advances in Cryptology — Crypto 2001 (LNCS 2139), Springer-Verlag, 2001.
3. Eli Biham, Adi Shamir, *Differential Fault Analysis of Secret Key Cryptosystems*, pp. 513–525, Advances in Cryptology — Crypto '97 (LNCS 1294), Springer-Verlag, 1997. *Revised*: Technion – Computer Science Department – Technical Report CS0910-revised, 1997.
4. S. Chow, P. Eisen, H. Johnson, P.C. van Oorschot, *White-Box Cryptography and an AES Implementation*, Proceedings of the Ninth Workshop on Selected Areas in Cryptography (SAC 2002), August 15–16, 2002 (Springer-Verlag LNCS, to appear).
5. J. Daemen, V. Rijmen, *The Design of Rijndael: AES — The Advanced Encryption Standard*, Springer-Verlag, 2001.
6. Leonard E. Dickson, *Linear Groups, with an Exposition of Galois Field Theory*, p. 77, Dover Publications, New York, 1958.
7. M. Jacob, D. Boneh, E. Felten, *Attacking an obfuscated cipher by injecting faults*, proceedings of 2nd ACM workshop on Digital Rights Management — ACM CCS-9 Workshop DRM 2002 (Springer-Verlag LNCS to appear).
8. M. Jakobsson, M.K. Reiter, *Discouraging Software Piracy Using Software Aging*, pp.1–12 in: Security and Privacy in Digital Rights Management — ACM CCS-8 Workshop DRM 2001 (LNCS 2320), Springer-Verlag, 2002.
9. J. Kilian, P. Rogaway, *How to protect DES against exhaustive key search*, pp.252–267 in: Advances in Cryptology — Crypto '96, Springer-Verlag LNCS, 1996.
10. Paul Kocher, Joshua Jaffe, Benjamin Jun, *Differential Power Analysis*, pp. 388–397, Advances in Cryptology — Crypto '99 (LNCS 1666), Springer-Verlag, 1999.
11. A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of Applied Cryptography*, pp. 250–259, CRC Press, 2001 (5th printing with corrections). Down-loadable from `http://www.cacr.math.uwaterloo.ca/hac/`